Software That Sees

2nd Edition
Covers OpenCV 2.5

Early Release

Learning

OpenCV

Computer Vision in C++ with the OpenCV Library

O'REILLY®

Adrian Kaehler & Gary Bradski

Preface

This book provides a working guide to the Open Source Computer Vision Library (OpenCV) and also provides a general background to the field of computer vision sufficient to use OpenCV effectively.

Purpose

Computer vision is a rapidly growing field, partly as a result of both cheaper and more capable cameras, partly because of affordable processing power, and partly because vision algorithms are starting to mature. OpenCV itself has played a role in the growth of computer vision by enabling thousands of people to do more productive work in vision. With its focus on real-time vision, OpenCV helps students and professionals efficiently implement projects and jump-start research by providing them with a computer vision and machine learning infrastructure that was previously available only in a few mature research labs. The purpose of this text is to:

- Better document OpenCV—detail what function calling conventions really mean and how to use them correctly.
- Rapidly give the reader an intuitive understanding of how the vision algorithms work.
- Give the reader some sense of what algorithm to use and when to use it.
- Give the reader a boost in implementing computer vision and machine learning algorithms by providing many working coded examples to start from.
- Provide intuitions about how to fix some of the more advanced routines when something goes wrong.

Simply put, this is the text the authors wished we had in school and the coding reference book we wished we had at work.

This book documents a tool kit, OpenCV, that allows the reader to do interesting and fun things rapidly in computer vision. It gives an intuitive understanding as to how the algorithms work, which serves to guide the reader in designing and debugging vision applications and also to make the formal descriptions of computer vision and machine learning algorithms in other texts easier to comprehend and remember.

After all, it is easier to understand complex algorithms and their associated math when you start with an intuitive grasp of how those algorithms work.

Who This Book Is For

This book contains descriptions, working coded examples, and explanations of the computer vision tools contained in the OpenCV library. As such, it should be helpful to many different kinds of users.

Professionals

For those practicing professionals who need to rapidly implement computer vision systems, the sample code provides a quick framework with which to start. Our descriptions of the intuitions behind the algorithms can quickly teach or remind the reader how they work.

Students

As we said, this is the text we wish had back in school. The intuitive explanations, detailed documentation, and sample code will allow you to boot up faster in computer vision, work on more interesting class projects, and ultimately contribute new research to the field.

Teachers

Computer vision is a fast-moving field. We've found it effective to have the students rapidly cover an accessible text while the instructor fills in formal exposition where needed and supplements with current papers or guest lectures from experts. The students can meanwhile start class projects earlier and attempt more ambitious tasks.

Hobbyists

Computer vision is fun, here's how to hack it.

We have a strong focus on giving readers enough intuition, documentation, and working code to enable rapid implementation of real-time vision applications.

What This Book Is Not

This book is not a formal text. We do go into mathematical detail at various points,[1] but it is all in the service of developing deeper intuitions behind the algorithms or to clarify the implications of any assumptions built into those algorithms. We have not attempted a formal mathematical exposition here and might even incur some wrath along the way from those who do write formal expositions.

This book is not for theoreticians because it has more of an "applied" nature. The book will certainly be of general help, but is not aimed at any of the specialized niches in computer vision (e.g., medical imaging or remote sensing analysis).

That said, it is the belief of the authors that having read the explanations here first, a student will not only learn the theory better but remember it longer. Therefore, this book would make a good adjunct text to a theoretical course and would be a great text for an introductory or project-centric course.

About the Programs in This Book

All the program examples in this book are based on OpenCV version 2.5. The code should definitely work under Linux or Windows and probably under OS-X, too. Source code for the examples in the book can be fetched from this book's website (http://www.oreilly.com/catalog/9780596516130). OpenCV can be loaded from its source forge site (http://sourceforge.net/projects/opencvlibrary).

OpenCV is under ongoing development, with official releases occurring once or twice a year. To keep up to date with the developments of the library, and for pointers to where to get the very latest updates and versions, you can visit OpenCV.org, the library's official website.

Prerequisites

For the most part, readers need only know how to program in C and perhaps some C++. Many of the math sections are optional and are labeled as such. The mathematics involves simple algebra and basic matrix

[1] Always with a warning to more casual users that they may skip such sections.

algebra, and it assumes some familiarity with solution methods to least-squares optimization problems as well as some basic knowledge of Gaussian distributions, Bayes' law, and derivatives of simple functions.

The math is in support of developing intuition for the algorithms. The reader may skip the *math* and the algorithm descriptions, using only the function definitions and code examples to get vision applications up and running.

How This Book Is Best Used

This text need not be read in order. It can serve as a kind of user manual: look up the function when you need it; read the function's description if you want the gist of how it works "under the hood". The intent of this book is more tutorial, however. It gives you a basic understanding of computer vision along with details of how and when to use selected algorithms.

This book was written to allow its use as an adjunct or as a primary textbook for an undergraduate or graduate course in computer vision. The basic strategy with this method is for students to read the book for a rapid overview and then supplement that reading with more formal sections in other textbooks and with papers in the field. There are exercises at the end of each chapter to help test the student's knowledge and to develop further intuitions.

You could approach this text in any of the following ways.

Grab Bag

> Go through Chapter 1–Chapter 3 in the first sitting, then just hit the appropriate chapters or sections as you need them. This book does not have to be read in sequence, except for Chapter 11 and Chapter 12 (Calibration and Stereo).

Good Progress

> Read just two chapters a week until you've covered Chapter 1–Chapter 12 in six weeks (Chapter 13 is a special case, as discussed shortly). Start on projects and dive into details on selected areas in the field, using additional texts and papers as appropriate.

The Sprint

> Just cruise through the book as fast as your comprehension allows, covering Chapter 1–Chapter 12. Then get started on projects and go into details on selected areas in the field using additional texts and papers. This is probably the choice for professionals, but it might also suit a more advanced computer vision course.

Chapter 13 is a long chapter that gives a general background to machine learning in addition to details behind the machine learning algorithms implemented in OpenCV and how to use them. Of course, machine learning is integral to object recognition and a big part of computer vision, but it's a field worthy of its own book. Professionals should find this text a suitable launching point for further explorations of the literature—or for just getting down to business with the code in that part of the library. This chapter should probably be considered optional for a typical computer vision class.

This is how the authors like to teach computer vision: Sprint through the course content at a level where the students get the gist of how things work; then get students started on meaningful class projects while the instructor supplies depth and formal rigor in selected areas by drawing from other texts or papers in the field. This same method works for quarter, semester, or two-term classes. Students can get quickly up and running with a general understanding of their vision task and working code to match. As they begin more challenging and time-consuming projects, the instructor helps them develop and debug complex systems. For longer courses, the projects themselves can become instructional in terms of project management. Build up working systems first; refine them with more knowledge, detail, and research later. The goal in such courses is for each project to aim at being worthy of a conference publication and with a few project papers being published subsequent to further (postcourse) work.

Conventions Used in This Book

The following typographical conventions are used in this book:

Italic

Indicates new terms, URLs, email addresses, filenames, file extensions, path names, directories, and Unix utilities.

`Constant width`

Indicates commands, options, switches, variables, attributes, keys, functions, types, classes, namespaces, methods, modules, properties, parameters, values, objects, events, event handlers, XMLtags, HTMLtags, the contents of files, or the output from commands.

`Constant width bold`

Shows commands or other text that could be typed literally by the user. Also used for emphasis in code samples.

`Constant width italic`

Shows text that should be replaced with user-supplied values.

[…]

Indicates a reference to the bibliography. The standard bibliographic form we adopt in this book is the use of the last name of the first author of a paper, followed by a two digit representation of the year of publication. Thus the paper "Self-supervised monocular road detection in desert terrain," authored by "H. Dahlkamp, A. Kaehler, D. Stavens, S. Thrun, and G. Bradski" in 2006, would be cited as: "[Dahlkamp06]".

This icon signifies a tip, suggestion, or general note.

This icon indicates a warning or caution.

Using Code Examples

OpenCV is free for commercial or research use, and we have the same policy on the code examples in the book. Use them at will for homework, for research, or for commercial products. We would very much appreciate referencing this book when you do, but it is not required. Other than how it helped with your homework projects (which is best kept a secret), we would like to hear how you are using computer vision for academic research, teaching courses, and in commercial products when you do use OpenCV to help you. Again, not required, but you are always invited to drop us a line.

Safari® Books Online

When you see a Safari® Books Online icon on the cover of your favorite technology book, that means the book is available online through the O'Reilly Network Safari Bookshelf.

Safari offers a solution that's better than e-books. It's virtual library that lets you easily search thousands of top tech books, cut and paste code samples, download chapters, and find quick answers when you need the most accurate, current information. Try it for free at http://safari.oreilly.com.

We'd Like to Hear from You

Please address comments and questions concerning this book to the publisher:

O'Reilly Media, Inc.

1005 Gravenstein Highway North

Sebastopol, CA 95472

800-998-9938 (in the United States or Canada)

707-829-0515 (international or local)

707-829-0104 (fax)

We have a web page for this book, where we list examples and any plans for future editions. You can access this information at:

http://www.oreilly.com/catalog/9780596516130/

You can also send messages electronically. To be put on the mailing list or request a catalog, send an email to:

info@oreilly.com

To comment on the book, send an email to:

bookquestions@oreilly.com

For more information about our books, conferences, Resource Centers, and the O'Reilly Network, see our website at:

http://www.oreilly.com

Acknowledgments

A long-term open source effort sees many people come and go, each contributing in different ways. The list of contributors to this library is far too long to list here, but see the *…/opencv/docs/HTML/Contributors/doc_contributors.html* file that ships with OpenCV.

Thanks for Help on OpenCV

Intel is where the library was born and deserves great thanks for supporting this project the whole way through. Open source needs a champion and enough development support in the beginning to achieve critical mass. Intel gave it both. There are not many other companies where one could have started and maintained such a project through good times and bad. Along the way, OpenCV helped give rise to—and now takes (optional) advantage of—Intel's Integrated Performance Primitives, which are hand-tuned assembly language routines in vision, signal processing, speech, linear algebra, and more. Thus the lives of a great commercial product and an open source product are intertwined.

Mark Holler, a research manager at Intel, allowed OpenCV to get started by knowingly turning a blind eye to the inordinate amount of time being spent on an unofficial project back in the library's earliest days. As divine reward, he now grows wine up in Napa's Mt. Veeder area. Stuart Taylor in the Performance Libraries group at Intel enabled OpenCV by letting us "borrow" part of his Russian software team. Richard Wirt was key to its continued growth and survival. As the first author took on management responsibility at Intel, lab director Bob Liang let OpenCV thrive; when Justin Rattner became CTO, we were able to put OpenCV on a more firm foundation under Software Technology Lab—supported by software guru Shinn-Horng Lee and indirectly under his manager, Paul Wiley. Omid Moghadam helped advertise OpenCV in the early days. Mohammad Haghighat and Bill Butera were great as technical sounding boards. Nuriel Amir, Denver Dash, John Mark Agosta, and Marzia Polito were of key assistance in launching the machine learning library. Rainer Lienhart, Jean-Yves Bouguet, Radek Grzeszczuk, and Ara Nefian were able technical contributors to OpenCV and great colleagues along the way; the first is now a professor, the second is now making use of OpenCV in some well-known Google projects, and the others are staffing research labs and start-ups. There were many other technical contributors too numerous to name.

On the software side, some individuals stand out for special mention, especially on the Russian software team. Chief among these is the Russian lead programmer Vadim Pisarevsky, who developed large parts of the library and also managed and nurtured the library through the lean times when boom had turned to bust; he, if anyone, is the true hero of the library. His technical insights have also been of great help during the writing of this book. Giving him managerial support and protection in the lean years was Valery Kuriakin, a man of great talent and intellect. Victor Eruhimov was there in the beginning and stayed through most of it. We thank Boris Chudinovich for all of the contour components.

Finally, very special thanks go to Willow Garage [WG], not only for its steady financial backing to OpenCV's future development but also for supporting one author (and providing the other with snacks and beverages) during the final period of writing this book.

Thanks for Help on the Book

While preparing this book, we had several key people contributing advice, reviews, and suggestions. Thanks to John Markoff, Technology Reporter at the *New York Times* for encouragement, key contacts, and general writing advice born of years in the trenches. To our reviewers, a special thanks go to Evgeniy Bart, physics postdoc at CalTech, who made many helpful comments on every chapter; Kjerstin Williams at Applied Minds, who did detailed proofs and verification until the end; John Hsu at Willow Garage, who went through all the example code; and Vadim Pisarevsky, who read each chapter in detail, proofed the function calls and the code, and also provided several coding examples. There were many other partial reviewers. Jean-Yves Bouguet at Google was of great help in discussions on the calibration and stereo chapters. Professor Andrew Ng at Stanford University provided useful early critiques of the machine learning chapter. There were numerous other reviewers for various chapters—our thanks to all of them. Of course, any errors result from our own ignorance or misunderstanding, not from the advice we received.

Finally, many thanks go to our editor, Michael Loukides, for his early support, numerous edits, and continued enthusiasm over the long haul.

Contributors:

Vadim Pisarevsky

Reviewers:

Kari Pulli

Kjerstin Williams

Evgeniy Bart

Professor Andrew Ng

GPU Appendix:

Khanh Yun-Ta

Anatoly Baksheev

Editors etc:

Michael Loukides

Rachel B. Steely

Rachel Roumeliotis

Adrian Adds…

Coming from a background in theoretical physics, the arc that brought me through supercomputer design and numerical computing on to machine learning and computer vision has been a long one. Along the way, many individuals stand out as key contributors. I have had many wonderful teachers, some formal instructors and others informal guides. I should single out Professor David Dorfan of UC Santa Cruz and Hartmut Sadrozinski of SLAC for their encouragement in the beginning, and Norman Christ for teaching me the fine art of computing with the simple edict that "if you cannot make the computer do it, you don't know what you are talking about". Special thanks go to James Guzzo, who let me spend time on this sort of thing at Intel—even though it was miles from what I was supposed to be doing—and who encouraged my participation in the Grand Challenge during those years. Finally, I want to thank Danny Hillis for creating the kind of place where all of this technology can make the leap to wizardry and for encouraging my work on the book while at Applied Minds. Such unique institutions are rare indeed in the world.

I also would like to thank Stanford University for the extraordinary amount of support I have received from them over the years. From my work on the Grand Challenge team with Sebastian Thrun to the STAIR Robot with Andrew Ng, the Stanford AI Lab was always generous with office space, financial support, and most importantly ideas, enlightening conversation, and (when needed) simple instruction on so many aspects of vision, robotics, and machine learning. I have a deep gratitude to these people, who have contributed so significantly to my own growth and learning.

No acknowledgment or thanks would be meaningful without a special thanks to my family, who never once faltered in their encouragement of this project or in their willingness to accompany me on trips up and down the state to work with Gary on this book. My thanks and my love go to them.

Gary Adds…

With three young kids at home, my wife Sonya put in more work to enable this book than I did. Deep thanks and love—even OpenCV gives her recognition, as you can see in the face detection section example image. Further back, my technical beginnings started with the physics department at the University of Oregon followed by undergraduate years at UC Berkeley. For graduate school, I'd like to thank my advisor Steve Grossberg and Gail Carpenter at the Center for Adaptive Systems, Boston University, where I first cut my academic teeth. Though they focus on mathematical modeling of the brain and I have ended up firmly on the engineering side of AI, I think the perspectives I developed there have made all the difference. Some of my former colleagues in graduate school are still close friends and gave advice, support, and even some editing of the book: thanks to Frank Guenther, Andrew Worth, Steve Lehar, Dan Cruthirds, Allen Gove, and Krishna Govindarajan.

I specially thank Stanford University, where I'm currently a consulting professor in the AI and Robotics lab. Having close contact with the best minds in the world definitely rubs off, and working with Sebastian Thrun and Mike Montemerlo to apply OpenCV on Stanley (the robot that won the $2M DARPA Grand Challenge) and with Andrew Ng on STAIR (one of the most advanced personal robots) was more technological fun than a person has a right to have. It's a department that is currently hitting on all cylinders and simply a great environment to be in. In addition to Sebastian Thrun and Andrew Ng there, I thank Daphne Koller for setting high scientific standards, and also for letting me hire away some key interns and students, as well as Kunle Olukotun and Christos Kozyrakis for many discussions and joint work. I also thank Oussama Khatib, whose work on control and manipulation has inspired my current interests in visually guided robotic manipulation. Horst Haussecker at Intel Research was a great colleague to have, and his own experience in writing a book helped inspire my effort.

Finally, thanks once again to Willow Garage for allowing me to pursue my lifelong robotic dreams in a great environment featuring world-class talent while also supporting my time on this book and supporting OpenCV itself.

1

Overview

What Is OpenCV?

OpenCV [OpenCV] is an open source (see http://opensource.org) computer vision library available from http://opencv.org. The library is written in C and C++[1] and runs under Linux, Windows, Mac OS X, iOS, and Android. Interfaces are available for Python, Java, Ruby, Matlab, and other languages.

OpenCV was designed for computational efficiency with a strong focus on real-time applications: optimizations were made at all levels, from algorithms to multicore and CPU instructions. For example, OpenCV supports optimizations for SSE, MMX, AVX, NEON, OpenMP, and TBB. If you desire further optimization on Intel architectures [Intel] for basic image processing, you can buy Intel's Integrated Performance Primitives (IPP) libraries [IPP], which consist of low-level optimized routines in many different algorithmic areas. OpenCV automatically uses the appropriate instructions from IPP at runtime. The GPU module also provides CUDA-accelerated versions of many routines (for Nvidia GPUs) and OpenCL-optimized ones (for generic GPUs).

One of OpenCV's goals is to provide a simple-to-use computer vision infrastructure that helps people build fairly sophisticated vision applications quickly. The OpenCV library contains over 500 functions that span many areas, including factory product inspection, medical imaging, security, user interface, camera calibration, stereo vision, and robotics. Because computer vision and machine learning often go hand-in-hand, OpenCV also contains a full, general-purpose Machine Learning Library (MLL). This sub-library is focused on statistical pattern recognition and clustering. The MLL is highly useful for the vision tasks that are at the core of OpenCV's mission, but it is general enough to be used for any machine learning problem.

Who Uses OpenCV?

Most computer scientists and practical programmers are aware of some facet of the role that computer vision plays. But few people are aware of all the ways in which computer vision is used. For example, most people are somewhat aware of its use in surveillance, and many also know that it is increasingly being used for images and video on the Web. A few have seen some use of computer vision in game interfaces. Yet few people realize that most aerial and street-map images (such as in Google's Street View) make heavy use of camera calibration and image stitching techniques. Some are aware of niche applications in safety monitoring, unmanned aerial vehicles, or biomedical analysis. But few are aware how pervasive machine vision has become in manufacturing: virtually everything that is mass-produced has been automatically inspected at some point using computer vision.

---

[1] The legacy C interface is still supported, and will remain so for the foreseeable future.

The BSD [BSD] open source license for OpenCV has been structured such that you can build a commercial product using all or part of OpenCV. You are under no obligation to open-source your product or to return improvements to the public domain, though we hope you will. In part because of these liberal licensing terms, there is a large user community that includes people from major companies (Google, IBM, Intel, Microsoft, Nvidia, SONY, and Siemens, to name only a few) and research centers (such as Stanford, MIT, CMU, Cambridge, Georgia Tech and INRIA). OpenCV is also present on the web for users at http://opencv.org, a website that hosts documentation, developer information, and other community resources including links to compiled binaries for various platforms. For vision developers, code, development notes and links to GitHub are at http://code.opencv.org. User questions are answered at http://answers.opencv.org/questions/ but there is still the original Yahoo groups user forum at http://groups.yahoo.com/group/OpenCV; it has almost 50,000 members. OpenCV is popular around the world, with large user communities in China, Japan, Russia, Europe, and Israel. OpenCV has a Facebook page at https://www.facebook.com/opencvlibrary.

Since its alpha release in January 1999, OpenCV has been used in many applications, products, and research efforts. These applications include stitching images together in satellite and web maps, image scan alignment, medical image noise reduction, object analysis, security and intrusion detection systems, automatic monitoring and safety systems, manufacturing inspection systems, camera calibration, military applications, and unmanned aerial, ground, and underwater vehicles. It has even been used in sound and music recognition, where vision recognition techniques are applied to sound spectrogram images. OpenCV was a key part of the vision system in the robot from Stanford, "Stanley", which won the $2M DARPA Grand Challenge desert robot race [Thrun06], and continues to play an important part in other many robotics challenges.

What Is Computer Vision?

Computer vision[2] is the transformation of data from 2D/3D stills or videos into either a decision or a new representation. All such transformations are done for achieving some particular goal. The input data may include some contextual information such as "the camera is mounted in a car" or "laser range finder indicates an object is 1 meter away". The decision might be "there is a person in this scene" or "there are 14 tumor cells on this slide". A new representation might mean turning a color image into a grayscale image or removing camera motion from an image sequence.

Because we are such visual creatures, it is easy to be fooled into thinking that computer vision tasks are easy. How hard can it be to find, say, a car when you are staring at it in an image? Your initial intuitions can be quite misleading. The human brain divides the vision signal into many channels that stream different pieces of information into your brain. Your brain has an attention system that identifies, in a task-dependent way, important parts of an image to examine while suppressing examination of other areas. There is massive feedback in the visual stream that is, as yet, little understood. There are widespread associative inputs from muscle control sensors and all of the other senses that allow the brain to draw on cross-associations made from years of living in the world. The feedback loops in the brain go back to all stages of processing including the hardware sensors themselves (the eyes), which mechanically control lighting via the iris and tune the reception on the surface of the retina.

In a machine vision system, however, a computer receives a grid of numbers from the camera or from disk, and, in most cases, that's it. For the most part, there's no built-in pattern recognition, no automatic control of focus and aperture, no cross-associations with years of experience. For the most part, vision systems are still fairly naïve. Figure 1-1 shows a picture of an automobile. In that picture we see a side mirror on the driver's side of the car. What the computer "sees" is just a grid of numbers. Any given number within that grid has a rather large noise component and so by itself gives us little information, but this grid of numbers is all the computer "sees". Our task then becomes to turn this noisy grid of numbers into the perception: "side mirror". Figure 1-2 gives some more insight into why computer vision is so hard.

---

[2] Computer vision is a vast field. This book will give you a basic grounding in the field, but we also recommend texts by Szeliski [Szeliski2011] for a good overview of practical computer vision algorithms, and Hartley [Hartley06] for how 3D vision really works.

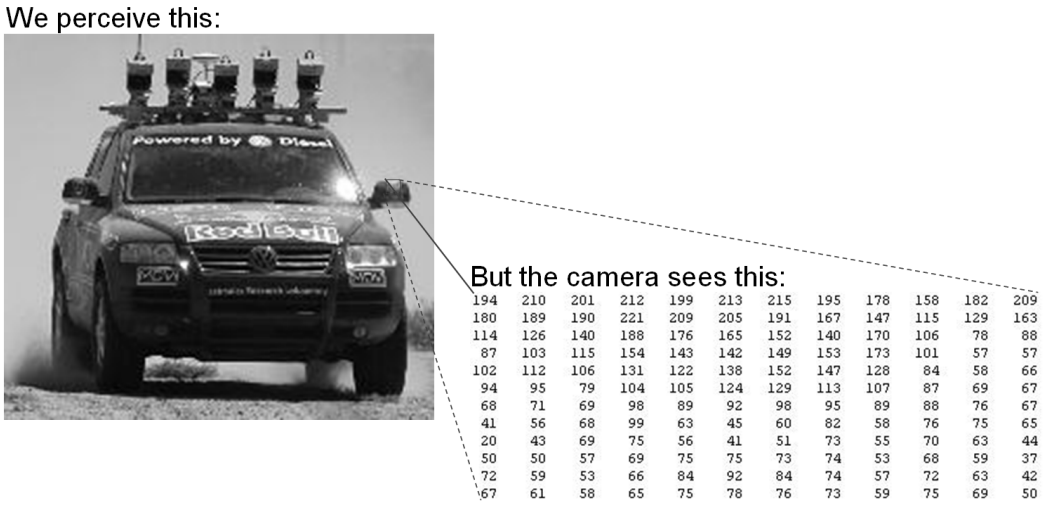


Figure 1-1. To a computer, the car's side mirror is just a grid of numbers

In fact, the problem, as we have posed it thus far, is worse than hard; it is formally impossible to solve. Given a two-dimensional (2D) view of a 3D world, there is no unique way to reconstruct the 3D signal. Formally, such an ill-posed problem has no unique or definitive solution. The same 2D image could represent any of an infinite combination of 3D scenes, even if the data were perfect. However, as already mentioned, the data is corrupted by noise and distortions. Such corruption stems from variations in the world (weather, lighting, reflections, movements), imperfections in the lens and mechanical setup, finite integration time on the sensor (motion blur), electrical noise and compression artifacts after image capture. Given these daunting challenges, how can we make any progress?

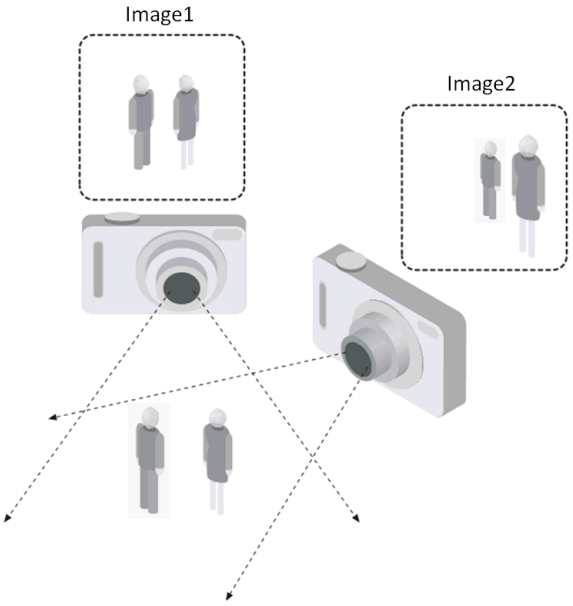


Figure 1-2: The ill-posed nature of vision: the 2D appearance of objects can change radically with viewpoints

In the design of a practical system, additional contextual knowledge can often be used to work around the limitations imposed on us by visual sensors. Consider the example of a mobile robot that must find and pick up staplers in a building. The robot might use the facts that a desk is an object found inside offices and that staplers are mostly found on desks. This gives an implicit size reference; staplers must be able to fit on desks. It also helps to eliminate falsely "recognizing" staplers in impossible places (e.g., on the ceiling or a window). The robot can safely ignore a 200-foot advertising blimp shaped like a stapler because the blimp

lacks the prerequisite wood-grained background of a desk. In contrast, with tasks such as image retrieval, all stapler images in a database may be of real staplers and so large sizes and other unusual configurations may have been implicitly precluded by the assumptions of those who took the photographs. That is, the photographer perhaps took pictures only of real, normal-sized staplers. Also, when taking pictures, people tend to center objects and put them in characteristic orientations. Thus, there is often quite a bit of unintentional implicit information within photos taken by people.

Contextual information can also be modeled explicitly with machine learning techniques. Hidden variables such as size, orientation to gravity, and so on can then be correlated with their values in a labeled training set. Alternatively, one may attempt to measure hidden bias variables by using additional sensors. The use of a laser range finder to measure depth allows us to accurately infer the size of an object.

The next problem facing computer vision is noise. We typically deal with noise by using statistical methods. For example, it may be impossible to detect an edge in an image merely by comparing a point to its immediate neighbors. But if we look at the statistics over a local region, edge detection becomes much easier. A real edge should appear as a string of such immediate neighbor responses over a local region, each of whose orientation is consistent with its neighbors. It is also possible to compensate for noise by taking statistics over time. Still, other techniques account for noise or distortions by building explicit models learned directly from the available data. For example, because lens distortions are well understood, one need only learn the parameters for a simple polynomial model in order to describe—and thus correct almost completely—such distortions.

The actions or decisions that computer vision attempts to make based on camera data are performed in the context of a specific purpose or task. We may want to remove noise or damage from an image so that our security system will issue an alert if someone tries to climb a fence or because we need a monitoring system that counts how many people cross through an area in an amusement park. Vision software for robots that wander through office buildings will employ different strategies than vision software for stationary security cameras because the two systems have significantly different contexts and objectives. As a general rule: the more constrained a computer vision context is, the more we can rely on those constraints to simplify the problem and the more reliable our final solution will be.

OpenCV is aimed at providing the basic tools needed to solve computer vision problems. In some cases, high-level functionalities in the library will be sufficient to solve the more complex problems in computer vision. Even when this is not the case, the basic components in the library are complete enough to enable creation of a complete solution of your own to almost any computer vision problem. In the latter case, there are some tried-and-true methods of using the library; all of them start with solving the problem using as many available library components as possible. Typically, after you've developed this first-draft solution, you can see where the solution has weaknesses and then fix those weaknesses using your own code and cleverness (better known as "solve the problem you actually have, not the one you imagine"). You can then use your draft solution as a benchmark to assess the improvements you have made. From that point, whatever weaknesses remain can be tackled by exploiting the context of the larger system in which your problem solution is embedded, or by setting out to improve some component of the system with your own novel contributions.

The Origin of OpenCV

OpenCV grew out of an Intel Research initiative to advance CPU-intensive applications. Toward this end, Intel launched many projects including real-time ray tracing and 3D display walls. One of the authors (Gary) working for Intel at that time was visiting universities and noticed that some top university groups, such as the MIT Media Lab, had well-developed and internally open computer vision infrastructures—code that was passed from student to student and that gave each new student a valuable head start in developing his or her own vision application. Instead of reinventing the basic functions from scratch, a new student could begin by building on top of what came before.

Thus, OpenCV was conceived as a way to make computer vision infrastructure universally available. With the aid of Intel's Performance Library Team,[3] OpenCV started with a core of implemented code and algorithmic specifications being sent to members of Intel's Russian library team. This is the "where" of OpenCV: it started in Intel's research lab with collaboration from the Software Performance Libraries group together with implementation and optimization expertise in Russia.

Chief among the Russian team members was Vadim Pisarevsky, who managed, coded, and optimized much of OpenCV and who is still at the center of much of the OpenCV effort. Along with him, Victor Eruhimov helped develop the early infrastructure, and Valery Kuriakin managed the Russian lab and greatly supported the effort. There were several goals for OpenCV at the outset:

- Advance vision research by providing not only open but also optimized code for basic vision infrastructure. No more reinventing the wheel.
- Disseminate vision knowledge by providing a common infrastructure that developers could build on, so that code would be more readily readable and transferable.
- Advance vision-based commercial applications by making portable, performance-optimized code available for free—with a license that did not require commercial applications to be open or free themselves.

Those goals constitute the "why" of OpenCV. Enabling computer vision applications would increase the need for fast processors. Driving upgrades to faster processors would generate more income for Intel than selling some extra software. Perhaps that is why this open and free code arose from a hardware vendor rather than a software company. Sometimes, there is more room to be innovative at software within a hardware company.

In any open source effort, it is important to reach a critical mass at which the project becomes self-sustaining. There have now been around seven million downloads of OpenCV, and this number is growing by hundreds of thousands every month[4]. The user group now approaches 50,000 members. OpenCV receives many user contributions, and central development has long since moved outside of Intel.[5] OpenCV's past timeline is shown in Figure 1-3. Along the way, OpenCV was affected by the dot-com boom and bust and also by numerous changes of management and direction. During these fluctuations, there were times when OpenCV had no one at Intel working on it at all. However, with the advent of multicore processors and the many new applications of computer vision, OpenCV's value began to rise. Similarly, rapid growth in the field of robotics has driven much use and development of the library. After becoming an open source library, OpenCV spent several years under active development at Willow Garage and Itseez, and now is supported by the OpenCV foundation at http//opencv.org. Today, OpenCV is actively being developed by the OpenCV.org foundation, Google supports on order of 15 interns a year in the Google Summer of Code program[6], and Intel is back actively supporting development. For more information on the future of OpenCV, see Chapter 14.

---

[3] Shinn Lee was of key help as was Stewart Taylor.

[4] It is noteworthy, that at the time of the publication of "Learning OpenCV" in 2006, this rate was 26,000 per month. Seven years later, the download rate has grown to over 160,000 downloads per month.

[5] As of this writing, Itseez (http://itseez.com/) is the primary maintainer of OpenCV

[6] Google Summer of Code https://developers.google.com/open-source/soc/

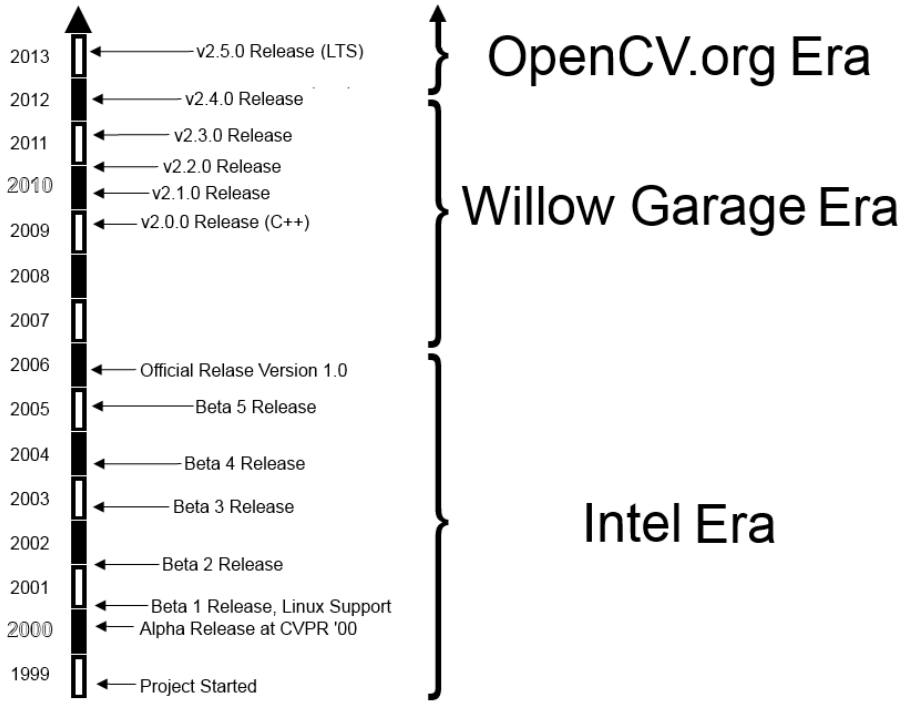


Figure 1-3: OpenCV timeline

Who Owns OpenCV?

Although Intel started OpenCV, the library is and always was intended to promote commercial and research use. It is therefore open and free, and the code itself may be used or embedded (in whole or in part) in other applications, whether commercial or research. It does not force your application code to be open or free. It does not require that you return improvements back to the library—but we hope that you will.

Downloading and Installing OpenCV

The main OpenCV site is at http://opencv.org, from which you can download the complete source code for the latest release, as well as many recent releases. The downloads themselves are found at the downloads page: http://opencv.org/downloads.html. However, if you want the very most up-to-date version it is always found on GitHub at https://github.com/Itseez/opencv, where the active development branch is stored. The computer vision developer's site (with links to the above) is at http://code.opencv.org/.

Installation

In modern times, OpenCV uses Git as its development version control system, and CMake to build[7]. In many cases, you will not need to worry about building, as compiled libraries exist for supported environments. However, as you become a more advanced user, you will inevitably want to be able to recompile the libraries with specific options tailored to your application and environment. On the tutorial pages at http://docs.opencv.org/doc/tutorials/tutorials.html under "introduction to OpenCV", there are descriptions of how to set up OpenCV to work with a number of combinations of operating systems and development tools.

[7] In olden times, OpenCV developers used Subversion for version control and automake to build. Those days, however, are long gone.

Windows

At the page: http://opencv.org/downloads.html, you will see a link to download the latest version of OpenCV for Windows. This link will download an executable file which you can run, and which will install OpenCV, register DirectShow filters, and perform various post-installation procedures. You are now almost ready to start using OpenCV.[8]

The one additional detail is that you will want to add is an OPENCV_DIR environment variable to make it easier to tell your compiler where to find the OpenCV binaries. You can set this by going to a command prompt and typing[9]:

```
setx -m OPENCV_DIR D:\OpenCV\Build\x86\vc10
```

If you built the library to link statically, this is all you will need. If you built the library to link dynamically, then you will also need to tell your system where to find the library binary. To do this, simply add `%OPENCV_DIR%\bin` to your library path. (For example, in Windows 7, right-click on your Computer icon, select Properties, and then click on Advanced System Settings. Finally select Environment Variables and add the OpenCV binary path to the Path variable.)

To add the commercial IPP performance optimizations to Windows, obtain and install IPP from the Intel site (http://www.intel.com/software/products/ipp/index.htm); use version 5.1 or later. Make sure the appropriate binary folder (e.g., *c:/program files/intel/ipp/5.1/ia64/bin*) is in the system path. IPP should now be automatically detected by OpenCV and loaded at runtime (more on this in Chapter 3).

Linux

Prebuilt binaries for Linux are not included with the Linux version of OpenCV owing to the large variety of versions of GCC and GLIBC in different distributions (SuSE, Debian, Ubuntu, etc.). In many cases however, your distribution will include OpenCV. If your distribution doesn't offer OpenCV, you will have to build it from sources. As with the Windows installation, you can start at the http://opencv.org/downloads.html page, but in this case the link will send you to Sourceforge[10], where you can select the tarball for the current OpenCV source code bundle.

To build the libraries and demos, you'll need GTK+ 2.x or higher, including headers. You'll also need *pkgconfig*, *libpng*, *libjpeg*, *libtiff*, and *libjasper* with development files (i.e., the versions with *-dev* at the end of their package names). You'll need Python 2.6 or later with headers installed (developer package). You will also need *libavcodec* and the other *libav** libraries (including headers) from *ffmpeg* `1.0` or later .

Download ffmpeg from http://ffmpeg.mplayerhq.hu/download.html.[11] The ffmpeg program has a lesser general public license (LGPL). To use it with non-GPL software (such as OpenCV), build and use a shared ffmpg library:

```
$> ./configure --enable-shared
$> make
$> sudo make install
```

You will end up with: */usr/local/lib/libavcodec.so.**, */usr/local/lib/libavformat.so.**, */usr/local/lib/libavutil.so.**, and include files under various */usr/local/include/libav**.

To build OpenCV once it is downloaded:[12]

---

[8] It is important to know that, although the Windows distribution contains binary libraries for release builds, it does not contain the debug builds of these libraries. It is therefore likely that, before developing with OpenCV, you will want to open the solution file and build these libraries for yourself.

[9] Of course, the exact path will vary depending on your installation, for example if you are installing on an ia64 machine, then the path will not include "x86", but rather "ia64".

[10] OpenCV has all of its many builds and versions available from Sourceforge. See links at http://opencv.org/downloads.html

[11] You can check out ffmpeg by: svn checkout svn://svn.mplayerhq.hu/ffmpeg/trunk ffmpeg.

```
$> mkdir build && cd build
$> cmake .. && make
$> sudo make install
$> sudo ldconfig
```

After installation is complete, the default installation path is */usr/local/lib/* and */usr/local/include/opencv2/*. Hence you need to add */usr/local/lib/* to */etc/ld.so.conf* (and run `ldconfig` afterwards) or add it to the `LD_LIBRARY_PATH` environment variable; then you are done.

To actually build the library, you will need to go unpack the *.tgz* file and go into the created source directory, and do the following:

```
mkdir release
cd release
cmake -D CMAKE_BUILD_TYPE=RELEASE -D CMAKE_INSTALL_PREFIX=/usr/local ..
make
sudo make install
```

The first and second commands create a new subdirectory and move you into it. The third command tells CMake how to configure your build. The example options we give are probably the right ones to get you started, but other options allow you to enable various options, determine what examples are built, python support, CUDA GPU support, etc. The last two commands actually build the library and install the results into the proper places.

To add the commercial IPP performance optimizations to Linux, install IPP as described previously. Let's assume it was installed in */opt/intel/ipp/5.1/ia32/*. Add *<your install_path>/bin/* and *<your install_path>/bin/linux32* `LD_LIBRARY_PATH` in your initialization script (*.bashrc* or similar):

```
LD_LIBRARY_PATH=/opt/intel/ipp/5.1/ia32/bin:/opt/intel/ipp/5.1
/ia32/bin/linux32:$LD_LIBRARY_PATH
export LD_LIBRARY_PATH
```

Alternatively, you can add *<your install_path>/bin* and *<your install_path>/bin/linux32*, one per line, to */etc/ld.so.conf* and then run *ldconfig* as root (or use sudo).

That's it. Now OpenCV should be able to locate IPP shared libraries and make use of them on Linux. See *../opencv/INSTALL* for more details.

MacOS X

As of this writing, full functionality on MacOS X is a priority but there are still some limitations (e.g., writing AVIs); these limitations are described in *../opencv/INSTALL*.

The requirements and building instructions are similar to the Linux case, with the following exceptions:

- By default, Carbon is used instead of GTK+.
- By default, QuickTime is used instead of ffmpeg.
- pkg-config is optional (it is used explicitly only in the *samples/c/build_all.sh* script).
- RPM and ldconfig are not supported by default. Use `configure+make+sudo make install` to build and install OpenCV, update `DYLD_LIBRARY_PATH` (unless `./configure --prefix=/usr` is used).

For full functionality, you should install *libpng*, *libtiff*, *libjpeg* and *libjasper* from *darwinports* and/or *fink* and make them available to *./configure* (see `./configure --help`). Then:

```
sudo port selfupdate
sudo port install opencv
```

---

[12] To build OpenCV using Red Hat Package Managers (RPMs), use **`rpmbuild -ta OpenCV-x.y.z.tar.gz`** (for RPM 4.x or later), or **`rpm -ta OpenCV-x.y.z.tar.gz`** (for earlier versions of RPM), where *OpenCV-x.y.z.tar.gz* should be put in */usr/src/redhat/SOURCES/* or a similar directory. Then install OpenCV using **`rpm -i OpenCV-x.y.z.*.rpm`**.

Notes on Building with CMake

The modern OpenCV library relies on CMake in its build system. This has the advantage of making the platform much easier to work with in a cross-platform environment. Whether you are using the command line version of CMake on Linux (or on Windows using a command line environment such as Cygwin), or using a visual interface to CMake such as *cmake-gui*, you can build OpenCV in just about any environment in the same way.

If you are not already familiar with CMake, the essential concept behind CMake is to allow the developers to specify all of the information needed for compilation in a platform independent way (files called *CMakeLists.txt* files), which CMake then converts to the platform dependent files used in your environment (e.g., makefiles on Unix and projects or workspaces in the Windows visual environment).

When you use CMake, you can supply additional options which tell CMake how to proceed in the generation of the build files. For example, it is CMake which you tell if you want a debug or release library or if you do or do not want to include a feature like the Intel Performance Primitives (IPP). An example CMake command line invocation, which we encountered earlier, might be:

```
cmake -D CMAKE_BUILD_TYPE=RELEASE -D CMAKE_INSTALL_PREFIX=/usr/local ..
```

The last argument '..' is actually a directory specification, which tells CMake where the root of the source directory to build is. In this case it is set to '..' (the directory above) because it is conventional in OpenCV to actually do the build in a subdirectory of the OpenCV root.

In this case, the `-D` option is used to set environment variables which CMake will use to determine what to put in your build files. These environment variables might be essentially enumerated values (such as `RELEASE` or `DEBUG` for `CMAKE_BUILD_TYPE`), or they might be strings (such as '*/usr/local*' for `CMAKE_INSTALL_PREFIX`). Below is a partial list[13] containing many of the common options that you are likely to want.

Table 1-1: Basic CMake options you will probably need.

Option	**Definition**	**Accepted Values**
`CMAKE_BUILD_TYPE`	Controls release vs. debug build	`RELEASE` or `DEBUG`
`CMAKE_INSTALL_PREFIX`	Where to put installed library	Path, e.g. */usr/local*
`CMAKE_VERBOSE`	Lots of extra information from CMake	`ON` or `OFF`

Table 1-2: Options which introduce features into the library. All of these can be either `ON` or `OFF`.

Option	**Definition**	**Default**
`WITH_1394`	Use libdc1394	`ON`
`WITH_CUDA`	CUDA support, requires toolkit	`OFF`
`WITH_EIGEN2`	Use Eigen library for linear algebra	`ON`
`WITH_FFMPEG`	Use ffmpeg for video I/O	`ON`

[13] You are probably wondering: "why not a complete list?" The reason is simply that the available options fluctuate with the passage of time. The ones we list here are some of the more important ones however which are likely to stay intact into the foreseeable future.

WITH_OPENGL	enable wrappers for OpenGL objects in the core module and the function: `cv::setOpenGLDrawCallback()`	OFF
WITH_GSTREAMER	Include Gstreamer support	ON
WITH_GTK	Include GTK support	ON
WITH_IPP	Intel Performance Primitives[14]	OFF
WITH_JASPER	JPEG 2000 support for imread()	ON
WITH_JPEG	JPEG support for imread()	ON
WITH_MIKTEX	PDF documentation on Windows	OFF
WITH_OPEN_EXR	EXR support for imread()	ON
WITH_OPENCL	OpenCL support (similar to CUDA)	OFF
WITH_OPENNI	Support for Kinect cameras	OFF
WITH_PNG	PNG support for loading	ON
WITH_QT	Qt based Highgui functions	OFF
WITH_QUICKTIME	Use Quicktime for video I/O (instead of QTKit – Mac only)	OFF
WITH_TBB	Intel Thread Building Blocks	OFF
WITH_TIFF	TIFF support for imread()	ON
WITH_UNICAP	Use Unicap library, provides I/O support for cameras using this standard.	OFF
WITH_V4L	Video for Linux support	ON
WITH_VIDEOINPUT	Use alternate library for video I/O (Windows only)	OFF
WITH_XIMEA	Support for Ximea cameras	OFF
WITH_XINE	Enable Xine multimedia library for video	OFF
ENABLE_SOLUTION_FOLDERS	Categorize binaries inside directories in Solution Explorer	OFF

[14] Associated with the WITH_IPP option, there is also the IPP_PATH option. The IPP_PATH can be set to any normal path name and indicates where the IPP libraries should be found. However, it should not be necessary if you have these libraries in the "usual" place for your platform.

Table 1.3: Options passed by CMake to the compiler.

Option	**Definition**	**Default**
ENABLE_SSE	Enable Streaming SIMD (SSE) instructions	ON
ENABLE_SSE2	Enable Intel SSE 2 instructions	ON
ENABLE_SSE3	Enable Intel SSE 3 instructions	OFF
ENABLE_SSSE3	Enable Supplemental SSE 3 instructions	OFF
ENABLE_SSE41	Enable Intel SSE 4.1 instructions	OFF
ENABLE_SSE42	Enable Intel SSE 4.1 instructions	OFF
USE_O3	Enable high optimization (i.e. -o3)	ON
USE_OMIT_FRAME_POINTER	Enable omit frame pointer (i.e. -fomit-frame-pointer)	ON

Table 1-4: 'Build' options control exactly what gets created at compile time. All of these can be either ON or OFF. In most all cases the default is OFF.

Option	**Definition**	**Default**
BUILD_DOCS	Generate OpenCV documentation	OFF
BUILD_EXAMPLES	Generate example code	OFF
BUILD_JAVA_SUPPORT	Create Java OpenCV libraries	OFF
BUILD_PYTHON_SUPPORT	Deprecated (use BUILD_NEW_PYTHON_SUPPORT)	OFF
BUILD_NEW_PYTHON_SUPPORT	Create Python OpenCV libraries	ON
BUILD_PACKAGE	Create zip archive with OpenCV sources	OFF
BUILD_SHARED_LIBS	Build OpenCV as dynamic library (default is static)	ON
BUILD_TESTS	Build test programs for each OpenCV module	ON
BUILD_PERF_TESTS	Build available function performance tests	OFF

Table 1-5: 'Install' options determine what compiled executables get placed in your binaries area.

INSTALL_C_EXAMPLES	Install the C and C++ examples in the "usual" place for binaries	OFF
INSTALL_PYTHON_EXAMPLES	Install the Python examples in the "usual" place for binaries	OFF

Not listed in these tables are additional variables which can be set to indicate the locations of various libraries (e.g., libjasper, etc.) in such case as they are not in their default locations. For more information on these more obscure options, a visit to the online documentation at *http://opencv.org* is recommended.

Getting the Latest OpenCV via Git

OpenCV is under active development, and bugs are often fixed rapidly when reports contain accurate descriptions and code that demonstrates the bug. However, official major OpenCV releases only occur two to four times a year. If you are seriously developing a project or product, you will probably want code fixes and updates as soon as they become available. To do this, you will need to access OpenCV's Git repository on Github.

This isn't the place for a tutorial in Git usage. If you've worked with other open source projects then you're probably familiar with it already. If you haven't, check out Version Control with Git by Jon Loeliger (O'Reilly). A command-line Git client is available for Linux, OS X, and most UNIX-like systems. For Windows users, we recommend TortoiseGit (http://code.google.com/p/tortoisegit/).

On Windows, if you want the latest OpenCV from the Git repository then you'll need to clone the OpenCV repository on https://github.com/Itseez/opencv.git.

On Linux and Mac, you can just use the following command:

```
git clone https://github.com/Itseez/opencv.git
```

More OpenCV Documentation

The primary documentation for OpenCV is the HTML documentation available at: http//opencv.org. In addition to this, there are extensive tutorials on many subjects at http://docs.opencv.org/doc/tutorials/tutorials.html, and an OpenCV Wiki (currently located at http://code.opencv.org/projects/opencv/wiki).

Online Documentation and the Wiki

As briefly mentioned earlier, there is extensive documentation as well as a wiki available at http://opencv.org. The documentation there is divided into several major components:

- Reference (http://docs.opencv.org/): This section contains the functions, their arguments, and some information on how to use them.
- Tutorials (http://docs.opencv.org/doc/tutorials/tutorials.html): There is a large collection of tutorials, these tell you how to accomplish various things. There are tutorials for basic subjects, like how to install OpenCV or create OpenCV projects on various platforms, and more advanced topics like background subtraction of object detection.
- Quick Start (http://opencv.org/quickstart.html): This is really a tightly curated subset of the tutorials, containing just ones that help you get up and running on specific platforms.

- Cheat Sheet (http://docs.opencv.org/trunk/opencv_cheatsheet.pdf): This is actually a single .pdf file which contains a truly excellent compressed reference to almost the entire library. Thank Vadim Pisarevsky for this excellent reference as you pin these two beautiful pages to your cubicle wall.
- Wiki (http://code.opencv.org/projects/opencv/wiki): The wiki contains everything you could possible want and more. This is where the roadmap can be found, as well as news, open issues, bugs tracking, and countless deeper topics like how to become a contributor to OpenCV.
- Q&A (http://answers.opencv.org/questions): This is a vast archive of literally thousands of questions people have asked, and answered. You can go there to ask questions of the OpenCV community, or to help others by answering their questions.

All of these are accessible under the "Documentation" button on the OpenCV.org homepage. Of all of those great resources, one warrants a little more discussion here, which is the Reference. The reference is divided into several sections, each of which pertains to what is called a *module* in the library. The exact module list has evolved over time, but the modules are the primary organizational structure in the library. Every function in the library is part of one module. Here are the current modules:

- core: The "core" is the section of the library which contains all of the basic object types and their basic operations.
- imgproc: The image processing module contains basic transformations on images, including filters and similar convolutional operators.
- highgui: This HighGUI module contains user interface functions which can be used to display images or take simple user input. It can be thought of as a very light weight window UI toolkit.
- video: The video library contains the functions you need to read and write video streams.
- calib3d: This module contains implementations of algorithms you will need to calibrate single cameras as well as stereo or multi-camera arrays.
- features2d: The features2d module contains algorithms for detecting, describing, and matching keypoint features.
- objdetect: This module contains algorithms for detecting specific objects, such as faces or pedestrians. You can train the detectors to detect other objects as well.
- ml: The Machine Learning Library is actually an entire library in itself, and contains a wide array of machine learning algorithms implemented in such as way as to work with the natural data structures of OpenCV.
- flann: FLANN stands for "Fast Library for Approximate Nearest Neighbors". This library contains methods you will not likely use directly, but which are used by other functions in other modules for doing nearest neighbor searches in large data sets.
- gpu: The GPU library contains implementations of most of the rest of the library functions optimized for operation on CUDA GPUs. There are also some functions which are only implemented for GPU operation. Some of these provide excellent results but require computational resources sufficiently high that implementation on non-GPU hardware would provide little utility.
- photo: This is a relatively new module which contains tools useful for computational photography.
- stitching: This entire module implements a sophisticated image stitching pipeline. This is new functionality in the library but, like the 'photo' module is a place where future growth is expected.
- nonfree: OpenCV contains some implementations of algorithms which are patented or are otherwise burdened by some usage restrictions (e.g., the SIFT algorithm). Those algorithms are segregated off to their own module, so that you will know that you will need to do some kind of special work in order to use them in a commercial product.

- contrib: This module contains new things that have yet to be blessed into the whole of the library.
- legacy: This module contains old things that have yet to be banished from the library altogether.
- ocl: The OCL module is a newer module, which could be considered analogous to the GPU module, except that it relies on OpenCL, a Khronos standard for open parallel computing. Though less featured than the GPU module at this time, the OCL module aims to provide implementations which can run on any GPU or other device supporting OpenCL. (This is in contrast to the GPU module which explicitly makes use of the Nvidia CUDA toolkit and so will only work on Nvidia GPU devices.)

Despite the ever-increasing quality of this online documentation, one task which is not within their scope is to provide a proper understanding of the algorithms implemented or of the exact meaning of the parameters these algorithms require. This book aims to provide this information, as well as a more in depth understanding of all of the basic building blocks of the library.

Exercises

1. Download and install the latest release of OpenCV. Compile it in debug and release mode.
2. Download and build the latest trunk version of OpenCV using Git.
3. Describe at least three ambiguous aspects of converting 3D inputs into a 2D representation. How would you overcome these ambiguities?
4. What shapes can a rectangle take when you look at in with perspective distortion (that is, when you look at it in the real world)?
5. Describe how you might start processing the image in Figure 1-1 to identify the mirror on the car?
6. How might you tell the difference between a edges in an image created by:
   a) A shadow?
   b) Paint on a surface?
   c) Two sides of a brick?
   d) The side of an object and the background?

2

Introduction to OpenCV 2.x

Include files

After installing the OpenCV library and setting up our programming environment, our next task is to make something interesting happen with code. In order to do this, we'll have to discuss header files. Fortunately, the headers reflect the new, modular structure of OpenCV introduced in Chapter 1. The main header file of interest is *../include/opencv2/opencv.hpp*. This header file just calls the header files for each OpenCV module:

#include "opencv2/core/core_c.h"

Old C data structures and arithmetic routines.

#include "opencv2/core/core.hpp"

New C++ data structures and arithmetic routines.

#include "opencv2/flann/miniflann.hpp"

Approximate nearest neighbor matching functions. (Mostly for internal use)

#include "opencv2/imgproc/imgproc_c.h"

Old C image processing functions.

#include "opencv2/imgproc/imgproc.hpp"

New C++ image processing functions.

#include "opencv2/video/photo.hpp"

Algorithms specific to handling and restoring photographs.

#include "opencv2/video/video.hpp"

Video tracking and background segmentation routines.

#include "opencv2/features2d/features2d.hpp"

Two-dimensional feature tracking support.

#include "opencv2/objdetect/objdetect.hpp"

Cascade face detector; latent SVM; HoG; planar patch detector.

#include "opencv2/calib3d/calib3d.hpp"

Calibration and stereo.

#include "opencv2/ml/ml.hpp"

Machine learning: clustering, pattern recognition.

#include "opencv2/highgui/highgui_c.h"

Old C image display, sliders, mouse interaction, I/O.

#include "opencv2/highgui/highgui.hpp"

New C++ image display, sliders, buttons, mouse, I/O.

#include "opencv2/contrib/contrib.hpp"

User-contributed code: flesh detection, fuzzy mean-shift tracking, spin images, self-similar features.

You may use the include file *opencv.hpp* to include any and every possible OpenCV function but, since it includes everything, it will cause compile time to be slower. If you are only using, say, image processing functions, compile time will be faster if you only include *opencv2/imgproc/imgproc.hpp*. These include files are located on disk under the *../modules* directory. For example, *imgproc.hpp* is located at *../modules/imgproc/include/opencv2/imgproc/imgproc.hpp*. Similarly, the sources for the functions themselves are located under their corresponding *src* directory. For example, `cv::Canny()` in the *imgproc* module is located in *../modules/improc/src/canny.cpp*.

With the above include files, we can start our first C++ OpenCV program.

> Legacy code such as the older blob tracking, hmm face detection, condensation tracker, and eigen objects can be included using *opencv2/legacy/legacy.hpp*, which is located in *../modules/legacy/include/opencv2/legacy/legacy.hpp*.

First Program—Display a Picture

OpenCV provides utilities for reading from a wide array of image file types, as well as from video and cameras. These utilities are part of a toolkit called HighGUI, which is included in the OpenCV package. On the http://opencv.org site, you can go to the tutorial pages off of the documentation links at http://docs.opencv.org/doc/tutorials/tutorials.html to see tutorials on various aspects of using OpenCV.

> In the tutorial section, the "introduction to OpenCV" tutorial explains how to set up OpenCV for various combinations of operating systems and development tools.

We will use an example from the "highgui module" to create a simple program that opens an image and displays it on the screen (Example 2-1).

Example 2-1: A simple OpenCV program that loads an image from disk and displays it on the screen

```
#include <opencv2/opencv.hpp> //Include file for every supported OpenCV function

int main( int argc, char** argv ) {
  cv::Mat img = cv::imread(argv[1],-1);
  if( img.empty() ) return -1;
  cv::namedWindow( "Example1", cv::WINDOW_AUTOSIZE );
  cv::imshow( "Example1", img );
  cv::waitKey( 0 );
  cv::destroyWindow( "Example1" );
}
```

Note that OpenCV functions live within a namespace called *cv*. To call OpenCV functions, you must explicitly tell the compiler that you are talking about the *cv* namespace by prepending *cv::* to each function

call. To get out of this bookkeeping chore, we can employ the `using namespace cv;` directive as shown in Example 2-2[1]. This tells the compiler to assume that functions might belong to that namespace. Note also the difference in include files between Example 2-1 and Example 2-2; in the former, we used the general include *opencv.hpp*, whereas in the latter, we used only the necessary include file to improve compile time.

Example 2-2: Same as Example 2-1 but employing the "using namespace" directive. For faster compile, we use only the needed header file, not the generic opencv.hpp.

```
#include "opencv2/highgui/highgui.hpp"

using namespace cv;

int main( int argc, char** argv ) {
  Mat img = imread( argv[1], -1 );
  if( img.empty() ) return -1;
  namedWindow( "Example2", WINDOW_AUTOSIZE );
  imshow( "Example2", img );
  waitKey( 0 );
  destroyWindow( "Example2" );
}
```

When compiled and run from the command line with a single argument, Example 2-1 loads an image into memory and displays it on the screen. It then waits until the user presses a key, at which time it closes the window and exits. Let's go through the program line by line and take a moment to understand what each command is doing.

```
cv::Mat img = cv::imread( argv[1], -1 );
```

This line loads the image.[2] The function `cv::imread()` is a high-level routine that determines the file format to be loaded based on the file name; it also automatically allocates the memory needed for the image data structure. Note that `cv::imread()` can read a wide variety of image formats, including BMP, DIB, JPEG, JPE, PNG, PBM, PGM, PPM, SR, RAS, and TIFF. A `cv::Mat` structure is returned. This structure is the OpenCV construct with which you will deal the most. OpenCV uses this structure to handle all kinds of images: single-channel, multichannel, integer-valued, floating-point-valued, and so on. The line immediately following

```
if( img.empty() ) return -1;
```

checks to see if an image was in fact read. Another high-level function, `cv::namedWindow()`, opens a window on the screen that can contain and display an image.

```
cv::namedWindow( "Example2", cv::WINDOW_AUTOSIZE );
```

This function, provided by the HighGUI library, also assigns a name to the window (in this case, `"Example2"`). Future HighGUI calls that interact with this window will refer to it by this name.

The second argument to `cv::namedWindow()` defines window properties. It may be set either to `0` (the default value) or to `cv::WINDOW_AUTOSIZE`. In the former case, the size of the window will be the same regardless of the image size, and the image will be scaled to fit within the window. In the latter case, the window will expand or contract automatically when an image is loaded so as to accommodate the image's true size but may be resized by the user.

---

[1] Of course, once you do this, you risk conflicting names with other potential namespaces. If the function `foo()` exists, say, in the *cv* and *std* namespaces, you must specify which function you are talking about using either `cv::foo()` or `std::foo()` as you intend. In this book, other than in our specific example of Example 2-2, we will use the explicit form *cv::* for objects in the OpenCV namespace, as this is generally considered to be better programming style.

[2] A proper program would check for the existence of `argv[1]` and, in its absence, deliver an instructional error message for the user. We will abbreviate such necessities in this book and assume that the reader is cultured enough to understand the importance of error-handling code.

```
cv::imshow( "Example2", img );
```

Whenever we have an image in a `cv::Mat` structure, we can display it in an existing window with `cv::imshow()`.[3] On the call to `cv::imshow()`, the window will be redrawn with the appropriate image in it, and the window will resize itself as appropriate if it was created using the `cv::WINDOW_AUTOSIZE` flag.

```
cv::waitKey( 0 );
```

The `cv::waitKey()` function asks the program to stop and wait for a keystroke. If a positive argument is given, the program will wait for that number of milliseconds and then continue even if nothing is pressed. If the argument is set to `0` or to a negative number, the program will wait indefinitely for a key-press. This function has another very important role: It handles any windowing events, such as creating windows and drawing their content. So it must be used after `cv::imshow()` in order to display that image.

With `cv::Mat`, images are automatically deallocated when they go out of scope, similar to the STL-style container classes. This automatic deallocation is controlled by an internal reference counter. For the most part, this means we no longer need to worry about the allocation and deallocation of images, which relieves the programmer from much of the tedious bookkeeping that the OpenCV 1.0 `IplImage` imposed.

```
cv::destroyWindow( "Example2" );
```

Finally, we can destroy the window itself. The function `cv::destroyWindow()` will close the window and deallocate any associated memory usage. For short programs, we will skip this step. For longer, complex programs, the programmer should make sure to tidy up the windows before they go out of scope to avoid memory leaks.

Our next task will be to construct a very simple—almost as simple as this one—program to read in and display a video file. After that, we will start to tinker a little more with the actual images.

Second Program—Video

Playing a video with OpenCV is almost as easy as displaying a single picture. The only new issue we face is that we need some kind of loop to read each frame in sequence; we may also need some way to get out of that loop if the movie is too boring. See Example 2-3.

Example 2-3: A simple OpenCV program for playing a video file from disk. In this example we only use specific module headers, rather than just opencv.hpp. This speeds up compilation, and so is sometimes preferable.

```
#include "opencv2/highgui/highgui.hpp"
#include "opencv2/imgproc/imgproc.hpp"

int main( int argc, char** argv ) {
  cv::namedWindow( "Example3", cv::WINDOW_AUTOSIZE );
  cv::VideoCapture cap;
  cap.open( string(argv[1]) );
  cv::Mat frame;
  while( 1 ) {
    cap >> frame;
    if( !frame.data ) break;             // Ran out of film
    cv::imshow( "Example3", frame );
    if( cv::waitKey(33) >= 0 ) break;
  }
  return 0;
}
```

[3] In the case where there is no window in existence at the time you call `imshow()`, one will be created for you with the name you specified in the `imshow()` call. This window can still be destroyed as usual with `destroyWindow()`.

Here we begin the function `main()` with the usual creation of a named window (in this case, named "`Example3`"). The video capture object `cv::VideoCapture` is then instantiated. This object can open and close video files of as many types as *ffmpeg* supports.

```
cap.open(string(argv[1]));
cv::Mat frame;
```

The capture object is given a string containing the path and filename of the video to be opened. Once opened, the capture object will contain all of the information about the video file being read, including state information. When created in this way, the `cv::VideoCapture` object is initialized to the beginning of the video. In the program, `cv::Mat frame` instantiates a data object to hold video frames.

```
cap >> frame;
if( !frame.data ) break;
cv::imshow( "Example3", frame );
```

Once inside of the `while()` loop, the video file is read frame by frame from the capture object stream. The program checks to see if data was actually read from the video file (`if(!frame.data)`) and quits if not. If a video frame is successfully read in, it is displayed using `cv::imshow()`.

```
if( cv::waitKey(33) >= 0 ) break;
```

Once we have displayed the frame, we then wait for 33 ms[4]. If the user hits a key during that time then we will exit the read loop. Otherwise, 33 ms will pass and we will just execute the loop again. On exit, all the allocated data is automatically released when they go out of scope.

Moving Around

Now it's time to tinker around, enhance our toy programs, and explore a little more of the available functionality. The first thing we might notice about the video player of Example 2-3 is that it has no way to move around quickly within the video. Our next task will be to add a slider trackbar, which will give us this ability. For more control, we will also allow the user to single step the video by pressing the 's' key, to go into run mode by pressing the 'r' key, and whenever the user jumps to a new location in the video with the slider bar, we pause there in single step mode.

The HighGUI toolkit provides a number of simple instruments for working with images and video beyond the simple display functions we have just demonstrated. One especially useful mechanism is the trackbar, which enables us to jump easily from one part of a video to another. To create a trackbar, we call `cv::createTrackbar()` and indicate which window we would like the trackbar to appear in. In order to obtain the desired functionality, we need a callback that will perform the relocation. Example 2-4 gives the details.

Example 2-4: Program to add a trackbar slider to the basic viewer window for moving around within the video file

```
#include "opencv2/highgui/highgui.hpp"
#include "opencv2/imgproc/imgproc.hpp"
#include <iostream>
#include <fstream>

using namespace std;

int g_slider_position = 0;
int g_run = 1, g_dontset = 0; //start out in single step mode
cv::VideoCapture g_cap;
```

[4] You can wait any amount of time you like. In this case, we are simply assuming that it is correct to play the video at 30 frames per second and allow user input to interrupt between each frame (thus we pause for input 33 ms between each frame). In practice, it is better to check the `cv::VideoCapture` structure in order to determine the actual frame rate of the video (more on this in Chapter 4).

```
void onTrackbarSlide( int pos, void *) {
  g_cap.set( cv::CAP_PROP_POS_FRAMES, pos );
  if( !g_dontset )
    g_run = 1;
  g_dontset = 0;
}

int main( int argc, char** argv ) {
  cv::namedWindow( "Example2_4", cv::WINDOW_AUTOSIZE );
  g_cap.open( string(argv[1]) );
  int frames = (int) g_cap.get(cv::CAP_PROP_FRAME_COUNT);
  int tmpw   = (int) g_cap.get(cv::CAP_PROP_FRAME_WIDTH);
  int tmph   = (int) g_cap.get(cv::CAP_PROP_FRAME_HEIGHT);
  cout << "Video has " << frames << " frames of dimensions("
       << tmpw << ", " << tmph << ")." << endl;
  cv::createTrackbar("Position", "Example2_4", &g_slider_position, frames,
                 onTrackbarSlide);
  cv::Mat frame;
  while(1) {
    if( g_run != 0 ) {
      g_cap >> frame; if(!frame.data) break;
      int current_pos = (int)g_cap.get(cv::CAP_PROP_POS_FRAMES);
      g_dontset = 1;
      cv::setTrackbarPos("Position", "Example2_4", current_pos);
      cv::imshow( "Example2_4", frame );
      g_run-=1;
    }
    char c = (char) cv::waitKey(10);
    if(c == 's') // single step
      {g_run = 1; cout << "Single step, run = " << g_run << endl;}
    if(c == 'r') // run mode
      {g_run = -1; cout << "Run mode, run = " << g_run <<endl;}
    if( c == 27 )
      break;
  }
  return(0);
}
```

In essence, the strategy is to add a global variable to represent the trackbar position and then add a callback that updates this variable and relocates the read position in the video. One call creates the trackbar and attaches the callback, and we are off and running.[5] Let's look at the details starting with the global variables.

```
int g_slider_position = 0;
int g_run = 1, g_dontset = 0;      // start out in single step mode
VideoCapture g_cap;
```

First we define a global variable, `g_slider_position`, to keep the trackbar slider position state. The callback will need access to the capture object `g_cap`, so we promote that to a global variable as well. Because we are considerate developers and like our code to be readable and easy to understand, we adopt the convention of adding a leading `g_` to any global variable. We also instantiate another global variable, `g_run`, which displays new frames as long it is different from zero. A positive number tells how many frames are displayed before stopping; a negative number means the system runs in continuous video mode.

To avoid confusion, when the user clicks on the trackbar to jump to a new location in the video, we'll leave the video paused there in the single step state by setting `g_run = 1`. This, however, brings up a subtle problem: as the video advances, we'd like the slider trackbar position in the display window to advance according to our location in the video. We do this by having the main program call the trackbar callback

[5] Note that some AVI and mpeg encodings do not allow you to move backward in the video.

function to update the slider position each time we get a new video frame. However, we don't want these programmatic calls to the trackbar callback to put us into single step mode. To avoid this, we introduce a final global variable, `g_dontset` to allow us to update trackbar position without triggering single state mode.

```
void onTrackbarSlide(int pos, void *) {
  g_cap.set(cv::CAP_PROP_POS_FRAMES, pos);
  if(!g_dontset)
    g_run = 1;
  g_dontset = 0;
}
```

Now we define a callback routine to be used when the user pokes the trackbar. This routine will be passed a 32-bit integer, `pos`, which will be the new trackbar position. Inside this callback, we use the new requested position in `cv::g_cap.set()` to actually advance the video playback to the new position. The `if` statement just sets the program to go into single step mode after the next new frame comes in, but only if the callback was triggered by a user click, not if the callback was called from the main function (which sets `g_dontset`).

The call to `cv::g_cap.set()` is one we will see often in the future, along with its counterpart `cv::g_cap.get()`. These routines allow us to configure (or query in the latter case) various properties of the `cv::VideoCapture` object. In this case, we pass the argument `cv::CAP_PROP_POS_FRAMES`, which indicates that we would like to set the read position in units of frames.[6]

```
  int frames = (int) g_cap.get(cv::CAP_PROP_FRAME_COUNT);
  int tmpw   = (int) g_cap.get(cv::CAP_PROP_FRAME_WIDTH);
  int tmph   = (int) g_cap.get(cv::CAP_PROP_FRAME_HEIGHT);
  cout << "Video has " << frames << " frames of dimensions("
       << tmpw << ", " << tmph << ")." << endl;
```

The core of the main program is the same as in Example 2-3, so we'll focus on what we've added. The first difference after opening the video is that we use `cv::g_cap.get()` to determine the number of frames in the video and the width and height of the video images. These numbers are printed out. We'll need the number of frames in the video to calibrate the slider (in the next step).

```
  createTrackbar("Position", "Example2_4", &g_slider_position, frames,
                 onTrackbarSlide);
```

Next we create the trackbar itself. The function `cv::createTrackbar()` allows us to give the trackbar a label[7] (in this case, `Position`) and to specify a window to put the trackbar in. We then provide a variable that will be bound to the trackbar, the maximum value of the trackbar (the number of frames in the video), and a callback (or NULL if we don't want one) for when the slider is moved.

```
    if( g_run != 0 ) {
      g_cap >> frame; if(!frame.data) break;
      int current_pos = (int)g_cap.get(cv::CAP_PROP_POS_FRAMES);
      g_dontset = 1;
      cv::setTrackbarPos("Position", "Example2_4", current_pos);
      cv::imshow( "Example2_4", frame );
      g_run-=1;
    }
```

---

[6] Because HighGUI is highly civilized, when a new video position is requested, it will automatically handle such issues as the possibility that the frame we have requested is not a key-frame; it will start at the previous key-frame and fast forward up to the requested frame without us having to fuss with such details.

[7] Because HighGUI is a lightweight, easy-to-use toolkit, `cv::createTrackbar()` does not distinguish between the name of the trackbar and the label that actually appears on the screen next to the trackbar. You may already have noticed that `cv::namedWindow()` likewise does not distinguish between the name of the window and the label that appears on the window in the GUI.

In the while loop, in addition to reading and displaying the video frame, we also get our current position in the video, set the `g_dontset` so that the next trackbar callback will not put us into single step mode, and then invoke the trackbar callback to update the position of the slider displayed to the user. The global `g_run` is decremented, which has the effect of either keeping us in single step mode or of letting the video run depending on its prior state set by user interaction via keyboard, as we'll see next.

```
    char c = (char) cv::waitKey(10);
    if(c == 's') // single step
      {g_run = 1; cout << "Single step, run = " << g_run << endl;}
    if(c == 'r') // run mode
      {g_run = -1; cout << "Run mode, run = " << g_run <<endl;}
    if( c == 27 )
      break;
```

At the bottom of the while loop, we look for keyboard input from the user. If 's' has been pressed, we go into single step mode (`g_run` is set to 1 which allows reading of a single frame above). If 'r' is pressed, we go into continuous video mode (`g_run` is set to -1 and further decrementing leaves it negative for any conceivable sized video). Finally, if ESC is pressed, the program will terminate. Note again, for short programs, we've omitted the step of cleaning up the window storage using `cv::destroyWindow()`.

A Simple Transformation

Great! Now you can use OpenCV to create your own video player, which will not be much different from countless video players out there already. But we are interested in computer vision, and we want to do some of that. Many basic vision tasks involve the application of filters to a video stream. We will modify the program we already have to do a simple operation on every frame of the video as it plays.

One particularly simple operation is the smoothing of an image, which effectively reduces the information content of the image by convolving it with a Gaussian or other similar kernel function. OpenCV makes such convolutions exceptionally easy to do. We can start by creating a new window called `"Example4-out"`, where we can display the results of the processing. Then, after we have called `cv::showImage()` to display the newly captured frame in the input window, we can compute and display the smoothed image in the output window. See Example 2-5.

Example 2-5: Loading and then smoothing an image before it is displayed on the screen

```
#include <opencv2/opencv.hpp>

void example2_5( cv::Mat & image ) {
  // Create some windows to show the input
  // and output images in.
  //
  cv::namedWindow( "Example2_5-in", cv::WINDOW_AUTOSIZE );
  cv::namedWindow( "Example2_5-out", cv::WINDOW_AUTOSIZE );

  // Create a window to show our input image
  //
  cv::imshow( "Example2_5-in", image );

  // Create an image to hold the smoothed output
  cv::Mat out;

  // Do the smoothing
  //  Could use GaussianBlur(), blur(), medianBlur() or bilateralFilter().
  cv::GaussianBlur( image, out, cv::Size(5,5), 3, 3);
  cv::GaussianBlur(   out, out, cv::Size(5,5), 3, 3);

  // Show the smoothed image in the output window
  //
  cv::imshow( "Example2_5-out", out );
```

```
  // Wait for the user to hit a key, windows will self destruct
  //
  cv::waitKey( 0 );
}
```

The first call to `cv::showImage()` is no different than in our previous example. In the next call, we allocate another image structure. Next, the C++ object `cv::Mat` makes life simpler for us; we just instantiate an output matrix "`out`" and it will automatically resize/reallocate and deallocate itself as necessary as it is used. To make this point clear, we use it in two consecutive calls to `cv::GaussianBlur()`. In the first call, the input images is blurred by a 5 × 5 Gaussian convolution filter and written to `out`. The size of the Gaussian kernel should always be given as odd numbers since the Gaussian kernel (specified here by `cv::Size(5,5)`) is computed at the center pixel in that area. In the next call to `cv::GaussianBlur()`, `out` is used as both the input and output since temporary storage is assigned for us in this case. The resulting double-blurred image is displayed and the routine then waits for any user keyboard input before terminating and cleaning up allocated data as it goes out of scope.

A Not-So-Simple Transformation

That was pretty good, and we are learning to do more interesting things. In Example 2-5, we used Gaussian blurring for no particular purpose. We will now use a function that uses Gaussian blurring to downsample it by a factor of 2 [Rosenfeld80]. If we downsample the image several times, we form a scale space, or image pyramid that is commonly used in computer vision to handle the changing scales in which a scene or object is observed.

For those who know some signal processing and the Nyquist-Shannon Sampling Theorem [Shannon49], when you downsample a signal (in this case, create an image where we are sampling every other pixel), it is equivalent to convolving with a series of delta functions (think of these as "spikes"). Such sampling introduces high frequencies into the resulting signal (image). To avoid this, we want to first run a high-pass filter over the signal first to band limit its frequencies so that they are all below the sampling frequency. In OpenCV, this Gaussian blurring and downsampling is accomplished by the function `cv::pyrDown()`. We use this very useful function in Example 2-6.

Example 2-6: Using `cv::pyrDown()` to create a new image that is half the width and height of the input image

```
#include <opencv2/opencv.hpp>

int main( int argc, char** argv ) {
  cv::Mat img = cv::imread( argv[1] ),img2;
  cv::namedWindow( "Example1", cv::WINDOW_AUTOSIZE );
  cv::namedWindow( "Example2", cv::WINDOW_AUTOSIZE );
  cv::imshow( "Example1", img );
  cv::pyrDown( img, img2);
  cv::imshow( "Example2", img2 );
  cv::waitKey(0);
  return 0;
};
```

Let's now look at a similar but slightly more involved example involving the *Canny edge detector* [Canny86] *cv::Canny()* (see Example 2-7). In this case, the edge detector generates an image that is the full size of the input image but needs only a single-channel image to write to and so we convert to a gray scale, single-channel image first using *cv::cvtColor()* with the flag to convert blue, green, red images to gray scale: *cv::BGR2GRAY*.

Example 2-7: The Canny edge detector writes its output to a single-channel (grayscale) image

```
#include <opencv2/opencv.hpp>

int main( int argc, char** argv ) {
```

```
    cv::Mat img_rgb = cv::imread( argv[1] );
    cv::Mat img_gry, img_cny;
    cv::cvtColor( img_rgb, img_gry, cv::BGR2GRAY);
    cv::namedWindow( "Example Gray", cv::WINDOW_AUTOSIZE );
    cv::namedWindow( "Example Canny", cv::WINDOW_AUTOSIZE );
    cv::imshow( "Example Gray", img_gry );
    cv::Canny( img_gry, img_cny, 10, 100, 3, true );
    cv::imshow( "Example Canny", img_cny );
    cv::waitKey(0);
}
```

This allows us to string together various operators quite easily. For example, if we wanted to shrink the image twice and then look for lines that were present in the twice-reduced image, we could proceed as in Example 2-8.

Example 2-8: Combining the pyramid down operator (twice) and the Canny subroutine in a simple image pipeline

```
cv::cvtColor( img_rgb, img_gry, cv::BGR2GRAY );
cv::pyrDown( img_gry, img_pyr );
cv::pyrDown( img_pyr, img_pyr2 );
cv::Canny( img_pyr2, img_cny, 10, 100, 3, true );
// do whatever with 'img_cny'
//
...
```

In Example 2-9, we show a simple way to read and write pixel values from Example 2-8.

Example 2-9: Getting and setting pixels in Example 2-8

```
  int x = 16, y = 32;
  cv::Vec3b intensity = img_rgb.at< cv::Vec3b >(y, x);
  uchar blue  = intensity.val[0]; // We could write img_rgb.at< cv::Vec3b >(x,y)[0]
  uchar green = intensity.val[1];
  uchar red   = intensity.val[2];
  std::cout << "At (x,y) = (" << x << ", " << y <<
            "): (blue, green, red) = (" <<
            (unsigned int)blue <<
            ", " << (unsigned int)green << ", " <<
            (unsigned int)red << ")" << std::endl;

  std::cout << "Gray pixel there is: " <<
               (unsigned int)img_gry.at<uchar>(x, y) << std::endl;

  x /= 4; y /= 4;
  std::cout << "Pyramid2 pixel there is: " <<
               (unsigned int)img_pyr2.at<uchar>(x, y) << std::endl;

  img_cny.at<uchar>(x, y) = 128; // Set the Canny pixel there to 128
```

Input from a Camera

Vision can mean many things in the world of computers. In some cases, we are analyzing still frames loaded from elsewhere. In other cases, we are analyzing video that is being read from disk. In still other cases, we want to work with real-time data streaming in from some kind of camera device.

OpenCV—more specifically, the HighGUI portion of the OpenCV library—provides us with an easy way to handle this situation. The method is analogous to how we read videos from disk since the `cv::VideoCapture` object works the same for files on disk or from camera. For the former, you give it a path/filename, and for the latter, you give it a camera ID number (typically "0" if only one camera is connected to the system). The default value is –1, which means "just pick one"; naturally, this also works

quite well when there is only one camera to pick (see Chapter 4 for more details). Camera capture from file or from camera is demonstrated in Example 2-10.

Example 2-10: The same object can load videos from a camera or a file

```
#include <opencv2/opencv.hpp>
#include <iostream>

int main( int argc, char** argv ) {
  cv::namedWindow( "Example2_10", cv::WINDOW_AUTOSIZE );
  cv::VideoCapture cap;
  if (argc==1) {
    cap.open(0);           // open the default camera
  } else {
    cap.open(argv[1]);
  }
  if( !cap.isOpened() ) {  // check if we succeeded
    std::cerr << "Couldn't open capture." << std::endl;
    return -1;
  }
  // The rest of program proceeds as in Example 2-3
  …
```

In Example 2-10, if a filename is supplied, it opens that file just like in Example 2-3, and if no filename is given, it attempts to open camera zero (`0`). We have added a check that something actually opened that will report an error if not.

Writing to an AVI File

In many applications, we will want to record streaming input or even disparate captured images to an output video stream, and OpenCV provides a straightforward method for doing this. Just as we are able to create a capture device that allows us to grab frames one at a time from a video stream, we are able to create a writer device that allows us to place frames one by one into a video file. The object that allows us to do this is `cv::VideoWriter`.

Once this call has been made, we may stream each frame to the `cv::VideoWriter` object, and finally call its `cv::VideoWriter.release()` method when we are done. Just to make things more interesting, Example 2-11 describes a program that opens a video file, reads the contents, converts them to a log-polar format (something like what your eye actually sees, as described in Chapter 6), and writes out the log-polar image to a new video file.

Example 2-11: A complete program to read in a color video and write out the log-polar transformed video

```
#include <opencv2/opencv.hpp>
#include <iostream>

int main( int argc, char* argv[] ) {
  cv::namedWindow( "Example2_10", cv::WINDOW_AUTOSIZE );
  cv::namedWindow( "Log_Polar", cv::WINDOW_AUTOSIZE );
  cv::VideoCapture capture;
  double fps = capture.get( cv::CAP_PROP_FPS );
  cv::Size size(
    (int)capture.get( cv::CAP_PROP_FRAME_WIDTH ),
    (int)capture.get( cv::CAP_PROP_FRAME_HEIGHT )
  );
  cv::VideoWriter writer;
  writer.open( argv[2], CV_FOURCC('M','J','P','G'), fps, size );
  cv::Mat logpolar_frame(size,CV::U8C3), bgr_frame;
  for(;;) {
    capture >> bgr_frame;
    if( bgr_frame.empty() ) break; // end if done
```

```
    cv::imshow( "Example2_10", bgr_frame );
    cv::logPolar(
      bgr_frame,                      // Input color frame
      logpolar_frame,                 // Output log-polar frame
      cv::Point2f(                    // Centerpoint for log-polar transformation
        bgr_frame.cols/2,             //  x
        bgr_frame.rows/2              //  y
      ),
      40,                             // Magnitude (scale parameter)
      cv::WARP_FILL_OUTLIERS          // Fill outliers with 'zero'
    );
    cv::imshow( "Log_Polar", logpolar_frame );
    writer << logpolar_frame;
    char c = cv::waitKey(10);
    if( c == 27 ) break;        // allow the user to break out
  }
  capture.release();
}
```

Looking over this program reveals mostly familiar elements. We open one video and read some properties (frames per second, image width and height) that we'll need to open a file for the `cv::VideoWriter` object. We then read the video frame by frame from the `cv::VideoReader` object, convert the frame to log-polar format, and write the log-polar frames to this new video file one at a time until there are none left or until the user quits by pressing ESC. Then we close up.

The call to `cv::VideoWriter` object contains several parameters that we should understand. The first is just the filename for the new file. The second is the *video codec* with which the video stream will be compressed. There are countless such codecs in circulation, but whichever codec you choose must be available on your machine (codecs are installed separately from OpenCV). In our case, we choose the relatively popular *MJPG* codec; this is indicated to OpenCV by using the macro `CV_FOURCC()`, which takes four characters as arguments. These characters constitute the "four-character code" of the codec, and every codec has such a code. The four-character code for *motion jpeg* is "MJPG," so we specify that as `CV_FOURCC('M','J','P','G')`. The next two arguments are the replay frame rate and the size of the images we will be using. In our case, we set these to the values we got from the original (color) video.

Summary

Before moving on to the next chapter, we should take a moment to take stock of where we are and look ahead to what is coming. We have seen that the OpenCV API provides us with a variety of easy-to-use tools for reading and writing still images and videos from and to files along with capturing video from cameras. We have also seen that the library contains primitive functions for manipulating these images. What we have not yet seen are the powerful elements of the library, which allow for more sophisticated manipulation of the entire set of abstract data types that are important to practical vision problem solving.

In the next few chapters, we will delve more deeply into the basics and come to understand in greater detail both the interface-related functions and the image data types. We will investigate the primitive image manipulation operators and, later, some much more advanced ones. Thereafter, we will be ready to explore the many specialized services that the API provides for tasks as diverse as camera calibration, tracking, and recognition. Ready? Let's go!

Exercises

Download and install OpenCV if you have not already done so. Systematically go through the directory structure. Note in particular the *docs* directory, where you can load *index.htm*, which further links to the main documentation of the library. Further explore the main areas of the library. The *core* module contains the basic data structures and algorithms, *imgproc* contains the image processing and vision algorithms, *ml*

includes algorithms for machine learning and clustering, and *highgui* contains the I/O functions. Check out the *…/samples/cpp* directory, where many useful examples are stored.

1. Using the install and build instructions in this book or on the website *http://opencv.org*, build the library in both the debug and the release versions. This may take some time, but you will need the resulting library and *dll* files. Make sure you set the *cmake* file to build the samples *…/opencv/samples/* directory.
2. Go to where you built the *…/opencv/samples/* directory (the authors build in *…/trunk/eclipse_build/bin*) and look for *lkdemo.c* (this is an example motion tracking program). Attach a camera to your system and run the code. With the display window selected, type "r" to initialize tracking. You can add points by clicking on video positions with the mouse. You can also switch to watching only the points (and not the image) by typing "n." Typing "n" again will toggle between "night" and "day" views.
3. Use the capture and store code in Example 2-11 together with the `cv::PyrDown()` code of Example 2-6 to create a program that reads from a camera and stores down-sampled color images to disk.
4. Modify the code in Exercise 3 and combine it with the window display code in Example 2-2 to display the frames as they are processed.
5. Modify the program of Exercise 4 with a slider control from Example 2-4 so that the user can dynamically vary the pyramid downsampling reduction level by factors of between 2 and 8. You may skip writing this to disk, but you should display the results.

3

Getting to Know OpenCV

OpenCV Data Types

OpenCV has many data types, which are designed to make the representation and handling of important concepts of computer vision relatively easy and intuitive. At the same time, many algorithm developers require a set of relatively powerful primitives that can be generalized or extended for their particular needs. OpenCV attempts to address both of these needs through the use of templates for fundamental data types, and specializations of those templates that make everyday operations easier.

From an organizational perspective, it is convenient to divide the data types into three major categories. First, the *basic data types* are those that are assembled directly from C++ primitives (`int`, `float`, etc.). These types include simple vectors and matrices, as well as representations of simple geometric concepts like points, rectangles, sizes, and the like. The second category contains *helper objects*. These objects represent more abstract concepts such as the garbage collecting pointer class, range objects used for slicing, and abstractions such as termination criteria. The third category is what might be called *large array types*. These are objects whose fundamental purpose is to contain arrays or other assemblies of primitives or, more often, the basic data types mentioned first. The star example of this latter group is the `cv::Mat` class, which is used to represent arbitrary-dimensional arrays containing arbitrary basic elements. Objects such as images are specialized uses of the `cv::Mat` class but, unlike in earlier versions of OpenCV (i.e., before version 2.1), such specific use does not require a different class or type. In addition to `cv::Mat`, this category contains related objects such as the sparse matrix `cv::SparseMat` class, which is more naturally suited to non-dense data such as histograms.

In addition to these types, OpenCV also makes heavy use of the *Standard Template Library* (STL) [STL]. This vector class is particularly relied on by OpenCV, and many OpenCV library functions now have vector template objects in their argument lists. We will not cover STL in this book, other than as necessary to explain relevant functionality. If you are already comfortable with STL, many of the template mechanisms used "under the hood" in OpenCV will be familiar to you.

Overview of the Basic Types

The most straightforward of the basic data types is the template class `cv::Vec<>`. `cv::Vec<>` is a container class for primitives,[1] which we will refer to as "the fixed vector classes." Why not just use

[1] Actually, this is an oversimplification that we will clear up a little later in the chapter. In fact, `cv::Vec<>` is a vector container for anything, and uses templating to create this functionality. As a result, `cv::Vec<>` can contain other

standard template library classes? The key difference is that the fixed vector classes are classes intended for small vectors whose dimensions are known at compile time. This allows for efficient code. What "small" means in practice is that if you have more than just a few elements, you are probably using the wrong class. (In fact, as of version 2.2, this number cannot exceed nine in any case). We will look shortly at the `cv::Mat` class, which is the right way to handle big arrays of any number of dimensions, but for now, think of the fixed vector classes as being handy and speedy for little guys.

Even though `cv::Vec<>` is a template, you will not tend to see or use it in that form most of the time. Instead, there are aliases (`typedef`'s) for common instantiations of the `cv::Vec<>` template. They have names like `cv::Vec2i`, `cv::Vec3i`, or `cv::Vec4d` (for a two-element integer vector, a three-element integer vector, or a four-element double-precision floating-point vector, respectively). In general, anything of the form `cv::Vec{2,3,4,6}{b,s,i,f,d}` is valid for any combination of two to six dimensions[2] and the five data types.

In addition to the fixed vector class, there are also "fixed matrix classes." They are associated with the template `cv::Matx<>`. Just like the fixed vector classes, `cv::Matx<>` is not intended to be used for large arrays, but rather is designed for the handling of certain specific small matrix operations. In computer vision, there are a lot of 2-by-2 or 3-by-3 matrices running around, and a few 4-by-4, which are used for various transformations and the like. `cv::Matx<>` is designed to hold these sorts of objects. As with `cv::Vec<>`, `cv::Matx<>` is normally accessed through aliases of the form `cv::Matx{1,2,3,4,6}{1,2,3,4,6}{f,d}`. It is important to notice that with the fixed matrix classes (like the fixed vector classes, but unlike `cv::Mat`, which we will come to shortly), the dimensionality of the fixed matrix classes must be known at compile time. Of course, it is precisely this knowledge that makes it possible to make operations with the fixed matrix classes highly efficient.

Closely related to the fixed vector classes are the point classes. The point classes are containers for two or three values of one of the primitive types. The point classes are derived from their own template, so they are not directly descended from the fixed vector classes, but they can be cast to and from them. The main difference between the point classes and the fixed vector classes is that their members are accessed by named variables (`mypoint.x`, `mypoint.y`, etc.) rather than by a vector index (`myvec[0]`, `myvec[1]`, etc.). As with `cv::Vec<>`, the point classes are typically invoked using aliases for the instantiation of an appropriate template. Those aliases have names like `cv::Point2i`, `cv::Point2f`, or `cv::Point2d`, or `cv::Point3i`, `cv::Point3f`, or `cv::Point3d`.

The class `cv::Scalar` is essentially a four-dimensional point. As with the point classes, `cv::Scalar` is actually associated with a template that can generate an arbitrary four-component vector, but the keyword `cv::Scalar` specifically is aliased to a four-component vector with double-precision components. Unlike the point classes, the elements of a `cv::Scalar` object are accessed with an integer index, the same as `cv::Vec<>`. This is because `cv::Scalar` is directly derived from an instantiation of `cv::Vec<>` (specifically, from `cv::Vec<double,4>`).

Next on our tour are `cv::Size` and `cv::Rect`. As with the point classes, these two are derived from their own templates. `cv::Size` is mainly distinguished by having data members `width` and `height` rather than `x` and `y`, while `cv::Rect` has all four. The class `cv::Size` is actually an alias for `cv::Size2i`, which is itself an alias of a more general template in the case of `width` and `height` being integers. For floating-point values of `width` and `height`, use the alias `cv::Size2f`. Similarly, `cv::Rect` is an alias for the integer form of rectangle. There is also a class to represent a rectangle that is not axis-aligned. It is called `cv::RotatedRect` and contains a `cv::Point2f` called `center`, a `cv::Size2f` called `size`, and one additional `float` called `angle`.

---

class objects, either from OpenCV or elsewhere. In most usages, however, `cv::Vec` is used as a container for C primitive types like `int` or `float`.

[2] Except five, for some reason five is not an option.

Basic Types: Getting Down to Details

Each of the basic types is actually a relatively complicated object, supporting its own interface functions, overloaded operators, and the like. In this section, we will take a somewhat more encyclopedic look at what each type offers, and how some of the otherwise similar appearing types differ from one another.

As we go over these classes, we will try to hit on the high points of their interfaces, but not get into every gory detail. Instead, we will provide examples that should convey what you can and can't do with these objects. For the lowest-level details, you should consult *…/opencv2/core/core.hpp*.

The Point Classes

Of the OpenCV basic types, the point classes are probably the simplest. As we mentioned earlier, these are implemented based on a template structure, such that there can be points of any type: integer, floating-point, and so on. There are actually two such templates, one for two-dimensional and one for three-dimensional points. The big advantage of the point classes is that they are simple and have very little overhead. Natively, they do not have a lot of operations defined on them, but they can be cast to somewhat more generalized types, such as the fixed vector classes or the fixed matrix classes (discussed later), when needed.

In most programs, the point classes are instantiated using aliases that take forms like `cv::Point2i` or `cv::Point3f`, with the last letter indicating the desired primitive from which the point is to be constructed. (Here, `b` is an unsigned character, `s` is a short integer, `i` is a 32-bit integer, `f` is a 32-bit floating-point number, and `d` is a 64-bit floating-point number.)

Table 3-1 is the (relatively short) list of functions natively supported by the point classes. It is important to note that there are several important operations that are supported, but they are supported indirectly through implicit casting to the fixed vector classes (see below). These operations notably contain all of the vector and singleton[3] overloaded algebraic operators and comparison operators.

Table 3-1: Operations supported directly by the point classes

Operation	Examples
Default constructors	`cv::Point2i p();` `cv::Point3f p();`
Copy constructor	`cv::Point3f p2( p1 );`
Value constructors	`cv::Point2i p( x0, x1 );` `cv::Point3d p( x0, x1, x2 );`
Cast to the fixed vector classes	`(cv::Vec3f) p;`

Member access	`p.x; p.y; // and for three-dimensional`
	`// point classes: p.z`

Dot product	`float x = p1.dot( p2 )`
Double-precision dot	`double x = p1.ddot( p2 )`

[3] You might have expected us to use the word "scalar" here, but we avoided doing so because `cv::Scalar` is an existing class in the library. As you will see shortly, a `cv::Scalar` in OpenCV is (somewhat confusingly) an array of four numbers, approximately equivalent to a `cv::Vec` with four elements! In this context, the word "singleton" can be understood to mean "a single object of whatever type the vector is an array of."

product	
Cross product	`p1.cross( p2 ) // (for three-dimensional point classes only)`
Query if point *p* is inside of rectangle *r*	`p.inside( r ) // (for two-dimensional point classes only)`

These types can be cast to and from the old C interface types `CvPoint` and `CvPoint2D32f`. In the case in which a floating-point-valued instance of one of the point classes is cast to `CvPoint`, the values will automatically be rounded.

`class cv::Scalar`

The class `cv::Scalar` is really a four-dimensional point class. Like the others, it is actually associated with a template class, but the alias for accessing it returns an instantiation of that template in which all of the members are double-precision floating-point numbers. The `cv::Scalar` class also has some special member functions associated with uses of four-component vectors in computer vision.

Table 3-2: Operations supported directly by class `cv::Scalar`

Operation	**Example**
Default constructors	`cv::Scalar s();`
Copy constructor	`cv::Scalar s2( s1 );`
Value constructors	`cv::Scalar s( x0 );` `cv::Scalar s( x0, x1, x2, x3 );`
Element-wise multiplication	`s1.mul( s2 );`
(Quaternion) conjugation	`s.conj(); // (returns cv::Scalar(s0,-s1,-s2,-s2))`
(Quaternion) real test	`s.isReal(); // (returns true iff s1==s2==s3==0)`

You will notice that for `cv::Scalar`, the operation "cast to the fixed vector classes" does not appear in *Table 3-2* (as it did in the case of the point classes). This is because, unlike the point classes, `cv::Scalar` inherits directly from an instantiation of the fixed vector class template. As a result, it inherits all of the vector algebra operations, member access functions (i.e., `operator[]`), and other properties from the fixed vector classes. We will get to that class later, but for now, just keep in mind that `cv::Scalar` is shorthand for a four-dimensional double-precision vector that has a few special member functions attached that are useful for various kinds of four-vectors.

The class `cv::Scalar` can be freely cast to and from the old C interface `CvScalar` type.

The Size Classes

The size classes are, in practice, similar to the corresponding point classes, and can be cast to and from them. The primary difference between the two is that the point class' data members are named `x` and `y`, while the corresponding data members in the size classes are named `width` and `height`. The three aliases for the size classes are `cv::Size`, `cv::Size2i`, and `cv::Size2f`. The first two of these are equivalent and imply integer size, while the last is for 32-bit floating-point sizes. As with the point classes, the size classes can be cast to and from the corresponding old-style OpenCV classes (in this case, `CvSize` and `CvSize2D32f`).

Table 3-3: Operations supported directly by the size classes

Operation	Example

Default constructors	`cv::Size sz();`
	`cv::Size2i sz();`
	`cv::Point2f sz();`

Copy constructor	`cv::Size sz2( sz1 );`
Value constructors	`cv::Size2f sz( w, h );`
Member access	`sz.width; sz.height;`
Compute area	`sz.area();`

Unlike the point classes, the size classes do not support casting to the fixed vector classes. This means that the size classes have more restricted utility. On the other hand, the point classes and the fixed vector classes can be cast to the size classes without any problem.

`class cv::Rect`

The rectangle classes include the members `x` and `y` of the point class (representing the upper-left corner of the rectangle) and the members `width` and `height` of the size class (representing the extent of the rectangle). The rectangle classes, however, do not inherit from the point or size classes, and so in general they do not inherit operators from them.

Table 3-4: Operations supported directly by class `cv::Rect*`

Operation	Example
Default constructors	`cv::Rect r();`
Copy constructor	`cv::Rect r2( r1 );`
Value constructors	`cv::Rect( x, y, w, h );`
Construct from origin and size	`cv::Rect( p, sz );`
Construct from two corners	`cv::Rect( p1, p2 );`
Member access	`r.x; r.y; r.width; r.height;`
Compute area	`r.area();`
Extract upper-left corner	`r.tl();`
Extract lower-right corner	`r.lr;`
Determine if point *p* is inside of rectangle *r*	`r.contains( p );`

Cast operators and copy constructors exist to allow `cv::Rect` to be computed from or cast to the old-style `cv::CvRect` type as well. `cv::Rect` is actually an alias for a rectangle template instantiated with integer members.

The class `cv::Rect` also supports a variety of overloaded operators that can be used for the computation of various geometrical properties of two rectangles or a rectangle and another object.

Table 3-5. Overloaded operators *that* take objects of type `cv::Rect`

Operation	Example
Intersection of rectangles r1 and r2	`cv::Rect r3 = r1 & r2;` `r1 &= r2;`
Minimum area rectangle containing rectangles r1 and r2	`cv::Rect r3 = r1 | r2;` `r1 |= r2;`

Translate rectangle r by an amount x	`cv::Rect rx = r + v; // v is a cv::Point2i`
	`r += v;`

Enlarge a rectangle r by an amount given by size s	`cv::Rect rs = r + s; // s is a cv::Point2i`
	`r += s;`

Compare rectangles r1 and r2 for exact equality	`bool eq = (r1 == r2);`
Compare rectangles r1 and r2 for inequality	`bool ne = (r1 != r2);`

`class cv::RotatedRect`

The `cv::RotatedRect` class is one of the few classes in the C++ OpenCV interface that is not a template underneath. It is a container, which holds a `cv::Point2f` called `center`, a `cv::Size2f` called `size`, and one additional `float` called `angle`, with the latter representing the rotation of the rectangle around `center`. One very important difference between `cv::RotatedRect` and `cv::Rect` is the convention that a `cv::RotatedRect` is located in "space" relative to its center, while the `cv::Rect` is located relative to its upper-left corner.

Table 3-6: Operations supported directly by class `cv::RotatedRect`

Operation	Example
Default constructors	`cv::RotatedRect rr();`
Copy constructor	`cv::RotatedRect rr2( rr1 );`
Construct from two corners	`cv::RotatedRect( p1, p2 );`
Value constructors; takes a point, a size, and an angle	`cv::RotatedRect rr( p, sz, theta ) ;`
Member access	`rr.center; rr.size; rr.angle;`
Return a list of the corners	`rr.points( pts[4] );`

The Fixed Matrix Classes

The fixed matrix classes are for matrices whose dimensions are known at compile time (hence "fixed"). As a result, all memory for their data is allocated on the stack, which means that they allocate and clean up quickly. Operations on them are fast and there are specially optimized implementations for small matrices (2-*by*-2, 3-*by*-3, etc.). The fixed matrix classes are also central to many of the other basic types in the C++ interface to OpenCV. The fixed vector class derives from the fixed matrix classes and other classes either derive from the fixed vector class (like `cv::Scalar`), or they rely on casting to the fixed vector class for many important operations. As usual, the fixed matrix classes are really a template. The template is called

`cv::Matx<>`, but individual matrices are usually allocated using aliases. The basic form of such an alias is `cv::Matx{1,2,...}{1,2,...}{f,d}`, where the numbers can be any number from one to six, and the trailing letter has the same meaning as with the previous types.[4]

In general, you should use the fixed matrix classes when you are representing something that is really a matrix with which you are going to do matrix algebra. If your object is really a big data array, like an image, or a huge list of points or something like that, the fixed matrix classes are not the correct solution; you should be using `cv::Mat` (which we will get to shortly). Fixed matrix classes are for small matrices where you know the size at compile time: (e.g., a camera matrix).

Table 3-7: Operations supported by class `cv::Matx`

Operation	Example
Default constructor	`cv::Matx33f m33f(); cv::Matx43d m43d();`
Copy constructor	`cv::Matx22d m22d( n22d );`
Value constructors	`cv::Matx21f m(x0,x1); cv::Matx44d m(x0,x1,x2,x3,x4,x5,x6,x7,x8,x9,x10,x11,x12,x13,x14,x15);`
Matrix of identical elements	`m33f = cv::Matx33f::all( x );`
Matrix of zeros	`m23d = cv::Matx23d::zeros();`
Matrix of ones	`m16f = cv::Matx16f::ones();`
Create a unit matrix	`m33f = cv::Matx33f::eye();`
Create a matrix that can hold the diagonal of another	`m31f = cv::Matx33f::diag(); // create a matrix of // size 3-by-1 of floats`
Create a matrix with uniformly distributed entries	`m33f = cv::Matx33f::randu( min, max );`
Create a matrix with normally distributed entries	`m33f = cv::Matx33f::nrandn( mean, variance );`
Member access	`m( i, j ), m( i ); // one argument for // one-dimensional matrices only`
Matrix algebra	`m1 = m0; m0 * m1; m0 + m1; m0 - m1;`
Singleton algebra	`m * a; a * m; m / a;`
Comparison	`m1 == m2; m1 != m2;`
Dot product	`m1.dot( m2 );  // (sum of element-wise`

[4] At the time of writing, the relevant header file called *core.hpp* does not actually contain every possible combination of these integers. For example, there is no 1-by-1 matrix alias, nor is there a 5-by-5. This may or may not change in later releases, but you will pretty much never want the missing ones anyway. If you really do want one of the odd ones, you can just instantiate the template yourself (e.g., `cv::Matx<5,5,float>`).

	`// multiplications, precision of m)`
Dot product	`m1.ddot( m2 ); // (sum of element-wise multiplications,` `// double precision)`
Reshape a matrix	`m91f = m33f.reshape<9,1>();`
Cast operators	`m44f = (Matx44f) m44d`
Extract i,j-th algebraic minor	`m44f.get_minor( i, j );`
Extract row *i*	`m14f = m44f.row( i );`
Extract column *j*	`m41f = m44f.col( j );`
Extract matrix diagonal	`m41f = m44f.diag();`
Compute transpose	`n44f = m44f.t();`
Invert matrix	`n44f = m44f.inv( method ); // (default method is` `// cv::DECOMP_LU)`

Solve linear system	`m31f = m33f.solve( rhs31f, method )`
	`m32f = m33f.solve<2>( rhs32f, method ); // (template`
	`// form[5]; default method is DECOMP_LU)`

Per-element multiplication	`m1.mul( m2 );`

Note that many of the fixed matrix functions are static relative to the class (i.e., you access them directly as members of the class rather than as members of a particular object). For example, if you would like to construct a 3-by-3 identity matrix, you have a handy class function for it: `cv::Mat33f::eye()`. Note that, in this example, `eye()` does not need any arguments because it is a member of the class, and the class is already a specialization of the `cv::Matx<>` template to 3-by-3.

The Fixed Vector Classes

The fixed vector classes are derived from the fixed matrix classes. They are really just convenience functions for `cv::Matx<>`. In the proper sense of C++ inheritance, it is correct to say that the fixed vector template `cv::Vec<>` is a `cv::Matx<>` whose number of columns is one. The readily available aliases for specific instantiations of `cv::Vec<>` are of the form `cv::Vec{2,3,4,6}{b,s,w,i,f,d}`, where the last character has the usual meanings (with the addition of `w`, which indicates an `unsigned short`).

Table 3-8: Operations supported by class `cv::Vec`

Operation	**Example**
Default constructor	`Vec2s v2s(); Vec6f v6f();  // etc…`

[5] The template form is used when the right hand side of the implied matrix equation has multiple columns. In this case, we are essentially solving for 'k' different systems at once. This value of 'k' must be supplied as the template argument to `solve<>()`. It will also determine the number of columns in the result matrix.

Copy constructor	`Vec3f u3f( v3f );`
Value constructors	`Vec2f v2f(x0,x1); Vec6d v6d(x0,x1,x2,x3,x4,x5);`
Member access	`v4f[ i ]; v3w( j ); // (operator() and operator[]` `// both work)`
Vector cross-product	`v3f.cross( u3f );`

The primary conveniences of the fixed vector classes are the ability to access elements with a single ordinal, and a few specific additional functions that would not make sense for a general matrix (e.g., cross product). We can see this in Table 3-8 by the relatively small number of novel methods added to the large number of methods inherited from the fixed matrix classes.

The Complex Number Classes

One more class type should be included in the basic types: the complex number classes. The OpenCV complex number classes are not identical to, but are compatible with, and can be cast to and from, the classes associated with the STL complex number class template `complex<>`. The most substantial difference between the OpenCV and STL complex number classes is in member access. In the STL classes, the real and imaginary parts are accessed through the member functions `real()` and `imag()`, while in the OpenCV class, they are directly accessible as (public) member variables `re` and `im`.

Table 3-9: Operations supported by the OpenCV complex number classes

Operation	**Example**
Default constructor	`cv::Complexf z1; cv::Complexd z2;`
Copy constructor	`cv::Complexf z2( z1 );`
Value constructors	`cv::Complexd z1(re0); cv::Complexd(re0,im1) ;`
Copy constructor	`cv::Complexf u2f( v2f );`
Member access	`z1.re; z1.im;`
Complex conjugate	`z2 = z1.conj();`

Like many basic types, the complex classes are aliases for underlying templates. `cv::Complexf` and `cv::Complexd` are aliases for single- and double-precision complex floating point numbers, respectively.

Helper Objects

In addition to the basic types and the big containers (which we will get to in the next section), there is a family of helper objects, which are important for controlling various algorithms (such as termination criteria) or for doing various operations on the containers (such as “ranges” or “slices”). There is also one very important object, the “smart” pointer object `cv::Ptr`. Looking into `cv::Ptr`, we will examine the garbage collecting system, which is integral to the C++ interface to OpenCV. This system allows us to not have to worry about the details of allocation and deallocation of objects in the manner that was once so onerous in the earlier C-based OpenCV interface (i.e., before version 2.1).

class cv::TermCriteria

Many algorithms require a stopping condition to know when to quit. Generally, stopping criteria take the form of either some finite number of iterations that are allowed (called `COUNT` or `MAX_ITER`) or some kind of error parameter that basically says "if you are this close, you can quit" (called `EPS`—short for "epsilon," everyone's favorite tiny number). In many cases, it is desirable to have both of these at once, so that if the algorithm never gets "close enough," then it will still quit at some point.

`cv::TermCriteria` objects encapsulate one, the other, or both of the stopping criteria so that they can be passed conveniently to an OpenCV algorithm function. They have three member variables, `type`, `maxCount`, and `epsilon`, which can be set directly (they are public) or, more often, are just set by the constructor with the form `TermCriteria( int type, int maxCount, double epsilon )`. The variable `type` is set to one of the following values: `cv::TermCriteria::COUNT` or `TermCriteria::EPS`. You can also "or" (i.e., `|`) the two together. The value `cv::TermCriteria::COUNT` is a synonym for `cv::TermCriteria::MAX_ITER`, so you can use that if you prefer. If the termination criterion includes `cv::TermCriteria::COUNT`, then you are telling the algorithm to terminate after `maxCount` iterations. If the termination criterion includes `cv::TermCriteria::EPS`, then you are telling the algorithm to terminate after some metric associated with the algorithm's convergence falls below `epsilon`[6] The `type` argument has to be set accordingly for `maxCount` or `epsilon` to be used.

class cv::Range

The `cv::Range` class is used to specify a continuous sequence of integers. `cv::Range` objects have two elements, `start` and `end`, which—similar to `cv::TermCriteria`—are often set with the constructor `cv::Range( int start, int end )`. Ranges are inclusive of their start value, but not inclusive of their end value, so `cv::Range rng( 0, 4 )` includes the values `0`, `1`, `2`, `3`, but not `4`.

Using `size()`, one can find the number of elements in a range. From the above, `rng.size()` would be equal to `4`. There is also a member `empty()` that tests if a range has no elements. Finally, `cv::Range::all()` can be used anywhere a range is required to indicate whatever range the consumer has available.

The cv::Ptr template, and Garbage Collection 101

One very useful object type in C++ is a "smart" pointer.[7] This pointer allows us to create a reference to a thing, and then pass that around. You can create more references to that thing, and then all of those references will be counted. As references go out of scope, the reference count for the smart pointer is decremented. Once all of the references (instances of the pointer) are gone, the "thing" will automatically be cleaned up (deallocated). The programmer doesn't have to do this bookkeeping anymore.

The way this all works is as follows: first, you define an instance of the pointer template for the class object that you want to "wrap." You do this with a call like `cv::Ptr<Matx33f> p( new cv::Matx33f )`. The constructor for the template object takes a pointer to the object to be pointed to. Once you do this, you have your smart pointer `p`, which is a sort of pointer-like object that you can pass around and use just like a normal pointer (i.e., it supports operators such as `operator*()` and `operator->()`). Once you have `p`, you can create other objects of the same type without passing them a pointer to a new object. For example, you could create: `Ptr<Mat33f> q`, and when you assign the value of `p` to `q`, somewhere behind the scenes, the "smart" action of the smart pointer comes into play. You see, just like a usual pointer, there is still only one actual `cv::Mat33f` object out there that `p` and `q` both point to. The

---

[6] The exact termination criteria are clearly algorithm dependent, but the documentation will always be clear how a particular algorithm interprets `epsilon`.

[7] If you are familiar with some of the more recent additions to the C++ standard, you will recognize a similarity between the OpenCV `cv::Ptr<>` template and the `smart_ptr<>` template. Similarly, there is a smart pointer `shared_ptr<>` in the Boost library. Ultimately, they all function more or less the same.

difference is that both p and q *know* that they are each one of two pointers. Should p disappear (such as by falling out of scope), q knows that it is the only remaining reference to the original matrix. If q should then disappear and its destructor is called (implicitly), q will know that is the last one left, and that it should deallocate the original matrix. You can think of this like the last person out of a building being responsible for turning out the lights and locking the door (and in this case, burning the building to the ground as well).

The cv::Ptr<> template class supports several additional functions in its interface related to the reference counting functionality of the smart pointer. Specifically, the functions addref() and release() increment and decrement the internal reference counter of the pointer. These are relatively dangerous functions to use, but are available in case you need to micromanage the reference counters yourself.

There is also a function empty(), which can be used to determine if a smart pointer is pointing to an object that has been deallocated. This could happen if you called release() on the object one or more times. In this case, you would still have a smart pointer around, but the object pointed to might already have been destroyed. There is a second application of empty(), which is to determine if the internal object pointer inside of the smart pointer object happens to be NULL for some other reason. This might occur if you assigned the smart pointer by calling a function that might just return NULL in the first place (cvLoadImage(),fopen(), etc.)[8]

The final member of Ptr<> that you will want to know about is delete_obj(). This is a function that gets called automatically when the reference count gets to zero. By default, this function is defined but does nothing. It is there so that you can overload it in the case of instantiation of cv::Ptr<>, which points to a class that requires some specific operation in order to clean up the class to which it points. For example, let's say that you are working with an old-style (pre-version 2.1) IplImage.[9] In the old days, you might, for example, have called cvLoadImage() to load that image from disk. In the C interface, that would have looked like:

```
IplImage* img_p = cvLoadImage( … );
```

The modern version of this (while still using IplImage, rather than cv::Mat, which we are still working our way up to) would look like:

```
Ptr<IplImage> img_p = cvLoadImage( "an_image" );
```

or (if you prefer):

```
Ptr<IplImage> img_p( cvLoadImage("an_image" ) );
```

Now you can use img_p in exactly the same way as a pointer (which is to say, for readers experienced with the pre-version 2.1 interface, "exactly as you would have back then"). Conveniently, this particular template instantiation is actually already defined for you somewhere in the vast sea of header files that make up OpenCV. If you were to go search it out, you would find the following template function defined:

```
template<> inline void cv::Ptr<IplImage>::delete_obj() {
    cvReleaseImage(&obj);
}
```

(The variable obj is the name of the class member variable inside of Ptr<> that actually holds the pointer to the allocated object.) As a result of this definition, you will not need to deallocate the IplImage* pointer you got from cvLoadImage(). Instead, it will be automatically deallocated for you when img_p falls out of scope.

---

[8] For the purposes of this example, we will make reference to IplImage and cvLoadImage(), both constructs from the ancient pre-version 2.1 interface that are now deprecated. We won't really cover them in detail in this book, but all you need to know for this example is that IplImage is the old data structure for images, and cvLoadImage() was the old function to get an image from disk and return a pointer to the resulting image structure.

[9] This example might seem a bit artificial, but in fact, if you have a large body of pre-v2.1 code you are trying to modernize, you will likely find yourself doing an operation like this quite often.

This example was a somewhat special (though highly relevant) situation, in that the case of a smart pointer to `IplImage` is sufficiently salient that it was defined for you by the library. In a somewhat more typical case, when the clean-up function does not exist for what you want, you will have to define it yourself. Consider the example of creating a file handle using a smart pointer to `FILE`.[10] In this case, we define our own overloaded version of `delete_obj()` for the `cv::Ptr<FILE>` template:

```
template<> inline void cv::Ptr<FILE>::delete_obj() {
    fclose(obj);
}
```

Then you could go ahead and use that pointer to open a file, do whatever with it, and later, just let the pointer fall out of scope (at which time the file handle would automatically be closed for you):

```
{
  Ptr<FILE> f(fopen("myfile.txt", "r"));
  if(f.empty())
    throw …;        // Throw an exception, we will get to this later on…
  fprintf(f, …);
  …
}
```

At the final brace, `f` falls out of scope, the internal reference count in `f` goes to zero, `delete_obj()` is called by `f`'s destructor, and (thus) `fclose()` is called on the file handle pointer (stored in `obj`).

> A tip for gurus: a serious programmer might worry that the incrementing and decrementing of the reference count might not be sufficiently atomic for the `Ptr<>` template to be safe in multithreaded applications. This, however, is not the case, and `Ptr<>` is thread safe. Similarly, the other reference counting objects in OpenCV are all thread-safe in this same sense.

`class cv::Exception` and Exception Handling

OpenCV uses exceptions to handle errors. OpenCV defines its own exception type `cv::Exception`, which is derived from the `STL` exception class `std::exception`. Really, this exception type has nothing special about it, other than being in the `cv::` namespace and so distinguishable from other objects that are also derived from `std::exception`.

The type `cv::Exception` has members `code`, `err`, `func`, `file`, and `line`, which are (respectively) a numerical error code, a string indicating the nature of the error that generated the exception, the name of the function in which the error occurred, the file in which the error occurred, and an integer indicating the line on which the error occurred in that file. `err`, `func`, and `file` are all `STL` strings.

There are several built-in macros for generating exceptions yourself. `CV_Error( errorcode, description )` will generate and throw an exception with a fixed text description. `CV_Error_( errorcode, printf_fmt_str, [printf-args] )` works the same, but allows you to replace the fixed description with a printf-like format string and arguments. Finally, there is `CV_Assert( condition )` and `CV_DbgAssert( condition )`. Both will test your condition and throw an exception if the condition is not met. The latter version, however, will only operate in debug builds. These macros are the strongly preferred method of throwing exceptions, as they will automatically take care of the fields `func`, `file`, and `line` for you.

The `cv::DataType<>` Template

When OpenCV library functions need to communicate the concept of a particular data type, they do so by creating an object of type `cv::DataType<>`. `cv::DataType<>` itself is a template, and so the actual objects passed around are specializations of this template. This is an example of what in C++ are generally called *traits*. This allows the `cv::DataType<>` object to both contain runtime information about the

[10] In this case, by `FILE` we mean `struct FILE`, as defined in the C standard library.

type, as well as to contain `typedef` statements in its own definition that allow it to refer to the same type at compile time.

This might sound a bit confusing, and it is, but that is an inevitable consequence of trying to mix runtime information and compile-time information in C++.[11] An example will help clarify.

The template class definition for `DataType` is the following:

```
template<typename _Tp> class DataType
{
  typedef _Tp        value_type;
  typedef value_type work_type;
  typedef value_type channel_type;
  typedef value_type vec_type;

  enum {
    generic_type = 1,
    depth        = -1,
    channels     = 1,
    fmt          = 0,
    type         = CV_MAKETYPE(depth, channels)
  };
};
```

Let's try to understand what this means, and then follow it with an example. First, we can see that `cv::DataType<>` is a template, and expects to be specialized to a class called `_Tp`. It then has four `typedef` statements that allow the type of the `cv::DataType<>`, as well as some other related types, to be extracted from the `cv::DataType<>` instantiated object at compile time. In the template definition, these are all the same, but we will see in our example of a specialization of the template that they do not have to be (and often should not be). The next section is an `enum` that contains several components.[12] These are the `generic_type`, the `depth`, the number of `channels`, the format `fmt`, and the `type`. To see what all of these things mean, we'll look at two example specializations of `cv::DataType<>`, from *core.hpp*. The first is the `cv::DataType<>` definition for `float`:

```
template<> class DataType<float>
{
public:
  typedef float      value_type;
  typedef value_type work_type;
  typedef value_type channel_type;
  typedef value_type vec_type;

  enum {
    generic_type = 0,
    depth        = DataDepth<channel_type>::value,
    channels     = 1,
    fmt          = DataDepth<channel_type>::fmt,
    type         = CV_MAKETYPE(depth, channels)
  };
};
```

The first thing to notice is that this is a definition for a C++ built-in type. It is useful to have such definitions for the built-in types, but we can also make them for more complicated objects. In this case, the `value_type` is of course `float`, and the `work_type`, `channel_type`, and `vec_type` are all the same. We will see more clearly what these are for in the next example. For the constants in the `enum`, this

---

[11] You don't have this sort of problem in languages (e.g., Python) that support variable introspection and have an intrinsic runtime concept of data types.

[12] If this construct is awkward to you, remember that you can always assign integer values to the "options" in an `enum` declaration. In effect, this is a way of stashing a bunch of integer constants that will be fixed at compile time.

example will do just fine. The first variable `generic_type` is set to 0, as it is zero for all types defined in *core.hpp*. The `depth` variable is the data type identifier used by OpenCV. For example, `cv::DataDepth<float>::value` resolves to the constant `cv::F32`. The entry `channels` is one because `float` is just a single number; we will see an alternative to this in the next example. The variable `fmt` gives a single character representation of the format. In this case, `cv::DataDepth<float>::fmt` resolves to `f`. The last entry is `type`, which is a representation similar to depth, but which includes the number of channels (in this case, one). `CV_MAKETYPE(cv::F32,1)` resolves to `cv::F32C1`.

The important thing about `DataType<>`, however, is to communicate the nature of more complicated constructs. This is essential, for example, for allowing algorithms to be implemented in a manner that is agnostic to the incoming data type (i.e., algorithms that use introspection to determine how to proceed with incoming data).

Consider the example of an instantiation of `cv::DataType<>` for a `cv::Rect<>` (itself containing an as yet unspecialized type `_Tp`):

```
template<typename _Tp> class DataType<Rect_<_Tp> >
{
public:
  typedef Rect_<_Tp>                                value_type;
  typedef Rect_<typename DataType<_Tp>::work_type> work_type;
  typedef _Tp                                       channel_type;
  typedef Vec<channel_type, channels>               vec_type;

  enum {
    generic_type = 0,
    depth        = DataDepth<channel_type>::value,
    channels     = 4,
    fmt          = ((channels-1)<<8) + DataDepth<channel_type>::fmt,
    type         = CV_MAKETYPE(depth, channels)
  };
};
```

This is a much more complicated example. First, notice that `cv::Rect` itself does not appear. You will recall that earlier we mentioned that `cv::Rect` was actually an alias for a template, and that template is called `cv::Rect_<>`. So this template could be specialized as `cv::DataType<Rect>` or, for example, `cv::DataType< Rect_<float> >`. For the case `cv::DataType<Rect>`, we recall that all of the elements are integers, so if we consider that case, all of the instantiations of the template parameter `_Tp` resolve to `int`. We can see that the `value_type` is just the compile-time name of the thing that the `cv::DataType<>` is describing (namely `Rect`). The `work_type`, however, is defined to be the `work_type` of `cv::DataType<int>` (which, not surprisingly is `int`). What we see is that the `work_type` is telling us what kind of variables the `cv::DataType<>` is made of (i.e., what we "do work" on). The channel type is also `int`. This means that if we want to represent this variable as a multichannel object, it should be represented as some number of `int` objects. Finally, just as `channel_type` tells us how to represent this `cv::DataType<>` as a multichannel object, `vec_type` tells us how to represent it as an object of type `cv::Vec<>`. The alias `cv::DataType<Rect>::vec_type` will resolve to `cv::Vec<int,4>`. Moving on to the runtime constants: `generic_type` is again 0, `depth` is `CV::S32`, `channels` is `4` (because there are actually four values, the same reason the `vec_type` instantiated to a `cv::Vec<>` of size `4`), `fmt` resolves to `0x3069` (since `i` is `0x69`), and `type` resolves to `cv::S32C4`.

class InputArray and class OutputArray

Many OpenCV functions take arrays as arguments and return arrays as return values, but in OpenCV, there are many kinds of arrays. We have already seen that OpenCV supports some small array types (`cv::Scalar`, `cv::Vec`, `cv::Matx`), and STL's `std::vector<>` in addition to the large array types in the next section (`cv::Mat` and `cv::SparseMat`). In order to keep the interface from becoming onerously complicated (and repetitive), OpenCV defines the types `cv::InputArray` and

`cv::OutputArray`. In effect, these types mean "any of the above" with respect to the many array forms supported by the library. There is even a `cv::InputOutputArray`, specifying an array for in place computation.

The primary difference between `cv::InputArray` and `cv::OutputArray` is that the former is assumed to be `const` (i.e., read only). You will typically see these two types used in the definitions of library routines. You will not tend to use them yourself, but when being introduced to library functions, their presence means that you can use any array type, including a single `cv::Scalar`, and the result should be what you expect.

Related to `cv::InputArray` is the special function `cv::noArray()` that returns `cv::InputArray`. The returned object can be passed to any input requiring `cv::InputArray` to indicate that this input is not being used.

Large Array Types

Finally, our journey brings us to the *large array types*. Chief among these is `cv::Mat`, which could be considered the epicenter of the entire C++ implementation of the OpenCV library. The overwhelming majority of functions in the OpenCV library are members of the `cv::Mat` class or take a `cv::Mat` as an argument or return `cv::Mat` as a return value.

> If you are familiar with the C interface (pre-version 2.1 implementation) of the OpenCV library, you will remember the data types `IplImage` and `CvMat`. You might also recall `CvArr`. In the C++ implementation, these are all gone, replaced with `cv::Mat`. This means no more dubious casting of `void*` pointers in function arguments, and in general a tremendous enhancement in the cleanliness of the library internally.

The `cv::Mat` class is used to represent *dense* arrays of any number of dimensions. In this context, dense means that for every entry in the array, there is a data value stored in memory corresponding to that entry, even if that entry is zero. Most images, for example, are stored as dense arrays. The alternative would be a *sparse* array. In the case of a sparse array, only nonzero entries are typically stored. This can result in a great savings of storage space if many of the entries are in fact zero, but can be very wasteful if the array is relatively dense. A common case for using a sparse array rather than a dense array would be a histogram. For many histograms, most of the entries are zero, and storing all those zeros is not necessary. For the case of sparse arrays, OpenCV has the alternative data structure `cv::SparseMat`.

class cv::Mat: N-Dimensional Dense Arrays

The class `cv::Mat` can be used for arrays of any number of dimensions. The data is stored in the array in what can be thought of as an *n*-dimensional analog of "raster scan order." This means that in a one-dimensional array, the elements are sequential. In a two-dimensional array, the data is organized into rows, and each row appears one after the other. For three-dimensional arrays, each plane is filled out row by row, and then the planes are packed one after the other.

Each matrix contains an element signaling the contents of the array called `flags`, an element that indicates the number of dimensions called `dims`, two elements that indicate the number of rows and columns called `rows` and `cols` (these are not valid for `dims>2`), a pointer to where the array data is stored called `data`, and a reference counter analogous to the reference counter used by `cv::Ptr<>` called `refcount`. This latter member allows `cv::Mat` to behave very much like a smart pointer for the data contained in `data`. The memory layout in `data` is described by the array `step[]`. The data array is laid out such that the address of an element whose indices are given by $\left(i_0, \dagger i_i, \dagger \ldots, i_{N_d-1}\right)$ is:

$$\&\left(mtx_{i_0,i_1,\ldots,\dagger i_{N_d-1}N_d}\right) = mtx.data + mtx.step\left[0\right] * i_0 + mtx.step\left[1\right] * i_1 + \ldots + mtx.step\left[N_d - 1\right] * i_{N_d-1}$$

In the simple case of a two-dimensional array, this reduces to:

$$\&\left(mtx_{i,j}\right) = mtx.data + mtx.step\left[0\right] * i + mtx.step\left[1\right] * j$$

The data itself contained in `cv::Mat` is not required to be simple primitives. Each element of the data in a `cv::Mat` can itself be either a single number, or multiple numbers. In the case of multiple numbers, this is what the library refers to as a *multichannel* array. In fact, an *n*-dimensional array and an (*n*-1)-dimensional multichannel array are actually very similar objects, but because of the frequency of occasions in which it is useful to be able to think of an array as a *vector valued* array, the library contains special provisions for such structures.[13]

One reason why this distinction is made is because of memory access. By definition, an *element* of an array is the part that may be vector-valued. For example, an array might be said to be a two-dimensional three-channel array of 32-bit floats; in this case, the element of the array is the three 32-bit floats with a size of 12 bytes. When laid out in memory, rows of an array may not be absolutely sequential; there may be small gaps that buffer each row before the next. The difference between an n-dimensional single-channel array and an $(n-1)$-dimensional multichannel array is that this padding will always occur at the end of full rows (i.e., the channels in an element will always be sequential).

Creating an Array

An array can be created simply by instantiating a variable of type `cv::Mat`. An array created in this manner has no size and no data type. It can, however, later be asked to allocate data by using a member function such as `create()`. One variation of `create()` takes as arguments a number of rows, a number of columns, and a type, and configures the array to represent a two-dimensional object. The type of an array determines what kind of elements it has. Valid types in this context specify both the fundamental type of element as well as the number of channels. All such types are defined in the library header, and have the form `cv::{U8,S16,U16,S32,F32,F64}C{1,2,3}`.[14] For example, `cv::F32C3` would imply a 32-bit floating-point three-channel array.

You can also specify these things when you first allocate the matrix if you prefer. There are many constructors for `cv::Mat`, one of which takes the same arguments as `create()` (and an optional fourth argument with which to initialize all of the elements in your new array).

```
cv::Mat m;
m.create( 3, 10, cv::F32C3 );        // 3 rows, 10 columns of 3-channel 32-bit floats
m.setTo( cv::Scalar( 1.0f, 0.0f, 1.0f ) );   // 1st channel is 1.0, 2nd 0.0, 3rd 1.0
```

is equivalent to

```
cv::Mat m( 3, 10, cv::F32C3, cv::Scalar( 1.0f, 0.0f, 1.0f ) );
```

It is important to understand that the data in an array is not *attached* rigidly to the array object. The `cv::Mat` object is really a header for a data area, which—in principle—is an entirely separate thing. For example, it is possible to assign one matrix `n` to another matrix `m` (i.e., `m=n`). In this case, the data pointer inside of `m` will be changed to point to the same data as `n`. The data pointed to previously by the data

---

[13] Pre-2.1 OpenCV array types had an explicit element `IplImage::nChannels`, which indicated the number of channels. Because of the more general way in which such concepts are captured in the `cv::Mat` object, this information is no longer directly stored in a class variable. Rather, it is returned by a member function `cv::channels()`.

[14] OpenCV allows for arrays with more than three channels, but to construct one of these, you will have to call one of the functions `cv::{U8,S16,U16,S32,F32,F64}C()`. These functions take a single argument, which is the number of channels. So `cv::U8C(3)` is equivalent to `cv::U8C3`, but since there is no macro for `cv::U8C7`, to get this you would have to call `cv::U8C(7)`.

element of `m` (if any) will be deallocated[15]. At the same time, the reference counter for the data area that they both now share will be incremented. Last, but not least, the members of `m` that characterize the data in `m` (such as `rows`, `cols`, and `flags`) will be updated to accurately describe the data now pointed to by `data` in `m`. This all results in a very convenient behavior, in which arrays can be assigned to one another, and the work necessary to do this is done automatically behind the scenes to give the correct result.

The following is a complete list of the constructors available for `cv::Mat`. The list appears rather unwieldy, but in fact you will only use a small fraction of these most of the time. Having said that, when you need one of the more obscure ones, you will probably find that you are glad it is there.

Table 3-10: `cv::Mat` *constructors that do not copy data*

Constructor	Description
`cv::Mat();`	Default constructor
`cv::Mat( int rows, int cols, int type );`	Two-dimensional arrays by type
`cv::Mat( int rows, int cols, int type, const Scalar& s );`	Two-dimensional arrays by type with initialization value
`cv::Mat( int rows, int cols, int type, void* data, size_t step=AUTO_STEP );`	Two-dimensional arrays by type with preexisting data
`cv::Mat( cv::Size sz, int type );`	Two-dimensional arrays by type (size in `sz`)
`cv::Mat( cv::Size sz, int type, const Scalar& s );`	Two-dimensional arrays by type with initializtion value (size in `sz`)
`cv::Mat( cv::Size sz, int type, void* data, size_t step=AUTO_STEP );`	Two-dimensional arrays by type with preexisting data (size in `sz`)
`cv::Mat( int ndims, const int* sizes, int type );`	Multidimensional arrays by type
`cv::Mat( int ndims, const int* sizes, int type, const Scalar& s );`	Multidimensional arrays by type with initialization value
`cv::Mat( int ndims, const int* sizes, int type, void* data, size_t step=AUTO_STEP );`	Multidimensional arrays by type with preexisting data

Table 3-10 lists the basic constructors for the `cv::Mat` object. Other than the default constructor, these fall into three basic categories: those that take a number of rows and a number of columns to create a two-dimensional array, those that use a `cv::Size` object to create a two-dimensional array, and those that construct n-dimensional arrays and require you to specify the number of dimensions and pass in an array of integers specifying the size of each of the dimensions.

[15] Technically, it will only be deallocated if `m` was the last `cv::Mat` around which pointed to that particular data.

In addition, some of these allow you to initialize the data, either by providing a `cv::Scalar` (in which case, the entire array will be initialized to that value), or by providing a pointer to an appropriate data block that can be used by the array. In this latter case, you are essentially just creating a header to the existing data (i.e., no data is copied; the data member is set to point to the data indicated by the `data` argument).

Table 3-11: `cv::Mat` *constructors that copy data from other* `cv::Mat`*'s*

Constructor	Description
`cv::Mat( const Mat& mat );`	Copy constructor
`cv::Mat( const Mat& mat, const cv::Range& rows, const cv::Range& cols );`	Copy constructor that copies only a subset of rows and columns
`cv::Mat( const Mat& mat, const cv::Rect& roi );`	Copy constructor that copies only a subset of rows and columns specified by a region of interest
`cv::Mat( const Mat& mat, const cv::Range* ranges );`	Generalized region of interest copy constructor that uses an array of ranges to select from an *n*-dimensional array
`cv::Mat( const cv::MatExpr& expr );`	Copy constructor that initializes `m` with the result of an algebraic expression of other matrices

The copy constructors (Table 3-11) show how to create an array from another array. In addition to the basic copy constructor, there are three methods for constructing an array from a sub-region of an existing array and one constructor that initializes the new matrix using the result of some matrix expression.

The sub-region (also known as "region of interest") constructors come in three flavors: one that takes a range of rows and a range of columns (this works only on a two-dimensional matrix), one that uses a `cv::Rect` to specify a rectangular sub-region (also works only on a two-dimensional matrix), and a final one that takes an array of ranges. In this latter case, the number of valid ranges pointed to by the pointer argument `ranges` is required to be equal to the number of dimensions of the array `mat`. It is this third option that you must use if `mat` is a multidimensional array with `ndim` greater than `2`.

Table 3-12: `cv::Mat` *constructors for pre-version 2.1 data types*

Constructor	Description
`cv::Mat( const CvMat* old, bool copyData=false );`	Constructor for `m` that creates `m` from an old-style `CvMat`, with optional data copy
`cv::Mat( const IplImage* old, bool copyData=false );`	Constructor for `m` that creates `m` from an old-style `IplImage`, with optional data copy

If you are modernizing or maintaining pre-version 2.1 code that still contains the C-style data structures, you may want to create a new C++-style `cv::Mat` structure from an existing `CvMat` or `IplImage` structure. In this case, you have two options. You can construct a header on the existing data (by setting `copyData` to false) or you can set `copyData` to true (in which case, new memory will be allocated for `m` and all of the data from `old` will be copied into `m`).

These objects do a lot more for you than you might realize at first. In particular, they allow for expressions that mix the C++ and C data-types by functioning as implicit constructors for the C++ data types on demand. Thus, it is possible to simply use a pointer to one of the C structures wherever a `cv::Mat` is expected and have a reasonable expectation that things will work out correctly. (This is why the `copyData` member defaults to `false`.)

In addition to these constructors, there are corresponding cast operators that will convert a `cv::Mat` into `CvMat` or `IplImage` on demand. These also do not copy data.

Table 3-13: `cv::Mat` *template constructors*

Constructor	Description
`cv::Mat( const cv::Vec<T,n>& vec, bool copyData=true );`	Construct a one-dimensional array of type `T` and size `n` from a `cv::Vec` of the same type
`cv::Mat( const cv::Matx<T,m,n>& vec, bool copyData=true );`	Construct a two-dimensional array of type `T` and size `m`-by-`n` from a `cv::Matx` of the same type
`cv::Mat( const std::vector<T>& vec, bool copyData=true );`	Construct an one-dimensional array of type `T` from an STL `vector` containing elements of the same type

The last set of constructors is the template constructors. These constructors are template not because they construct a template form of `cv::Mat`, but because they construct an instance of `cv::Mat` from something that is itself template. These constructors allow either an arbitrary `cv::Vec<>` or `cv::Matx<>` to be used to create a `cv::Mat` array of corresponding dimension and type, or to use an STL `vector<>` object of arbitrary type to construct an array of that same type.

Table 3-14: `static` *functions that create* `cv::Mat`

Function	Description
`cv::Mat::zeros( rows, cols, type );`	Create a `cv::Mat` of size `rows`-by-`cols`, which is full of zeros, with type `type` (`cv::F32`, etc.)
`cv::Mat::ones( rows, cols, type );`	Create a `cv::Mat` of size `rows`-by-`cols`, which is full of ones, with type `type` (`cv::F32`, etc.)
`cv::Mat::eye( rows, cols, type );`	Create a `cv::Mat` of size `rows`-by-`cols`, which is an identity matrix, with type `type` (`cv::F32`, etc.)

The class `cv::Mat` also provides a number of static member functions to create certain kinds of commonly used arrays. These include functions like `zeros()`, `ones()`, and `eye()`, which construct a matrix full of zeros, a matrix full of ones, or an identity matrix, respectively.[16]

Accessing Array Elements Individually

There are several ways to access a matrix, which are designed to be convenient in different contexts. In recent versions of OpenCV, however, a great deal of effort has been invested to make them all comparably, if not identically, efficient. The two primary options for accessing individual elements are to access them by location, or to access them through iteration.

The basic means of direct access is the (template) member function `at<>()`[17]. There are many variations of this function that take different arguments for arrays of different numbers of dimensions. The way this function works is that you specialize the `at<>()` template to the type of data that the matrix contains, then access that element using the row and column locations of the data you want. Here is a simple example:

```
cv::Mat m = cv::Mat::eye( 10, 10, 32FC1 );
printf(
  "Element (3,3) is %f\n",
  m.at<float>(3,3) //(row, col)
);
```

For a multichannel array, the analogous example would look like this:

```
cv::Mat m = cv::Mat::eye( 10, 10, 32FC2 );
printf(
  "Element (3,3) is (%f,%f)\n",
  m.at<cv::Vec2f>(3,3)[0],
  m.at<cv::Vec2f>(3,3)[1]
);
```

Note that when you want to specify a template function like `at<>()` to operate on a multichannel array, the best way to do this is to use a `cv::Vec<>` object (either a premade alias or the template form).

Similar to the vector case, you can create an array made of a more sophisticated type, such as complex numbers:

```
cv::Mat m = cv::Mat::eye( 10, 10, cv::DataType<cv::Complexf>::type );
printf(
  "Element (3,3) is %f + i%f\n",
  m.at<cv::Complexf>(3,3).re,
  m.at<cv::Complexf>(3,3).im,
);
```

It is also worth noting the use of the `cv::DataType<>` template here. The matrix constructor requires a runtime value that is a variable of type `int` that happens to take on some "magic" values that the constructor understands. By contrast, `cv::Complexf` is an actual object type, a purely compile-time construct. The need to generate one of these representations (runtime) from the other (compile time) is precisely why the `cv::DataType<>` template exists. *Table 3-15* lists the available variations of the `at<>()` template.

Table 3-15: Variations of the `at<>()` *accessor function*

Example	Description
`M.at<int>( i );`	Element *i* from integer array *M*

[16] In the case of `cv::Mat::eye()` and `cv::Mat::ones()`, if the array created is multi-channel, only the first channel will be set `1.0` while the other channels will be `0.0`.

[17] For two channels, the order is (row, col), `at<>(row, col)`.

`M.at<float>( i, j );`	Element *`( i, j )`* (row, col) from float array *`M`*
`M.at<int>( pt );`	Element at location *`(pt.x, pt.y)`* in integer matrix *`M`*
`M.at<float>( i, j, k );`	Element at location *`( i, j, k )`* in three-dimensional float array *`M`*
`M.at<uchar>( idx );`	Element at *`n`*-dimensional location indicated by *`idx[]`* in array *`M`* of unsigned characters

To access a two-dimensional array, you can also extract a C-style pointer to a specific row of the array. This is done with the `ptr<>()` template member function of `cv::Mat`. (Recall that the data in the array is contiguous by row, thus accessing a specific column in this way would not make sense; we will see the right way to do that shortly.) As with `at<>()`, `ptr<>()` is a template function instantiated with a type name. It takes an integer argument indicating the row you wish a pointer to. The function returns a pointer to the primitive type of which the array is constructed (i.e., if the array type is `F32C3`, the return value will be of type `float*`). Thus, given a three-channel matrix `mtx` of type `float`, the construction `mtx.ptr<Vec3f>(3)` would return a pointer to the first (floating-point) channel of the first element in row `3` of `mtx`. This is generally the fastest way to access elements of an array,[18] because once you have the pointer, you are right down there with the data.

> There are thus two ways to get a pointer to the data in a matrix `mtx`. One way is to go about it with the `ptr<>()` member function. The other is to directly use the member pointer `data`, and to use the member array `step[]` to compute addresses. The latter option is similar to what one tended to do in the C interface, but is generally no longer preferred over access methods such as `at<>()`, `ptr<>()`, and the iterators. Having said this, direct address computation may still be most efficient, particularly when dealing with arrays of dimension greater than two.

There is one last important point to keep in mind about C-style pointer access. If you want to access everything in an array, you will likely iterate one row at a time, because the rows may or may not be packed continuously in the array. However, there is a member function `isContinuous()` that will tell you if the members are continuously packed. If they are, you can just grab the pointer to the very first element of the first row and cruise through the entire array as it if were a giant one-dimensional array.

The other form of sequential access is to use the iterator mechanism built into `cv::Mat`. This mechanism is based on, and works more or less identically to, the analogous mechanism provided by the STL containers. The basic idea is that OpenCV provides a pair of iterator templates, one for `const` and one for non-`const` arrays. These iterators are named `cv::MatIterator<>` and `cv::MatConstIterator<>`, respectively. The `cv::Mat` methods `begin()` and `end()` return objects of this type. This method of iteration is convenient because the iterators are smart enough to handle the continuous packing as well as the non-continuous packing cases automatically, as well as handling any number of dimensions in the array.

[18] The difference in performance between using `at<>()` and direct pointer access depends on compiler optimization. Access through `at<>()` will tend to be comparable to (though slightly slower than) direct pointer access in code with a good optimizer, but may be more than order of magnitude slower if that optimizer is turned off (e.g., when you do a debug build). Access through iterators is almost always slower than either of these. In almost all cases, however, using built-in OpenCV functions will be faster than any loop you write regardless of the direct access methods described here, so avoid that kind of construct wherever possible.

Each iterator must be declared and specified to the type of object from which the array is constructed. Here is a simple example of the iterators being used to compute the "longest" element in a three-dimensional array of three-channel elements (a three-dimensional vector field):

```
int sz[3] = { 4, 4, 4 };
cv::Mat m( 3, sz, cv::F32C3 );  // A three-dimensional array of size 4-by-4-by-4
cv::randu( m, -1.0f, 1.0f );   // fill with random numbers from -1.0 to 1.0

float max = 0.0f;              // minimum possible value of L2 norm
cv::MatConstIterator<cv::Vec3f> it = m.begin();
while( it != m.end() ) {
  len2 = (*it)[0]*(*it)[0]+(*it)[1]*(*it)[1]+(*it)[2]*(*it)[2];
  if( len2 > max ) max = len2;
  it++;
}
```

Iterator-based access is typically used when doing operations over an entire array, or element-wise across multiple arrays. Consider the case of adding two arrays, or of converting an array from the RGB color space to the HSV color space. In such cases, the same exact operation will be done at every pixel location.

The *N*-ary Array Iterator: `NAryMatIterator`

There is another form of iteration which, though it does not handle discontinuities in the packing of the arrays in the manner of `cv::MatIterator<>`, allows us to handle iteration over many arrays at once. This iterator is called `cv::NAryMatIterator`, and requires only that all of the arrays that are being iterated over be of the same geometry (number of dimensions and extent in each dimension).

Instead of returning single elements of the arrays being iterated over, the N-ary iterator operates by returning chunks of those arrays, called *planes*. A plane is a portion (typically one- or two-dimensional slice) of the input array in which the data is guaranteed to be contiguous in memory.[19] This is how discontinuity is handled; you are given the contiguous chunks one by one. For each such plane, you can either operate on it using array operations, or iterate trivially over it yourself. (In this case, "trivially" means to iterate over it in a way which does not need to check for discontinuities inside of the chunk.)

The concept of the plane is entirely separate from the concept of multiple arrays being iterated over simultaneously. Consider the following code example, in which we will sum just a single multidimensional array plane by plane:

```
  const int n_mat_size = 5;
  const int n_mat_sz[] = { n_mat_size, n_mat_size, n_mat_size };
  cv::Mat n_mat( 3, n_mat_sz, cv::F32C1 );

  cv::RNG rng;
  rng.fill( n_mat, cv::RNG::UNIFORM, 0.f, 1.f );

  const cv::Mat* arrays[] = { &n_mat, 0 };
  cv::Mat my_planes[1];
  cv::NAryMatIterator it( arrays, my_planes );
```

At this point, you have your N-ary iterator. Continuing our example, we will compute the sum of `m0` and `m1`, and place the result in `m2`. We will do this plane by plane, however:

```
// On each iteration, it.planes[i] will be the current plane of the
// i-th array from 'arrays'.
//

  float s = 0.f;                                // Total sum over all planes
  int   n = 0;                                  // Total number of planes
```

[19] In fact the dimensionality of the "plane" is not limited to 2; it can be larger. What is always the case is that the planes will be contiguous in memory.

```
for (int p = 0; p < it.nplanes; p++, ++it) {
  s += cv::sum(it.planes[0])[0];
  n++;
}
```

In this example, we first create the three-dimensional array `n_mat` and fill it with 125 random floating point numbers between `0.0` and `1.0`. To initialize the `cv::NAryMatIterator` object, we need to have two things. First, we need a C-style array containing pointers to all of the `cv::Mat`'s we wish to iterate over (in this example, there is just one). This array must always be terminated with a `0` or `NULL`. Next, we need another C-style array of `cv::Mat`'s that can be used to refer to the individual planes as we iterate over them (in this case, there is also just one).

Once we have created the *N*-ary iterator, we can iterate over it. Recall that this iteration is over the planes that make up the arrays we gave to the iterator. The number of planes (the same for each array, because they have the same geometry) will always be given by `it.nplanes`. The *N*-ary iterator contains a C-style array called `planes` that holds headers for the current plane in each input array. In our example, there is only one array being iterated over, so we need only refer to `it.planes[0]` (the current plane in the one and only array). In this example, we then call `cv::sum()` on each plane and accumulate the final result.

To see the real utility of the N-ary iterator, consider a slightly expanded version of this example in which there are two arrays we would like to sum over:

```
const int n_mat_size = 5;
const int n_mat_sz[] = { n_mat_size, n_mat_size, n_mat_size };
cv::Mat n_mat0( 3, n_mat_sz, cv::F32C1 );
cv::Mat n_mat1( 3, n_mat_sz, cv::F32C1 );

cv::RNG rng;
rng.fill( n_mat0, cv::RNG::UNIFORM, 0.f, 1.f );
rng.fill( n_mat1, cv::RNG::UNIFORM, 0.f, 1.f );

const cv::Mat* arrays[] = { &n_mat0, &n_mat1, 0 };
cv::Mat my_planes[2];
cv::NAryMatIterator it( arrays, my_planes );

float s = 0.f;                    // Total sum over all planes in both arrays
int   n = 0;                      // Total number of planes
for(int p = 0; p < it.nplanes; p++, ++it) {
  s += cv::sum(it.planes[0])[0];
  s += cv::sum(it.planes[1])[0];
  n++;
}
```

In this second example, you can see that the C-style array called `arrays` is given pointers to both input arrays, and two matrices are supplied in the `my_planes` array. When it is time to iterate over the planes, at each step, `planes[0]` contains a plane in `n_mat0` and `planes[1]` contains the corresponding plane in `n_mat1`. In this simple example, we just sum the two planes and add them to our accumulator. In an only slightly extended case, we could use element-wise addition to sum these two planes and place the result into the corresponding plane in a third array.

Not shown in the example, but also important, is the member `it.size`, which indicates the size of each plane. The size reported is the number of elements in the plane, so it does not include a factor for the number of channels. In our previous example, if `it.nplanes` was four, then `it.size` would have been sixteen:

```
////////////// compute dst[*] = pow(src1[*], src2[*]) //////////////
const Mat* arrays[] = { src1, src2, dst, 0 };
float* ptrs[3];

NAryMatIterator it(arrays, (uchar**)ptrs);
for( size_t i = 0; i < it.nplanes; i++, ++it )
```

```
    {
        for( size_t j = 0; j < it.size; j++ )
        {
            ptrs[2][j] = std::pow(ptrs[0][j], ptrs[1][j]);
        }
}
```

Accessing Array Elements by Block

In the previous section, we saw ways to access individual elements of an array, either singularly or by iterating sequentially through them all. Another common situation that arises is when you need to access a subset of an array as another array. This might be to select out a row or a column, or any sub-region of the original array.

There are many methods that do this for us in one way or another; all of them are member functions of the `cv::Mat` class and return a subsection of the array on which they are called. The simplest of these methods are `row()` and `col()`, which take a single integer and return the indicated row or column of the array whose member we are calling. Clearly these make sense only for a two-dimensional array; we will get to the more complicated case momentarily.

When you use `m.row()` or `m.col()` (for some array `m`), or any of the other functions we are about to discuss, it is important to understand that the data in `m` is not copied to the new arrays. Consider an expression like `m2 = m.row(3)`. This expression means to create a new array header `m2`, and to arrange its `data` pointer, `step` array, and so on, such that it will access the data in row `3` in `m`. If you modify the data in `m2`, you will be modifying the data in `m`. Later, we will visit the `copyTo()` method, which actually will copy data. The main advantage of the way this is handled in OpenCV is that the amount of time required to create a new array that accesses part of an existing array is not only very small, but also independent of the size of either the old or the new array.

Closely related to `row()` and `col()` are `rowRange()` and `colRange()`. These functions do essentially the same thing as their simpler cousins, except that they will extract an array with multiple contiguous rows (or columns). Both functions can be called in one of two ways, either by specifying an integer start and end row (or column), or by passing a `cv::Range` object that indicates the desired rows (or columns). In the case of the two-integer method, the range is inclusive of the start index but exclusive of the end index (you may recall that `cv::Range` uses a similar convention).

The member function `diag()` works the same as `row()` or `col()`, except that the array returned from `m.diag()` references the diagonal elements of a matrix. `m.diag()` expects an integer argument that indicates which diagonal is to be extracted. If that argument is zero, then it will be the main diagonal. If it is positive, it will be offset from the main diagonal by that distance in the upper half of the array. If it is negative, then it will be from the lower half of the array.

The last way to extract a sub-matrix is with `operator()`. Using this operator, you can pass either a pair of ranges (a `cv::Range` for rows and a `cv::Range` for columns) or a `cv::Rect` to specify the region you want. This is the only method of access that will allow you to extract a sub-volume from a higher-dimensional array. In this case, a pointer to a C-style array of ranges is expected and that array must have as many elements as the number of dimensions of the array.

Table 3-16: Block access methods of `cv::Mat`

Example	Description
`m.row( i );`	Array corresponding to row i of m
`m.col( j );`	Array corresponding to column j of m
`m.rowRange( i0, i1 );`	Array corresponding to rows i0 through i1-1 of matrix m

`m.rowRange( cv::Range( i0, i1 ) );`	Array corresponding to rows i0 through i1-1 of matrix m
`m.colRange( j0, j1 );`	Array corresponding to columns j0 through j1-1 of matrix m
`m.colRange( cv::Range( j0, j1 ) );`	Array corresponding to columns j0 through j1-1 of matrix m
`m.diag( d );`	Array corresponding to the d-offset diagonal of matrix m
`m( cv::Range(i0,i1), cv::Range(j0,j1) );`	Array corresponding to the subrectangle of matrix m with one corner at i0, j0 and the opposite corner at (i1-1, j1-1)
`m( cv::Rect(i0,i1,w,h) );`	Array corresponding to the subrectangle of matrix m with one corner at i0, j0 and the opposite corner at (i0+w-1, j0+h-1)
`m( ranges );`	Array extracted from m corresponding to the subvolume that is the intersection of the ranges given by ranges[0]-ranges[ndim-1]

Matrix Expressions: Algebra and `cv::Mat`

One of the things enabled by the move to C++ in version 2.1 is the overloading of operators and the ability to create algebraic expressions consisting of matrix arrays[20] and singletons. The primary advantage of this is code clarity, as many operations can be combined into one expression that is both more compact and often more meaningful.

In the background, many important features of OpenCV's array class are being used to make these operations work. For example, matrix headers are created automatically as needed and workspace data areas are allocated (only) as required. When no longer needed, data areas are deallocated invisibly and automatically. The result of the computation is finally placed in the destination array by `operator=()`. However, one important distinction, is that this form of `operator=()` is not assigning a `cv::Mat` or a `cv::Mat` (as it might appear), but rather a `cv::MatExpr` (the expression itself[21]) to a `cv::Mat`. This distinction is important because data is always copied into the result (left-hand side) array. Recall that though `m2=m1` is legal, it means something slightly different. In this first case, `m2` would be another reference to the data in `m1`. By contrast, `m2=m1+m0` means something different again. Because `m1+m0` is a

[20] For clarity, we use the word "array" when referring to a general object of type `cv::Mat`, and use the word "matrix" for those situations in which the manner in which the array is being used indicates that it is representing a mathematical object that would be called a matrix. The distinction is a purely semantic one, and not manifest in the actual design of OpenCV.

[21] The underlying machinery of `cv::MatExpr` is more detail than we need, but you can think of `cv::MatExpr` as being a symbolic representation of the algebraic form of the right-hand side. The great advantage of `cv::MatExpr` is that when it is time to evaluate an expression, it is often clear that some operations can be removed or simplified without evaluation (such as computing the transpose of the transpose of a matrix, adding zero, or multiplying a matrix by its own inverse).

matrix expression, it will be evaluated and a pointer to the results will be assigned in `m2`. The results will reside in a newly allocated data area.[22]

Table 3-17 lists examples of the algebraic operations available. Note that in addition to simple algebra, there are comparison operators, operators for constructing matrices (such as `cv::Mat::eye()`, which we encountered earlier), and higher level operations for computing transposes and inversions. The key idea here is that you should be able to take the sorts of relatively nontrivial matrix expressions that occur when doing computer vision and express them on a single line in a clear and concise way.

Table 3-17: Operations available for matrix expressions

Example	Description
`m0 + m1, m0 - m1;`	Addition or subtraction of matrices
`m0 + s; m0 - s; s + m0, s - m1;`	Addition or subtraction between a matrix and a singleton
`-m0;`	Negation of a matrix
`s * m0; m0 * s;`	Scaling of a matrix by a singleton
`m0.mul( m1 ); m0/m1;`	Per element multiplication of m0 and m1, per-element division of m0 by m1
`m0 * m1;`	Matrix multiplication of m0 and m1
`m0.inv( method );`	Matrix inversion of m0 (default value of `method` is `DECOMP_LU`)
`m0.t();`	Matrix transpose of m0 (no copy is done)
`m0>m1; m0>=m1; m0==m1; m0<=m1; m0<m1;`	Per element comparison, returns `uchar` matrix with elements `0` or `255`
`m0&m1; m0|m1; m0^m1; ~m0;` `m0&s; s&m0; m0|s; s|m0; m0^s; s^m0;`	Bitwise logical operators between matrices or matrix and a singleton
`min(m0,m1); max(m0,m1); min(m0,s);` `min(s,m0); max(m0,s); max(s,m0);`	Per element minimum and maximum between two matrices or a matrix and a singleton
`cv::abs( m0 );`	Per element absolute value of `m0`
`m0.cross( m1 ); m0.dot( m1 );`	Vector cross and dot product (vector cross product is only defined for 3-by-1 matrices)
`cv::Mat::eye( Nr, Nc, type );`	Class static matrix initializers that return

[22] If you are a real expert, this will not surprise you. Clearly a temporary array must be created to store the result of `m1+m0`. Then `m2` really is just another reference, but it is another reference to that temporary array. When `operator+()` exits, its reference to the temporary array is discarded, but the reference count is not zero. `m2` is left holding the one and only reference to that array.

`cv::Mat::zeros( Nr, Nc, type );`	fixed N_r-by-N_c matrices of type `type`
`cv::Mat::ones( Nr, Nc, type );`	

The matrix inversion operator `inv()` is actually a frontend to a variety of algorithms for matrix inversion. There are currently three options. The first option is `cv::DECOMP_LU`, which means LU decomposition, and which works for any nonsingular matrix. The second is `cv::DECOMP_CHOLESKY`, which solves the inversion by Cholesky decomposition. Cholesky decomposition only works for symmetric, positive definite matrices, but is much faster than LU decomposition for large matrices. The last option is `cv::DECOMP_SVD`, which solves the inversion by singular value decomposition. SVD is the only workable option for matrices that are singular or not even squares (the pseudo-inverse is then computed).

Not included in *Table 3-17* are all of the functions like `cv::norm()`, `cv::mean()`, `cv::sum()`, and so on (some of which we have not gotten to yet, but you can probably guess what they do) that convert matrices to other matrices or to scalars. Any such object can still be used in a matrix expression.

Saturation Casting

In OpenCV, you will often do operations that risk overflowing or underflowing the available values in the destination of some computation. This is particularly common when one is doing operations on unsigned types that involve subtraction, but it can happen anywhere. To deal with this problem, OpenCV relies on a construct called *saturation casting*.

What this means is that OpenCV arithmetic and other operations that act on arrays will check for underflows and overflows automatically; in these cases, the library functions will replace the resulting value of an operation with the lowest or highest available value, respectively. Note that this is not what C language operations normally and natively do.

You may want to implement this particular behavior in your own functions as well. OpenCV provides some handy templated casting operators to make this easy for you. These are implemented as a template function called `cv::saturate_cast<>()`, which allows you to specify the type to which you would like to cast the argument. Here is an example:

```
uchar& Vxy = m0.at<uchar>( y, x );
Vxy = cv::saturate_cast<uchar>((Vxy-128)*2 + 128);}
```

In this example code, the variable `Vxy` is first assigned to be a reference to an element of an 8-bit array `m0`. This array then has 128 subtracted from it, multiply that by two (scale that up), and add 128 (so the result is twice as far from `128` as the original). The usual C arithmetic rules would assign `Vxy-128` to a (32-bit) signed integer; followed by integer multiplication by `2` and integer addition of `128`. Notice, however, that if the original value of `Vxy` were (for example) `10`, then `Vxy-128` would be `-118`. The value of the expression would then be `-108`. This number will not fit into the 8-bit unsigned variable `Vxy`. This is where `cv::saturation_cast<uchar>()` comes to the rescue. It takes the value of `-108` and, recognizing that it is too low for an `unsigned char`, converts it to `0`.

More Things an Array Can Do

At this point, we have touched on most of the members of the `cv::Mat` class. Of course, there are a few things that were missed, as they did not fall into any specific category that was discussed so far. In this section we will review the leftovers that you will need in your daily life of OpenCV programming.

Table 3-18: More class member functions of `cv::Mat`

Example	Description
`m1 = m0.clone();`	Make a complete copy of `m0`, copying all data elements as well; cloned array will be continuous
`m0.copyTo( m1 );`	Copy contents of `m0` onto `m1`, reallocating `m1` if necessary (equivalent to `m1=m0.clone()`)

`m0.copyTo( m1, mask );`	As `m0.copyTo(m1)` except only entries indicated in the array `mask` are copied
`m0.convertTo( m1, type, scale, offset );`	Convert elements of `m0` to `type` (i.eg., `cv::F32`) and write to `m1` after scaling by `scale` (default 1.0) and adding `offset` (default 0.0)
`m0.assignTo( m1, type );`	*internal use only* (resembles convertTo)
`m0.setTo( s, mask );`	Set all entries in `m0` to singleton value `s`; if `mask` is present, only set those value corresponding to nonzero elements in `mask`
`m0.reshape( chan, rows );`	Changes effective shape of a two-dimensional matrix; `chan` or `rows` may be zero, which implies "no change"; data is not copied
`m0.push_back( s );`	Extend a *M*-by-1 matrix and insert the singleton `s` at the end
`m0.push_back( m1 );`	Extend an *M*-by-*N* by K rows and copy `m1` into that rows; `m1` must be *K*-by-*N*
`m0.pop_back( n );`	Remove `n` rows from the end of an M-by-N (default value of `n` is one)[23]
`m0.locateROI( size, offset );`	Write whole size of `m0` to `cv::Size size`; if `m0` is a "view" of a larger matrix, write location of starting corner to `Point& offset`
`m0.adjustROI( t, b, l, r );`	Increase the size of a view by `t` pixels above, `b` pixels below, `l` pixels to the left, and `r` pixels to the right
`m0.total();`	Compute the total number of array elements (does not include channels)
`m0.isContinuous();`	Return `true` only if the rows in `m0` are packed without space between them in memory
`m0.elemSize();`	Return the size of the elements of `m0` in bytes (eE.g., a three-channel float matrix would return 12 bytes)
`m0.elemSize1();`	Return the size of the subelements of `m0` in bytes (eE.g., a three-channel float matrix would return 4 bytes)
`m0.type();`	Return a valid type identifier for the elements of `m0` (e.g., `cv::F32C3`)
`m0.depth();`	Return a valid type identifier for the individial channels of

[23] Many implementations of "pop" functionality return the popped element. This one does not; its return type is void.

	`m0` (e.g., `cv::F32`)
`m0.channels();`	Return the number of channels in the elements of `m0`.
`m0.size();`	Return the size of the `m0` as a `cv::Size` object.
`m0.empty();`	Return `true` only if the array has no elements (i.e., `m0.total==0` or `m0.data==NULL`)

`class cv::SparseMat`: Sparse Arrays

The class `cv::SparseMat` is used when an array is likely to be very large compared to the number of nonzero entries. This situation often arises in linear algebra with sparse matrices, but it also comes up when one wishes to represent data, particularly histograms, in higher-dimensional arrays, since most of space will be empty. A sparse representation only stores data that is actually present and so can save a great deal of memory. In practice, many sparse objects would be too huge to represent at all in a dense format. The disadvantage of sparse representations is that computation with them is slower (on a per-element basis). This last point is important, in that computation with sparse matrices is not categorically slower, as there can be a great economy in knowing in advance that many operations need not be done at all.

The OpenCV sparse matrix class `cv::SparseMat` functions analogously to the dense matrix class `cv::Mat` in most ways. It is defined similarly, supports most of the same operations, and can contain the same data types. Internally, the way data is organized is quite different. While `cv::Mat` uses a data array closely related to a C data array (one in which the data is sequentially packed and addresses are directly computable from the indices of the element), `cv::SparseMat` uses a hash table to store just the nonzero elements.[24] That hash table is maintained automatically, so when the number of (nonzero) elements in the array becomes too large for efficient look-up, the table grows automatically.

Accessing Sparse Array Elements

The most important difference between sparse and dense arrays is how elements are accessed. Sparse arrays provide four different access mechanisms: `cv::SparseMat::ptr()`, `cv::SparseMat::ref()`, `cv::SparseMat::value()`, and `cv::SparseMat::find()`.

The `cv::SparseMat::ptr()` method has several variations, the simplest of which has the template:

```
uchar* cv::SparseMat::ptr( int i0, bool createMissing, size_t* hashval=0 )
```

This particular version is for accessing a one-dimensional array. The first argument `i0` is the index of the requested element. The next argument `createMissing` indicates whether the element should be created if it is not already present in the array. When `cv::SparseMat::ptr()` is called, it will return a pointer to the element if that element is already defined in the array, but `NULL` if that element is not defined. However, if the `createMissing` argument is `true`, however, that element will be created and a valid non-`NULL` pointer will be returned to that new element. To understand the final argument `hashval`, it is necessary to recall that the underlying data representation of a `cv::SparseMat` is as a hash table. Looking up objects in a hash table requires two steps: the first being the computation of the hash key (in this case, from the indices), and the second being the searching of a list associated with that key. Normally, that list will be short (ideally only one element), so the primary computational cost in a lookup is the computation of the hash key. If this key has already been computed (as with `cv::SparseMat::hash()`, which will be covered in the next section), then time can be saved by not recomputing it. In the case of `cv::SparseMat::ptr()`, if the argument `hashval` is left with its default argument of `NULL`, the hash key will be computed. If, however, a key is provided, it will be used.

---

[24] Actually, zero elements may be stored, if those elements have become zero as a result of computation on the array. If you want to clean up such elements, you must do so yourself. This is the function of the method `SparseMat::erase()`, which we will visit shortly.

There are also variations of `cv::SparseMat::ptr()` that allow for two or three indices, as well as a version whose first argument is a pointer to an array of integers (i.e., `const int* idx`), which is required to have the same number of entries as the dimension of the array being accessed.

In all cases, the function `cv::SparseMat::ptr()` returns a pointer to an unsigned character (i.e., `uchar*`), which will typically need to be recast to the correct type for the array.

The accessor template function `SparseMat::ref<>()` is used to return a reference to a particular element of the array. This function, like `SparseMat::ptr()`, can take one, two, or three indices, or a pointer to an array of indices, and also supports an optional pointer to the hash value to use in the lookup. Because it is a template function, you must specify the type of object being referenced. So, for example, if your array were of type `cv::F32`, then you might call `SparseMat::ref<>()` like this:

```
a_sparse_mat.ref<float>( i0, i1 ) += 1.0f;
```

The template method `cv::SparseMat::value<>()` is identical to `SparseMat::ref<>()`, except that it returns the value and not a reference to the value. As such, this method is itself a "`const` method."[25]

The final accessor function is `cv::SparseMat::find<>()`, which works similarly to `cv::SparseMat::ref<>()` and `cv::SparseMat::value<>()`, but returns a pointer to the requested object. Unlike `cv::SparseMat::ptr()`, however, this pointer is of the type specified by the template instantiation of `cv::SparseMat::find<>()`, and so does not need to be recast. For purposes of code clarity, `cv::SparseMat::find<>()` is preferred over `cv::SparseMat::ptr()` wherever possible. `cv::SparseMat::find<>()`, however, is a `const` method, and returns a `const` pointer, so the two are not always interchangeable.

In addition to direct access through the four functions just outlined, it is also possible to access the elements of a sparse matrix through iterators. As with the dense array types, the iterators are normally templated. The templated iterators are `cv::SparseMatIterator_<>` and `cv::SparseMatConstIterator_<>`, together with their corresponding `cv::SparseMat::begin<>()` and `cv::SparseMat::end<>()` routines. (The `const` forms of the `begin()` and `end()` routines return the `const` iterators.) There are also non-template iterators `cv::SparseMatIterator` and `cv::SparseMatConstIterator`, which are returned by the non-template `SparseMat::begin()` and `SparseMat::end()` routines.

Here is an example in which we print out all of the nonzero elements of a sparse array:

```
// Create a 10x10 sparse matrix with a few nonzero elements
//
int size[] = {10,10};
cv::SparseMat sm( 2, size, cv::F32 );
for( int i=0; i<10; i++ ) { // Fill the array
  int idx[2];
  idx[0] = size[0] * rand();
  idx[1] = size[1] * rand();
  sm.ref<float>( idx ) += 1.0f;
}

// Print out the nonzero elements
//
cv::SparseMatConstIterator_<float> it     = sm.begin<float>();
cv::SparseMatConstIterator_<float> it_end = sm.end<float>();

for(; it != it_end; ++it) {
  const cv::SparseMat::Node* node = it.node();
  printf(" (%3d,%3d) %f\n", node->idx[0], node->idx[1], *it );
```

[25] For those of you not familiar with "`const` correctness,"", this means that the method is declared in its prototype such that the `this` pointer passed to `SparseMat::value<>()` is guaranteed to be a constant pointer, and thus `SparseMat::value<>()` can be called on `const` objects, while functions like `SparseMat::ref<>()` cannot. The next function `SparseMat::find<>()` is also a `const` function.

```
}
```

In this example, we also slipped in the method `node()`, which is defined for the iterators. `node()` returns a pointer to the internal data node in the sparse matrix that is indicated by the iterator. The returned object of type `cv::SparseMat::Node` has the following definition:

```
struct Node
{
  size_t hashval;
  size_t next;
  int idx[cv::MAX_DIM];
};
```

This structure contains both the index of the associated element (note that element `idx` is of type `int[]`), as well as the hash value associated with that node (element `hashval` is the same hash value as can be used with `SparseMat::ptr()`, `SparseMat::ref()`, `SparseMat::value()`, and `SparseMat::find()`.)

Functions Unique to Sparse Arrays

As stated earlier, sparse matrices support many of the same operations as dense matrices. In addition, there are several methods that are unique to sparse matrices. These are listed in Table 3-19, and include the functions mentioned in the previous sections.

Table 3-19: Additional class member functions of `cv::SparseMat`

Example	Description
`cv::SparseMat sm();`	Create a sparse matrix without initialization
`cv::SparseMat sm( 3, sz, cv::F32 );`	Create a three-dimensional sparse matrix with dimensions given by the array `sz` of type `float`
`cv::SparseMat sm( sm0 );`	Create a new sparse matrix which is a copy of existing sparse matrix `sm0`
`cv::SparseMat( m0, try1d );`	Create a sparse matrix from an existing dense matrix `m0`; if the bool `try1d` is true, convert `m0` to a one-dimensional sparse matrix if the dense matrix was N-by-1 or 1-by-N
`cv::SparseMat( &old_sparse_mat );`	Create a new sparse matrix from a pointer to a pre-version 2.1 C-style sparse matrix of type `CvSparseMat`
`CvSparseMat* old_sm = (cv::SparseMat*) sm;`	Cast operator creates a pointer to a pre-version 2.1 C-style sparse matrix; that `CvSparseMat` object is created and all data is copied into it, then its pointer is returned
`size_t n = sm.nzcount();`	Return the number of nonzero elements in `sm`

`size_t h = sm.hash( i0 );`
`size_t h = sm.hash( i0, i1 );`
`size_t h = sm.hash( i0, i1, i2 );`
`size_t h = sm.hash( idx );`
	Return the hash value for element `i0` in a one-dimensional sparse matrix, `i0`, `i1` in a two-dimensional sparse matrix, `i0`, `i1`, `i2` in a three-dimensional sparse matrix, or the element indicated by the array of integers `idx` in an n-dimensional sparse matrix

`sm.ref<float>( i0 ) = f0;`
`sm.ref<float>( i0, i1 ) = f0;`
`sm.ref<float>( i0, i1, i2 ) = f0;`
`sm.ref<float>( idx ) = f0;`
	Assign the value `f0` to element `i0` in a one-dimensional sparse matrix, `i0`, `i1` in a two-dimensional sparse matrix, `i0`, `i1`, `i2` in a three-dimensional sparse matrix, or the element indicated by the array of integers `idx` in an n-dimensional sparse matrix

`f0 = sm.value<float>( i0 );`
`f0 = sm.value<float>( i0, i1 );`
`f0 = sm.value<float>( i0, i1, i2 );`
`f0 = sm.value<float>( idx );`
	Assign the value to `f0` from element `i0` in a one-dimensional sparse matrix, `i0`, `i1` in a two-dimensional sparse matrix, `i0`, `i1`, `i2` in a three-dimensional sparse matrix, or the element indicated by the array of integers `idx` in an n-dimensional sparse matrix

`p0 = sm.find<float>( i0 );`
`p0 = sm.find<float>( i0, i1 );`
`p0 = sm.find<float>( i0, i1, i2 );`
`p0 = sm.find<float>( idx );`
	Assign to `p0` the address of element `i0` in a one-dimensional sparse matrix, `i0`, `i1` in a two-dimensional sparse matrix, `i0`, `i1`, `i2` in a three-dimensional sparse matrix, or the element indicated by the array of integers `idx` in an n-dimensional sparse matrix

`sm.erase( i0, &hashval );`
`sm.erase( i0, i1, &hashval );`
`sm.erase( idx, &hashval );`
	Remove the element `i0`, `i1` in a two-dimensional sparse matrix, `i0`, `i1`, `i2` in a three-dimensional sparse matrix, or the element indicated by the array of integers `idx` in an n-dimensional sparse matrix. If `hashval` is not `NULL`, use the provided value instead of computing it

`cv::SparseMatIterator<float> it` `= sm.begin<float>();`	Create a sparse matrix iterator `it` and point it at the first value of the floating-point array `sm`
`cv::SparseMatIterator<uchar> it_end` `= sm.end<uchar>();`	Create a sparse matrix iterator `it_end` and initialize it to the value succeeding the final value in the byte array `sm`

The Template Structure

Thus far in this chapter, we have regularly eluded to the existence of template forms for almost all of the basic types. In fact, most programmers can get quite far into OpenCV without ever digging down into the templates. However, knowing how to use the templates directly can be of great help in getting things done. In this section, we will look at how that all works. If your C++ programming skills are not entirely up to par, you can probably just skim or skip over this section entirely.

OpenCV version 2.1 and later is built on a template meta-programming style similar to STL, Boost, and similar libraries. This sort of library design can be extremely powerful, both in terms of the quality and speed of the final code, as well as the flexibility it allows the developer. In particular, template structures of the kind used in OpenCV allow for algorithms to be implemented in an abstracted way that does not specifically rely on the primitive types that are native to C++ or even native to OpenCV.

As an example, consider the `cv::Point` class. When you instantiate an object of type `cv::Point`, you are actually instantiating a template object of type `cv::Point_<int>`. (Note the trailing underscore—this is the general convention in the library used to indicate a template.) This template could have been instantiated with a different type than `int`, obviously. In fact, it could have been instantiated with any type

that supports the same basic set of operators as `int` (i.e., addition, subtraction, multiplication, etc.). For example, OpenCV provides a type `cv::Complex` that you could have used. You also could have used the STL complex type `std::complex`, which has nothing to do with OpenCV at all. The same is true for some other types of your own construction. This same concept generalizes to other type templates such as `cv::Scalar_<>` and `cv::Rect_<>`, as well as `cv::Matx_<>` and `cv::Vec_<>`.

`cv::Mat_<>` and `cv::SparseMat_<>` Are a Little Bit Different

This concept also generalizes to `cv::Mat_<>` and `cv::SparseMat_<>`, but in a somewhat nontrivial way. When you use `cv::Point2i`, recall that this is nothing more or less than an alias (`typedef`) for `cv::Point_<int>`. In the case of the template `cv::Mat` and `cv::Mat_<>`, their relationship is not so simple. Recall that `cv::Mat` already has the capability of representing essentially any type, but that this is done at construction time by explicitly specifying the base type. In the case of `cv::Mat_<>`, the instantiated template is actually *derived from* the `cv::Mat` class, and in effect specializes that class. This simplifies access and other member functions that would otherwise need to be templated.

This is worth reiterating. The purpose of using the template forms `cv::Mat_<>` and `cv::SparseMat_<>` are so you don't have to use the template forms of their member functions. Consider this example, where we have a matrix defined by:

```
cv::Mat m( 10, 10, cv::F32C2 );
```

Individual element accesses to this matrix would need to specify the type of the matrix, as in the following:

```
m.at< Vec2f >( i0, i1 ) = cv::Vec2f( x, y );
```

Alternatively, if you had defined the matrix `m` using the template class, you could use `at()` without specialization, or even just use `operator()`:

```
cv::Mat_<Vec2f> m( 10, 10 );
m.at( i0, i1 ) = cv::Vec2f( x, y );
// or…
m( i0, i1 ) = cv::Vec2f( x, y );
```

There is a great deal of simplification in your code that results from using these template definitions and because if this, it is the preferred way to code.

These two ways of declaring a matrix and their associated `.at` methods are equivalent in efficiency. The second method is more "correct" because it allows the compiler to detect type mismatches when `m` is passed into a function that requires a certain type of matrix. If

```
cv::Mat m(10, 10, cv::F32C2 );
```

is passed into

```
void foo((cv::Mat_<char> *)myMat);
```

failure would occur during runtime in perhaps nonobvious ways. If you instead used

```
cv::Mat_<Vec2f> m( 10, 10 );
```

failure would be detected at compile time.

Template forms can be used to create template functions that operate on an array of a particular type. Consider our example from the previous section, where we created a small sparse matrix and then printed out its nonzero elements. We might try writing a function to achieve this as follows:

```
void print_matrix( const cv::SparseMat* sm ) {

  cv::SparseMatConstIterator_<float> it     = sm.begin<float>();
  cv::SparseMatConstIterator_<float> it_end = sm.end<float>();

  for(; it != it_end; ++it) {
    const cv::SparseMat::Node* node = it.node();
    printf(" (%3d,%3d) %f\n", node->idx[0], node->idx[1], *it );
```

```
  }
}
```

Though this function would compile and work when it is passed a two-dimensional matrix of type `cv::F32`, it would fail when a matrix of unexpected type was passed in. Let's look at how we could make this function more general.

The first thing we would want to address is the issue of the underlying data type. We could explicitly use the `cv::SparseMat_<float>` template, but it would be better still to make the function a template function. We would also need to get rid of the use of `printf()`, as it makes an explicit assumption that `*it` is a `float`. A better function might look like this:

```
template <class T> void print_matrix( const cv::SparseMat_<T>* sm ) {

  cv::SparseMatConstIterator_<T> it     = sm->begin();
  cv::SparseMatConstIterator_<T> it_end = sm->end();

  for(; it != it_end; ++it) {
    const typename cv::SparseMat_<T>::Node* node = it.node();
    cout <<"( " <<node->idx[0] <<", " <<node->idx[1]
      <<" ) = " <<*it <<endl;
  }
}

void calling_function1( void ) {
  …
  cv::SparseMat_<float> sm( ndim, size );
  …
  print_matrix<float>( &sm );
}

void calling_function2( void ) {
  …
  cv::SparseMat sm( ndim, size, cv::F32 );
  …
  print_matrix<float>( (cv::SparseMat_<float>*) &sm );
}
```

It is worth picking apart these changes. First though, before looking at changes, notice that the template for our function takes a pointer of type `const cv::SparseMat_<t>*`, a *pointer to* a sparse matrix template object. There is a good reason to use a pointer and not a reference here, because the caller may have a `cv::Mat` object (as is done in `calling_function2()`) and not a `cv::Mat_<>` template object (as used in `calling_function1()`). The `cv::Mat` can be dereferenced and then explicitly cast to a pointer to the sparse matrix template object type.

In the templated prototype, we have promoted the function to a template of `class T`, and now expect a `cv::SparseMat_<T>*` pointer as argument. In the next two lines, we declare our iterators using the template type, but `begin()` and `end()` no longer have templated instantiations. The reason for this is that `sm` is now an instantiated template, and because of that explicit instantiation, `sm` "knows" what sort of matrix it is, and thus specialization of `begin()` and `end()` is unnecessary. The declaration of the `Node` is similarly changed so that the `Node` we are using is explicitly taken from the `cv::SparseMat_<T>` instantiated template class.[26] Finally, we change the `printf()` statement to use stream output to `cout`. This has the advantage that the printing is now agnostic to the type of `*it`.

[26] The appearance of the `typename` keyword here is probably somewhat mysterious to most readers. It is a result of the dependent scoping rules in C++. If you should forget it, however, most modern compilers (e.g., g++) will throw you a friendly message reminding you to add it.

Array Operators

As we have seen so far in this chapter, there are many basic operations on arrays that are now handled by member functions of the array classes. In addition to those, however, there are many more operations that are most naturally represented as "friend" functions that either take array types as arguments, have array types as return values, or both. The functions, together with their arguments, will be covered in more detail after the table. For a shortcut, it is best to download the cheat sheet at http://docs.opencv.org/trunk/opencv_cheatsheet.pdf.

Table 3-20: Basic matrix and image operators

Function	Description
`cv::abs()`	Absolute value of all elements in an array
`cv::absdiff()`	Absolute value of differences between two arrays
`cv::add()`	Element-wise addition of two arrays
`cv::addWeighted()`	Element-wise weighted addition of two arrays (alpha blending)
`cv::bitwise_and()`	Element-wise bit-level AND of two arrays
`cv::bitwise_not()`	Element-wise bit-level NOT of two arrays
`cv::bitwise_or()`	Element-wise bit-level OR of two arrays
`cv::bitwise_xor()`	Element-wise bit-level XOR of two arrays
`cv::calcCovarMatrix()`	Compute covariance of a set of *n*-dimensional vectors
`cv::cartToPolar()`	Compute angle and magnitude from a two-dimensional vector field
`cv::checkRange()`	Check array for invalid values
`cv::compare()`	Apply selected comparison operator to all elements in two arrays
`cv::completeSymm()`	Symmetrize matrix by copying elements from one half to the other
`cv::convertScaleAbs()`	Scale array, take absolute value, then convert to 8-bit unsigned
`cv::countNonZero()`	Count nonzero elements in an array
`cv::arrToMat()`	Converts pre-version 2.1 array types to cv::Mat
`cv::dct()`	Compute discrete cosine transform of array
`cv::determinant()`	Compute determinant of a square matrix
`cv::dft()`	Compute discrete Fourier transform of array
`cv::divide()`	Element-wise division of one array by another
`cv::eigen()`	Compute eigenvalues and eigenvectors of a square matrix

`cv::exp()`	Element-wise exponentiation of array
`cv::extractImageCOI()`	Extract single channel from pre-version 2.1 array type
`cv::flip()`	Flip an array about a selected axis
`cv::gemm()`	Generalized matrix multiplication
`cv::getConvertElem()`	Get a single pixel type conversion function
`cv::getConvertScaleElem()`	Get a single pixel type conversion and scale function
`cv::idct()`	Compute inverse discrete cosine transform of array
`cv::idft()`	Compute inverse discrete Fourier transform of array
`cv::inRange()`	Test if elements of an array are within values of two other arrays
`cv::invert()`	Invert a square matrix
`cv::log()`	Element-wise natural log of array
`cv::magnitude()`	Compute magnitudes from a two-dimensional vector field
`cv::LUT()`	Converts array to indices of a look-up table
`cv::Mahalanobis()`	Compute Mahalanobis distance between two vectors
`cv::max()`	Compute element-wise maxima between two arrays
`cv::mean()`	Compute the average of the array elements
`cv::meanStdDev()`	Compute the average and standard deviation of the array elements
`cv::merge()`	Merge several single-channel arrays into one multichannel arrays
`cv::min()`	Compute element-wise minima between two arrays
`cv::minMaxLoc()`	Find minimum and maximum values in an array
`cv::mixChannels()`	Shuffle channels from input arrays to output arrays
`cv::mulSpectrums()`	Compute element-wise multiplication of two Fourier spectra
`cv::multiply()`	Element-wise multiplication of two arrays
`cv::mulTransposed()`	Calculate matrix product of one array
`cv::norm()`	Compute normalized correlations between two arrays
`cv::normalize()`	Normalize elements in an array to some value
`cv::perspectiveTransform()`	Perform perspective matrix transform of a list of vectors

`cv::phase()`	Compute orientations from a two-dimensional vector field
`cv::polarToCart()`	Compute two-dimensional vector field from angles and magnitudes
`cv::pow()`	Raise every element of an array to a given power
`cv::randu()`	Fill a given array with uniformly distributed random numbers
`cv::randn()`	Fill a given array with normally distributed random numbers
`cv::randShuffle()`	Randomly shuffle array elements
`cv::reduce()`	Reduce a two-dimensional array to a vector by a given operation
`cv::repeat()`	Tile the contents of one array into another
`cv::saturate_cast<>()`	Template function for guarding against overflow
`cv::scaleAdd()`	Element-wise sum of two arrays with optional scaling of the first
`cv::setIdentity()`	Set all elements of an array to 1 for the diagonal and 0 otherwise
`cv::solve()`	Solve a system of linear equations
`cv::solveCubic()`	Find the (only) real roots of a cubic equation
`cv::solvePoly()`	Find the complex roots of a polynomial equation
`cv::sort()`	Sort elements in either the rows or columns in an array
`cv::sortIdx()`	Same as `cv::sort()` except array is unmodified and indices are returned
`cv::split()`	Split a multichannel array into multiple single-channel arrays
`cv::sqrt()`	Compute element-wise square root of an array
`cv::subtract()`	Element-wise subtraction of one array from another
`cv::sum()`	Sum all elements of an array
`cv::theRNG()`	Return a random number generator
`cv::trace()`	Compute the trace of an array
`cv::transform()`	Apply matrix transformation on every element of an array
`cv::transpose()`	Transpose all elements of an array across the diagonal

In these functions, some general rules are followed. To the extent that any exceptions exist, they are noted in the function descriptions. Because one or more of these rules applies to just about every function described in this section, they are listed here for convenience:

Saturation

Outputs of calculations are saturation casted to the type of the output array.

Output

The output array will be created with `cv::Mat::create()` if its type and size do not match the inputs.

Scalars

Many functions such as `cv::add()` allow for addition of two arrays or an array and a scalar. Where the prototypes make the option clear, the result of providing a scalar argument is the same as if a second array had been provided with the same scalar value in every element.

Masks

Whenever a mask argument is present for a function, the output will only be computed for those elements where the mask value corresponding to that element in the output array is nonzero.

`dtype`

Many arithmetic and similar functions do not require the types of the input arrays to be the same, and even if they are the same, the output array may be of a different type than the inputs. In these cases, the output array must have its depth explicitly specified. This is done with the `dtype` argument. When present, `dtype` can be set to any of the basic types (`cv::F32` etc.) and the result array will be of that type. If the input arrays have the same type, then `dtype` can be set to its default value of `-1`, and the resulting type will be the same as the types of the input arrays.

In Place Operation

Unless otherwise specified, any operation with both an array input and an array output that are of the same size and type can use the same array for both (i.e., it is allowable to write the output on top of an input).

Multichannel

For those operations that do not naturally make use of multiple channels, if given multichannel arguments, each channel is processed separately.

`cv::abs()`

```
cv::MatExpr cv::abs( cv::InputArray src );
cv::MatExpr cv::abs( const cv::MatExpr& src                    // Matrix expresion
);
```

These functions compute the absolute value of an array or of some expression of arrays. The most common usage computes the absolute value of every element in an array. Because `cv::abs()` can take a matrix expression, it is able to recognize certain special cases and handle them appropriately. In fact, calls to `cv::abs()` are actually converted to calls to `cv::absDiff()` or other functions, and handled by those functions. In particular, the following special cases are implemented:

- `m2 = cv::abs( m0 - m1 )` is converted to `cv::absDiff( m0, m1, m2 )`
- `m2 = cv::abs( m0 )` is converted to `m2 = cv::absDiff( m0, cv::Scalar::all(0), m2 )`
- `m2 = cv::Mat_<Vec<uchar,n> >( cv::abs( alpha*m0 + beta ) )` (for alpha, beta real numbers) is converted to `cv::convertScaleAbs( m0, m2, alpha, beta )`

The third case might seem obscure, but this is just the case of computing a scale and offset (either of which could be trivial) to an n-channel array. This is typical of what one might do when computing a contrast correction for an image, for example.

In the cases that are implemented by `cv::absDiff()`, the result array will have the same size and type as the input array. In the case implemented by `cv::convertScaleAbs()`, however, the result type of the return array will always be `cv::U8`.

cv::absdiff()

```
void cv::absdiff(
  cv::InputArray  src1,                    // First input array
  cv::InputArray  src2,                    // Second input array
  cv::OutputArray dst                      // Result array
)
```

$$dst_i = saturate\left(\dagger\left|src1_i - src2_i\right|\right)$$

Given two arrays, `cv::absdiff()` computes the difference between each pair of corresponding elements in those arrays, and places the absolute value of that difference into the corresponding element of the destination array.

cv::add()

```
void cv::add(
  cv::InputArray  src1,                    // First input array
  cv::InputArray  src2,                    // Second input array
  cv::OutputArray dst,                     // Result array
  cv::InputArray  mask  = cv::noArray(),   // Optional mask, compute only where nonzero
  int             dtype = -1               // Output type for result array
)
```

$$dst_i = saturate\left(\dagger src1_i + src2_i\right)$$

`cv::add()` is a simple addition function: it adds all of the elements in `src1` to the corresponding elements in `src2` and puts the results in `dst`.

For simple cases, the same result can be achieved with the matrix operation:

```
dst = src1 + src2;
```

Accumulation is also supported:

```
dst += src1;
```

---

cv::addWeighted()

```
void cv::addWeighted(
  cv::InputArray  src1,                    // First input array
  double          alpha,                   // Weight for first input array
  cv::InputArray  src2,                    // Second input array
  double          beta,                    // Weight for second input array
  double          gamma,                   // Offset added to weighted sum
  cv::OutputArray dst,                     // Result array
  int             dtype = -1               // Output type for result array
)
```

The function `cv::addWeighted()` is similar to `cvAdd()` except that the result written to `dst` is computed according to the following formula:

$$dst_i = saturate\left(\dagger src1_i * \alpha\alpha\dagger + src2_i * \beta\beta\dagger + \gamma\right)\left(\dagger src1_i * \alpha\alpha\dagger + src2_i * \beta\beta\dagger + \gamma\right).$$

The two source images, `src1` and `src2` may be of any pixel type as long as both are of the same type. They may also have any number of channels (grayscale, color, etc.), as long as they agree.

This function can be used to implement *alpha blending* [Smith79; Porter84]; that is, it can be used to blend one image with another. In this case, the parameter `alpha` is the blending strength of `src1`, and `beta` is the blending strength of `src2`. You can convert to the standard alpha blend equation by choosing α between 0 and 1, setting $\beta = 1 - \alpha$, and setting γ to 0; this yields:

$$dst_i = saturate\Big(\dagger src1_i * \alpha\alpha\dagger + src2_i * \big(1-\alpha\dagger\big)\big(1-\alpha\dagger\big)\Big)\Big(\dagger src1_i * \alpha\alpha\dagger + src2_i * \big(1-\alpha\alpha\dagger\big)\Big).$$

However, `cv::addWeighted()` gives us a little more flexibility—both in how we weight the blended images and in the additional parameter γ, which allows for an additive offset to the resulting destination image. For the general form, you will probably want to keep `alpha` and `beta` at 0 or above, and their sum at no more than 1; `gamma` may be set depending on average or max image value to scale the pixels up. A program showing the use of alpha blending is shown in Example 3-1.

Example 3-1. Complete program to alpha blend the ROI starting at $(0,0)$ *in src2 with the ROI starting at* (x,y) *in src1*

```
// alphablend <imageA> <image B> <x> <y> <width> <height> alpha> <beta>
//
#include <cv.h>
#include <highgui.h>

int main(int argc, char** argv) {

  cv::Mat src1 = cv::imread(argv[1],1);
  cv::Mat src2 = cv::imread(argv[2],1);

  if( argc==9 && !src1.empty() && !src2.empty() ) {

    int    x     = atoi(argv[3]);
    int    y     = atoi(argv[4]);
    int    w     = atoi(argv[5]);
    int    h     = atoi(argv[6]);
    double alpha = (double)atof(argv[7]);
    double beta  = (double)atof(argv[8]);

    cv::Mat roi1( src1, cv::Rect(x,y,w,h) );
    cv::Mat roi2( src2, cv::Rect(0,0,w,h) );

    cv::addWeighted( roi1, alpha, roi2, beta, 0.0, roi1 );

    cv::namedWindow( "Alpha Blend", 1 );
    cv::imshow( "Alpha Blend", src2 );
    cv::waitKey( 0 );
  }

  return 0;
}
```

The code in Example 3-1 takes two source images: the primary one (`src1`) and the one to blend (`src2`). It reads in a rectangle ROI for `src1` and applies an ROI of the same size to `src2`, but located at the origin. It reads in `alpha` and `beta` levels but sets `gamma` to 0. Alpha blending is applied using `cv::addWeighted()`, and the results are put into `src1` and displayed. Example output is shown in Figure 3-1, where the face of a child is blended onto a cat. Note that the code took the same ROI as in the ROI addition example in Figure 3-1. This time we used the ROI as the target blending region.

Figure 3-1: The face of a child is alpha blended onto the face of a cat

cv::bitwise_and()

```
void cv::bitwise_and(
  cv::InputArray  src1,                  // First input array
  cv::InputArray  src2,                  // Second input array
  cv::OutputArray dst,                   // Result array
  cv::InputArray  mask  = cv::noArray(), // Optional mask, compute only where nonzero
)
```

$$dst_i = src1_i \wedge src2_i$$

`cv::bitwise_and()` is a per-element bitwise conjunction operation. For every element in `src1`, the bitwise AND is computed with the corresponding element in `src2` and put into the corresponding element of `dst`.

If you are not using a mask, the same result can be achieved with the matrix operation:

```
dst = src1 & src2;
```

---

cv::bitwise_not()

```
void cv::bitwise_not(
  cv::InputArray  src,                   // Input array
  cv::OutputArray dst,                   // Result array
  cv::InputArray  mask  = cv::noArray(), // Optional mask, compute only where nonzero

)
```

$$dst_i = \sim src1_i$$

`cv::bitwise_not()` is a per-element bitwise inversion operation. For every element in `src1` the logical inversion is computed and placed into the corresponding element of `dst`.

If you are not using a mask, the same result can be achieved with the matrix operation:

```
dst = !src1;
```

---

cv::bitwise_or()

```
void cv::bitwise_and(
  cv::InputArray  src1,                  // First input array
  cv::InputArray  src2,                  // Second input array
  cv::OutputArray dst,                   // Result array
  cv::InputArray  mask  = cv::noArray(), // Optional mask, compute only where nonzero
)
```

$$dst_i = src1_i \vee src2_i$$

$$dst_i = src1_i \vee sc$$

`cv::bitwise_or()` is a per-element bitwise disjunction operation. For every element in `src1`, the bitwise OR is computed with the corresponding element in `src2` and put into the corresponding element of `dst`.

If you are not using a mask, the same result can be achieved with the matrix operation:

```
dst = src1 | src2;
```

---

cv::bitwise_xor()

```
void cv::bitwise_and(
  cv::InputArray  src1,                  // First input array
  cv::InputArray  src2,                  // Second input array
  cv::OutputArray dst,                   // Result array
  cv::InputArray  mask  = cv::noArray(), // Optional mask, compute only where nonzero
)
```

$$dst_i = src1_i \oplus src2_i$$

$$dst_i = src1_i \oplus sc$$

`cv::bitwise_and()` is a per-element bitwise "exclusive or" operation. For every element in `src1`, the bitwise XOR is computed with the corresponding element in `src2` and put into the corresponding element of `dst`.

If you are not using a mask, the same result can be achieved with the matrix operation:

```
dst = src1 ^ src2;
```

---

cv::calcCovarMatrix()

```
void cv::calcCovarMatrix(
  const cv::Mat* samples,              // C-array of n-by-1 or 1-by-n matrices
  int            nsamples,             // number of matrices pointed to by 'samples'
  cv::Mat&       covar,                // reference to return array for covariance
  cv::Mat&       mean,                 // reference to return array for mean
  int            flags,                // special variations, see Table 3-21
  int            ctype = cv::F64        // output matrix type for covar
);

void cv::calcCovarMatrix(
  cv::InputArray samples,              // n-by-m matrix, use 'flags' for which is which
  cv::Mat&       covar,                // reference to return array for covariance
  cv::Mat&       mean,                 // reference to return array for mean
  int            flags,                // special variations, see Table 3-21
  int            ctype = cv::F64        // output matrix type for covar
);
```

Given any number of vectors, `cv::calcCovarMatrix()` will compute the mean and *covariance matrix* for the Gaussian approximation to the distribution of those points. This can be used in many ways,

of course, and OpenCV has some additional flags that will help in particular contexts (see Table 3-21). These flags may be combined by the standard use of the Boolean OR operator.

Table 3-21: Possible components of flags argument to `cv::calcCovarMatrix()`

Flag in flags argument	Meaning
`cv::COVAR_NORMAL`	Compute mean and covariance
`cv::COVAR_SCRAMBLED`	Fast PCA "scrambled" covariance
`cv::COVAR_USE_AVERAGE`	Use mean as input instead of computing it
`cv::COVAR_SCALE`	Rescale output covariance matrix
`cv::COVAR_ROWS`	Use rows of `samples` for input vectors
`cv::COVAR_COLS`	Use columns of `samples` for input vectors

There are two basic calling conventions for `cv::calcCovarMatrix()`. In the first, a pointer to an array of `cv::Mat` objects is passed along with `nsamples`, the number of matrices in that array. In this case, the arrays may be *n*-by-1 or 1-by-*n*. The second calling convention is to pass a single array that is *n*-by-*m*. In this case, either the flag `cv::COVAR_ROWS` should be supplied, to indicate that there are *n* (row) vectors of length *m*, or `cv::COVAR_COLS` should be supplied, to indicate that there are *m* (column) vectors of length *n*.

The results will be placed in `covar` in all cases, but the exact meaning of `avg` depends on the flag values (see Table 3-21).

The flags `cv::COVAR_NORMAL` and `cv::COVAR_SCRAMBLED` are mutually exclusive; you should use one or the other but not both. In the case of `cv::COVAR_NORMAL`, the function will simply compute the mean and covariance of the points provided:

Thus the normal covariance Σ^2_{normal} is computed from the m vectors of length n, where $\overline{v}_n$ is defined as the n^{th} element of the average vector: $\overline{v}$. The resulting covariance matrix is an n-by-n matrix. The factor z is an optional scale factor; it will be set to 1 unless the `cv::COVAR_SCALE` flag is used.

In the case of `cv::COVAR_SCRAMBLED`, `cv::calcCovarMatrix()` will compute the following:

This matrix is not the usual covariance matrix (note the location of the transpose operator). This matrix is computed from the same *m* vectors of length *n*, but the resulting *scrambled covariance* matrix is an *m*-by-*m* matrix. This matrix is used in some specific algorithms such as fast PCA for very large vectors (as in the *eigenfaces* technique for face recognition).

The flag `cv::COVAR_USE_AVG` is used when the mean of the input vectors is already known. In this case, the argument `avg` is used as an input rather than an output, which reduces computation time.

Finally, the flag `cv::COVAR_SCALE` is used to apply a uniform scale to the covariance matrix calculated. This is the factor *z* in the preceding equations. When used in conjunction with the `cv::COVAR_NORMAL` flag, the applied scale factor will be $1/m$ (or, equivalently, `1.0/nsamples`). If instead `cv::COVAR_SCRAMBLED` is used, then the value of *z* will be $1/n$ (the inverse of the length of the vectors).

The input and output arrays to `cv::calcCovarMatrix()` should all be of the same floating-point type. The size of the resulting matrix `covar` will be either n-by-n or m-by-m depending on whether the standard or scrambled covariance is being computed. It should be noted that when using the `cv::Mat*` form, the "vectors" input in `samples` do not actually have to be one-dimensional; they can be two-dimensional objects (e.g., images) as well.

cv::cartToPolar()

```
void cv::cartToPolar(
  cv::InputArray  x,
  cv::InputArray  y,
  cv::OutputArray magnitude,
  cv::OutputArray angle,
  bool            angleInDegrees = false
);
```

$$magnitude_i = \dagger\sqrt{x_i^2 + y_i^2}$$

$$angle_i = atan2\left(y_i, \dagger x_i\right)$$

This function `cv::cartToPolar()` takes two input arrays `x` and `y`, which are taken to be the x and y components of a vector field (note that this is not a single two-channel array, but two separate arrays). The arrays `x` and `y` must be of the same size. `cv::cartToPolar()` then computes the polar representation of each of those vectors. The magnitude of each vector is placed into the corresponding location in `magnitude`, and the orientation of each vector is placed into the corresponding location in `angle`. The returned angles are in radians unless the Boolean variable `angleInDegrees` is set to `true`.

cv::checkRange()

```
bool cv::checkRange(
  cv::InputArray src,
  bool           quiet  = true,
  Point*     pos    = 0,            // if non-Null, location of first outlier
  double         minVal = -DBL_MAX,     // Lower check bound (inclusive)
  double         maxVal =  DBL_MAX      // Upper check bound (exclusive)
);
```

This function `cv::checkRange()` tests every element of the input array `src` and determines if that element is in a given range. The range is set by `minVal` and `maxVal`, but any `NaN` or `inf` value is also considered out of range. If an out-of-range value is found, an exception will be thrown unless `quiet` is set to `true`, in which case the return value of `cv::checkRange()` will be `true` if all values are in range and `false` if any value is out of range. If the pointer `pos` is not `NULL`, then the location of the first outlier will be stored in `pos`.

cv::compare()

```
bool cv::compare(
  cv::InputArray  src1,                  // First input array
  cv::InputArray  src2,                  // Second input array
  cv::OutputArray dst,                   // Result array
  int             cmpop                  // Comparison operator, see Table 3-22
)
```

This function makes element-wise comparisons between corresponding pixels in two arrays, `src1` and `src2`, and places the results in the image `dst`. `cv::compare()` takes as its last argument a comparison operator, which may be any of the types listed in Table 3-22. In each case, the result `dst` will be an 8-bit array where pixels that match are marked with `255` and mismatches are set to `0`.

Table 3-22 Values of `cmpop` *used by* `cv::compare()` *and the resulting comparison operation performed*

Value of cmp_op	Comparison
`cv::CMP_EQ`	`(src1i == src2i)`
`cv::CMP_GT`	`(src1i  > src2i)`
`cv::CMP_GE`	`(src1i >= src2i)`

`cv::CMP_LT`	`(src1i  < src2i)`
`cv::CMP_LE`	`(src1i <= src2i)`
`cv::CMP_NE`	`(src1i != src2i)`

All the listed comparisons are done with the same functions; you just pass in the appropriate argument to indicate what you would like done.

These comparison functions are useful, for example, in background subtraction to create a mask of the changed pixels (e.g., from a security camera) such that only novel information is pulled out of the image.

These same results can be achieved with the matrix operations:

```
dst = src1 == src2;
dst = src1  > src2;
dst = src1 >= src2;
dst = src1  < src2;
dst = src1 <= src2;
dst = src1 != src2;
```

`cv::completeSymm()`

```
bool cv::completeSymm(
  cv::InputArray mtx,
  bool           lowerToUpper = false
)
```

$$mtx_{ij} = mtx_{ji} \quad \forall i > j \qquad \texttt{(lowerToUpper = false)}$$

$$mtx_{ij} = mtx_{ji} \quad \forall j > i \qquad \texttt{(lowerToUpper = false)}$$

Given a matrix (an array of dimension two) `mtx`, `cv::completeSymm()` symmetrizes the matrix by copying.[27] Specifically, all of the elements from the lower triangle are copied to their transpose position on the upper triangle of the matrix. The diagonal elements of the `mtx` are left unchanged. If the flag `lowerToUpper` is set to `true`, then the elements from the upper triangle are copied into the lower triangle instead.

`cv::convertScaleAbs()`

```
void cv::convertScaleAbs(
  cv::InputArray  src,                  // Input array
  cv::OutputArray dst,                  // Result array
  double          alpha = 1.0,          // Multiplicative scale factor
  double          beta  = 0.0           // Additive offset factor
);
```

$$dst_i = saturate_{uchar}\left(\left|\alpha * src_i + \beta\right|\right)$$

The `cv::convertScaleAbs()` function is actually several functions rolled into one; it will perform four operations in sequence. The first operation is to rescale the source image by the factor alpha, the second is to offset by (add) the factor beta, the third is to compute the absolute value of that sum, and the fourth operation is to cast that result (with saturation) to unsigned char (8-bit).

---

[27] Mathematically inclined readers will realize that there are other symmetrizing processes for matrices that are more "natural" than this operation, but this particular operation is useful in its own right, for example to complete a matrix when only half of it was computed, and so is exposed in the library.

When you simply pass the default values (`alpha = 1.0` or `beta = 0.0`), you need not have performance fears; OpenCV is smart enough to recognize these cases and not waste processor time on useless operations.

A similar result can be achieved, with somewhat greater generality using matrix algebra:

```
Mat_<uchar> B = saturate_cast<uchar>(
  cv::abs( src * alpha + beta )
)
```

This method will also generalize to types other than `uchar`:

```
Mat_<unsigned short> B = saturate_cast<unsigned short>(
  cv::abs( src * alpha + beta )
)
```

---

cv::countNonZero()

```
int cv::countNonZero(              // Return number of non-zero elements in mtx
  cv::InputArray mtx,              // Input array
);
```

$$count = \sum_{mtx_i \neq 0} 1$$

`cv::countNonZero()` returns the number of nonzero pixels in the array `mtx`.

cv::cvarrToMat()

```
cv::Mat cv::cvarrToMat(
  const CvArr* src,              // Input array: CvMat, IplImage, or CvMatND
  bool         copyData = false, // if false just make new header, else copy data
  bool         allowND  = true,  // if true, and possible, convert CvMatND to Mat
  int          coiMode  = 0      // if 0: error if COI set, if 1: ignore COI
);
```

`cv::cvarrToMat()` is used when you have an "old-style" (pre-version 2.1) image or matrix type and you want to convert it to a "new-style" (version 2.1 or later, which uses C++) cv::Mat object. By default, only the header for the new array is constructed without copying the data. Instead, the data pointers in the new header point to the existing data array (so do not deallocate it while the cv::Mat header is in use). If you want the data copied, just set `copyData` to `true`, and then you can freely do away with the original data object.

`cv::cvarrToMat()` can also take `CvMatND` structures, but it cannot handle all cases. The key requirement for the conversion is that the matrix should be continuous, or at least it should be representable as a sequence of continuous matrices. Specifically, `A.dim[i].size*A.dim.step[i]` should be equal to `A.dim.step[i-1]` for all `i`, or at worst all but one. If `allowND` is set to `true` (default), `cv::cvarrToMat()` will attempt the conversion when it encounters a `CvMatND` structure, and throw an exception if that conversion is not possible (the condition above). If `allowND` is set to `false`, then an exception will be thrown whenever a `CvMatND` structure is encountered.

Because the concept of COI[28] is handled differently in the post-version 2.1 library (which is to say, it no longer exists), COI has to be handled during the conversion. If the argument `coiMode` is `0`, then an exception will be thrown when `src` contains an active COI. If `coiMode` is nonzero, then no error will be reported, and instead a cv::Mat header, which corresponds to the entire image, will be returned, ignoring

---

[28] "COI" is an old concept from the pre-v2 library which meant "Channel of interest". In the old `IplImage` class, this COI was analogous to ROI (Region of interest), and could be set to cause certain functions to act only on the indicated channel.

the COI. (If you want to handle COI properly, you will have to check yourself if the image has the COI set, and if so, use `cv::extractImageCOI()` to create a header for just that channel.)

Most of the time, this function is used to help migrate old-style code to the new. In such case, you will probably need to both convert old `CvArr*` style structures to `cv::Mat`, as well as the reverse operation. The reverse operation is done using cast operators. If, for example, you have a matrix you defined as `cv::Mat A`, you can convert that to an `IplImage*` pointer simply with:

```
Mat A( 640, 480, cv::U8C3 );
IplImage img = A;   // casting is implicit in assignment
```

`cv::dct()`

```
void cv::dct(
  cv::InputArray  src,                    // Input array
  cv::OutputArray dst,                    // Result array
  int             flags,                  // for inverse transform or row-by-row
);
```

This function performs both the discrete cosine transform and the inverse transform depending on the `flags` argument. The source array `src` must be either one- or two-dimensional, and the size should be an even number (you can pad the array if necessary). The result array `dst` will have the same type and size as `src`. The argument `flags` is a bit-field and can be set to one or both of `DCT_INVERSE` or `DCT_ROWS`. If `DCT_INVERSE` is set, then the inverse transform is done instead of the forward transform. If the flag `DCT_ROWS` is set, then a two-dimensional *n*-by-*m* input is treated as *n* distinct one-dimensional vectors of length *m*. In this case, each such vector will be transformed independently.

The performance of `cv::dct()` depends strongly on the exact size of the arrays passed to it, and this relationship is not monotonic. There are just some sizes that work better than others. It is recommended that when passing an array to `cv::dct()`, you should first determine the most optimal size that is larger than your array, and extend your array to that size. OpenCV provides a convenient routine to compute such values for you, called `cv::getOptimalDFTSize()`.

As implemented, the discrete cosine transform of a vector of length *n* is computed using the discrete Fourier transform (`cv::dft()`) on a vector of length *n/2*. This means that to get the optimal size for a call to `cv::dct()`, you should compute it like this:

```
size_t optimal_dct_size = 2 * getOptimalDFTSize( (N+1) / 2 );
```

This function (and discrete transforms in general) is covered in much greater detail in Chapter 6, General Image Transforms. In that section, we will discuss the details of how to pack and unpack the input and output, as well as information on when and why you might want to use the discrete cosine transform.

`cv::dft()`

```
void cv::dft(
  cv::InputArray  src,                    // Input array
  cv::OutputArray dst,                    // Result array
  int             flags       = 0,        // for inverse transform or row-by-row, etc.
  int             nonzeroRows = 0         // assume only this many entries are nonzero
);
```

This function performs both the discrete Fourier transform as well as the inverse transform (depending on the `flags` argument). The source array `src` must be either one- or two--dimensional. The result array `dst` will have the same type and size as `src`. The argument `flags` is a bit-field and can be set to one or more of `DFT_INVERSE`, `DFT_ROWS`, `DFT_SCALE`, `DFT_COMPLEX_OUTPUT`, or `DFT_REAL_OUTPUT`. If `DFT_INVERSE` is set, then the inverse transform is done. If the flag `DFT_ROWS` is set, then a two-dimensional n-by-m input is treated as n distinct one-dimensional vectors of length m

and each such vector will be transformed independently. The flag DFT_SCALE normalizes the results by the number of elements in the array. This is normally done for DFT_INVERSE, as it guarantees that the inverse of the inverse will have the correct normalization.

The flags DFT_COMPLEX_OUTPUT and DFT_REAL_OUTPUT are useful because when the Fourier transform of a real array is computed, the result will have a complex-conjugate symmetry. So, even though the result is complex, the number of array elements that result will be equal to the number of elements in the real input array rather than double that number. Such a packing is the default behavior of cv::dft(). To force the output to be in complex form, set the flag DFT_COMPLEX_OUTPUT. In the case of the inverse transform, the input is (in general) complex, and the output will be as well. However, if the input array (to the inverse transform) has complex-conjugate symmetry (for example, if it was itself the result of a Fourier transform of a real array), then the inverse transform will be a real array. If you know this to be the case, and you would like the result array represented as a real array (and so use half the amount of memory), you can set the DFT_REAL_OUTPUT flag. (Note that if you do set this flag, cv::dft() does not check that the input array has the necessary symmetry, it simply assumes that it does.)

The last parameter to cv::dft() is nonzeroRows. This defaults to 0, but if set to any nonzero value, will cause cv::dft() to assume that only the first nonzeroRows of the input array are actually meaningful. (If DFT_INVERSE is set, then it is only the first nonzeroRows of the output array that are assumed to be nonzero.) This flag is particularly handy when computing cross-correlations of convolutions using cv::dft().

> Like the performance of cv::dct(), cv::dft() depends strongly on the exact size of the arrays passed to it, and this relationship is not monotonic. There are just some sizes that work better than others. It is recommended that when passing an array to cv::dft(), you should first determine the most optimal size larger than your array, and extend your array to that size. OpenCV provides a convenient routine to compute such values for you, called cv::getOptimalDFTSize().

Again, this function (and discrete transforms in general) is covered in much greater detail in Chapter 6, General Image Transforms. In that section, we will discuss the details of how to pack and unpack the input and output, as well as information on when and why you might want to use the discrete Fourier transform.

cv::cvtColor()

```
void cv::cvtColor(
  cv::InputArray  src,                    // Input array
  cv::OutputArray dst,                    // Result array
  int             code,                   // color mapping code, see Table 3-23
  int             dstCn = 0               // channels in output (0 for 'automatic')
);
```

cv::cvtColor() is used to convert from one color space (number of channels) to another [Wharton71] while retaining the same data type. The input array src can be an 8-bit array, a 16-bit unsigned array, or a 32-bit floating-point array. The output array dst will have the same size and depth as the input array. The conversion operation to be done is specified by the code argument, with possible values shown in *Table 3-23*.[29] The final parameter dstCn is the desired number of channels in the destination image. If the default value (of 0) is given, then the number of channels is determined by the number of channels in src and the conversion code.

Table 3-23: Conversions available by means of cv::cvtColor()

Conversion code	Meaning

[29] Long-time users of IPL should note that the function cvCvtColor() ignores the colorModel and channelSeq fields of the IplImage header. The conversions are done exactly as implied by the code argument.

`cv::BGR2RGB`
`cv::RGB2BGR`
`cv::RGBA2BGRA`
`cv::BGRA2RGBA`
	Convert between RGB and BGR color spaces (with or without alpha channel)

`cv::RGB2RGBA` `cv::BGR2BGRA`	Add alpha channel to RGB or BGR image
`cv::RGBA2RGB` `cv::BGRA2BGR`	Remove alpha channel from RGB or BGR image

`cv::RGB2BGRA`
`cv::RGBA2BGR`
`cv::BGRA2RGB`
`cv::BGR2RGBA`
	Convert RGB to BGR color spaces while adding or removing alpha channel

`cv::RGB2GRAY` `cv::BGR2GRAY`	Convert RGB or BGR color spaces to grayscale

`cv::GRAY2RGB`
`cv::GRAY2BGR`
`cv::RGBA2GRAY`
`cv::BGRA2GRAY`
	Convert grayscale to RGB or BGR color spaces (optionally removing alpha channel in the process)

`cv::GRAY2RGBA` `cv::GRAY2BGRA`	Convert grayscale to RGB or BGR color spaces and add alpha channel

`cv::RGB2BGR565`
`cv::BGR2BGR565`
`cv::BGR5652RGB`
`cv::BGR5652BGR`
`cv::RGBA2BGR565`
`cv::BGRA2BGR565`
`cv::BGR5652RGBA`
`cv::BGR5652BGRA`
	Convert from RGB or BGR color space to BGR565 color representation with optional addition or removal of alpha channel (16-bit images)

`cv::GRAY2BGR565` `cv::BGR5652GRAY`	Convert grayscale to BGR565 color representation or vice versa (16-bit images)

`cv::RGB2BGR555`
`cv::BGR2BGR555`
`cv::BGR5552RGB`
`cv::BGR5552BGR`
`cv::RGBA2BGR555`
`cv::BGRA2BGR555`
`cv::BGR5552RGBA`
`cv::BGR5552BGRA`
	Convert from RGB or BGR color space to BGR555 color representation with optional addition or removal of alpha channel (16-bit images)

`cv::GRAY2BGR555` `cv::BGR5552GRAY`	Convert grayscale to BGR555 color representation or vice versa (16-bit images)

`cv::RGB2XYZ`
`cv::BGR2XYZ`
`cv::XYZ2RGB`
`cv::XYZ2BGR`
	Convert RGB or BGR image to CIE XYZ representation or vice versa (Rec 709 with D65 white point)

`cv::RGB2YCrCb`
`cv::BGR2YCrCb`
`cv::YCrCb2RGB`
`cv::YCrCb2BGR`
	Convert RGB or BGR image to luma-chroma (aka YCC) color representation or vice versa

`cv::RGB2HSV`
`cv::BGR2HSV`
`cv::HSV2RGB`
`cv::HSV2BGR`
	Convert RGB or BGR image to HSV (hue saturation value) color representation or vice versa

`cv::RGB2HLS`
`cv::BGR2HLS`
`cv::HLS2RGB`
`cv::HLS2BGR`
	Convert RGB or BGR image to HLS (hue lightness saturation) color representation or vice versa

`cv::RGB2Lab`
`cv::BGR2Lab`
`cv::Lab2RGB`
`cv::Lab2BGR`
	Convert RGB or BGR image to CIE Lab color representation or vice versa

`cv::RGB2Luv`
`cv::BGR2Luv`
`cv::Luv2RGB`
`cv::Luv2BGR`
	Convert RGB or BGR image to CIE Luv color representation or vice versa

`cv::BayerBG2RGB`
`cv::BayerGB2RGB`
`cv::BayerRG2RGB`
`cv::BayerGR2RGB`
`cv::BayerBG2BGR`
`cv::BayerGB2BGR`
`cv::BayerRG2BGR`
`cv::BayerGR2BGR`
	Convert from Bayer pattern (single-channel) to RGB or BGR image

We will not go into the details of these conversions nor the subtleties of some of the representations (particularly the Bayer and the CIE color spaces) here. Instead, we will just note that OpenCV contains tools to convert to and from these various color spaces, which are of importance to various classes of users.

The color-space conversions all use the conventions: 8-bit images are in the range `0` to `255`, 16-bit images are in the range `0` to `65536`, and floating-point numbers are in the range `0.0` to `1.0`. When grayscale images are converted to color images, all components of the resulting image are taken to be equal; but for the reverse transformation (e.g., RGB or BGR to grayscale), the gray value is computed using the perceptually weighted formula:

$$Y = \left(0.299\right)R + \left(0.587\right)G + \left(0.114\right)B$$

In the case of HSV or HLS representations, hue is normally represented as a value from 0 to 360.[30] This can cause trouble in 8-bit representations and so, when converting to HSV, the hue is divided by 2 when the output image is an 8-bit image.

`cv::determinant()`

```
double cv::determinant(
  cv::InputArray mat
);
```

$$d = \det\left(mat\right)$$

`cv::determinant()` computes the determinant of a square array. The array must be of one of the floating-point data types and must be single-channel. If the matrix is small, then the determinant is directly computed by the standard formula. For large matrices, this is not efficient and so the determinant is computed by *Gaussian elimination*.

> It is worth noting that if you know that a matrix has a symmetric and positive determinant, you can use the trick of solving via *singular value decomposition* (SVD). For more information, see the upcoming section on `cv::SVD()`, but the trick is to set

[30] Excluding 360, of course.

both `U` and `V` to `NULL` and then just take the products of the matrix `W` to obtain the determinant.

cv::divide()

```
void cv::divide(
  cv::InputArray  src1,                 // First input array (numerators)
  cv::InputArray  src2,                 // Second input array (denominators)
  cv::OutputArray dst,                  // Results array (scale*src1/src2)
  double          scale = 1.0,          // Multiplicative scale factor
  int             dtype = -1            // Data type for dst, -1 to get from src2
)

void cv::divide(
  double          scale,                // Numerator for all divisions
  cv::InputArray  src2,                 // Input array (denominators)
  cv::OutputArray dst,                  // Results array (scale/src2)
  int             dtype = -1            // Data type for dst, -1 to get from src2
)
```

$$dst_i = saturate\left(scale * \dagger src1_i / src2_i\right)$$

$$dst_i = saturate\left(scale / src2_i\right)$$

`cv::divide()` is a simple division function; it divides all of the elements in `src1` by the corresponding elements in `src2` and puts the results in `dst`.

cv::eigen()

```
bool cv::eigen(
  cv::InputArray  src,
  cv::OutputArray eigenvalues,
  int             lowindex     = -1,
  int             highindex    = -1
);

bool cv::eigen(
  cv::InputArray  src,
  cv::OutputArray eigenvalues,
  cv::OutputArray eigenvectors,
  int             lowindex     = -1,
  int             highindex    = -1
);
```

Given a symmetric matrix `mat`, `cv::eigen()` will compute the *eigenvectors* and *eigenvalues* of that matrix. The matrix must be of one of the floating-point types. The results array `eigenvalues` will contain the eigenvalues of `mat` in descending order. If the array `eigenvectors` was provided, the eigenvectors will be stored as the rows of that array in the same order as the corresponding eigenvalues in `eigenvalues`. The additional parameters `lowindex` and `highindex` allow you to request only some of the eigenvalues to be computed (both must be used together). For example, if `lowindex=0` and `highindex=1`, then only the largest two eigenvectors will be computed. Regardless of the number of eigenvalues (and eigenvectors) requested, the result arrays will be of the same size (the requested values will always appear in the initial rows).

Similar to `cv::det()` (described previously), if the matrix in question is symmetric and positive definite,[31] then it is better to use SVD to find the eigenvalues and eigenvectors of mat.

cv::exp()

```
void cv::exp(
  cv::InputArray  src,
  cv::OutputArray dst
);
```

$$dst_i = e^{src_i}$$

`cv::exp()` exponentiates all of the elements in `src1` and puts the results in `dst`.

cv::extractImageCOI()

```
bool cv::extractImageCOI(
  cv::InputArray  mat,
  cv::OutputArray dst,
  int             coi = -1
);
```

The function `cv::extractImageCOI()` extracts the indicated COI from a legacy-style (pre-version 2.1) array, such as an `IplImage` or `CvMat` given by `src` and puts the result in `dst`. If the argument `coi` is provided, then that particular COI will be extracted. If not, then the COI field in `src` will be checked to determine which channel to extract.

This method here specifically works with legacy arrays. If you need to extract a single channel from a modern `cv::Mat` object, use `cv::mixChannels()` or `cv::split()`.

cv::flip()

```
void cv::flip(
  cv::InputArray  src,                    // Input array
  cv::OutputArray dst,                    // Results array, size and type of 'src'
  int             flipCode = 0            // >0: y-flip, 0: x-flip, <0: both
);
```

This function flips an image around the x-axis, the y-axis, or both. By default, `flipCode` is set to `0`, which flips around the *x*-axis.

If `flipCode` is set greater than zero (e.g., `+1`), the image will be flipped around the y-axis, and if set to a negative value (e.g., `-1`), the image will be flipped about both axes.

When doing video processing on Win32 systems, you will find yourself using this function often to switch between image formats with their origins at the upper-left and lower-left of the image.

cv::gemm()

```
void cv::gemm(
  cv::InputArray  src1,                   // First input array
  cv::InputArray  src2,                   // Second input array
  double          alpha,                  // Weight for 'src1' * 'src2' product
  cv::InputArray  src3,                   // Third (offset) input array
  double          beta,                   // Weight for 'src3' array
  cv::OutputArray dst,                    // Results array
  int             flags = 0               // Use to transpose source arrays
```

[31] This is, for example, always the case for covariance matrices. See cvCalcCovarMatrixSee cv::calcCovarMatrixcvCalcCovarMatrix().

```
);
```

Generalized matrix multiplication (GEMM) in OpenCV is performed by `cv::gemm()`, which performs matrix multiplication, multiplication by a transpose, scaled multiplication, and so on. In its most general form, `cv::gemm()` computes the following:

$$\boldsymbol{D} = \dagger\alpha \bullet op(\mathrm{src}_1) * \dagger \mathrm{op}(src_2) + \dagger\beta \bullet \mathrm{op}(\mathrm{src}_3)$$

where `src1`, `src2`, and `src3` are matrices, α and β are numerical coefficients, and *op()* is an optional transposition of the matrix enclosed. The transpositions are controlled by the optional argument `flags`, which may be 0 or any combination (by means of Boolean OR) of `cv::GEMM_1_T`, `cv::GEMM_2_T`, and `cv::GEMM_3_T` (with each flag indicating a transposition of the corresponding matrix).

All matrices must be of the appropriate size for the (matrix) multiplication, and all should be of floating-point types. The `cv::gemm()` function also supports two-channel matrices that will be treated as two components of a single complex number.

A similar result can be achieved using the matrix algebra operators. For example:

```
cv::gemm(src1,src2,alpha,src3,bets,dst,cv::GEMM_1_T|cv::GEMM_3_T)
```

would be equivalent to:

```
dst = alpha * src1.T() * src2 + beta * src3.T()
```

`cv::getConvertElem()` and `cv::getConvertScaleElem()`

```
cv::convertData cv::getConvertElem(       // Returns a conversion function (below)
  int fromType,                           // Input pixel type (e.g., cv::U8)
  int toType                              // Output pixel type (e.g., cv::F32)

);
cv::convertScaleData cv::getConvertScaleElem(  // Returns a conversion function
  int fromType,                           // Input pixel type (e.g., cv::U8)
  int toType                              // Output pixel type (e.g., cv::F32)
);

// Conversion functions are of these forms:
//
typedef void (*ConvertData)(
  const void* from,                       // Pointer to the input pixel location
  void*       to,                         // Pointer to the result pixel location
  int         cn                          // number of channels
);
typedef void (*ConvertScaleData)(
  const void* from,                       // Pointer to the input pixel location
  void*       to,                         // Pointer to the result pixel location
  int         cn,                         // number of channels
  double      alpha,                      // scale factor
  double      beta                        // offset factor
);
```

The functions `cv::getConvertElem()` and `cv::getConvertScaleElem()` return function pointers to the functions that are used for specific type conversions in OpenCV. The function returned by `cv::getConvertElem()` is defined (via `typedef`) to the type `cv::ConvertData`, which can be passed a pointer to two data areas and a number of "channels." The number of channels is given by the argument `cn` of the conversion function, which is really the number of contiguous-in-memory objects of `fromType` to convert. This means that if you wanted, you could convert an entire (contiguous in memory) array by simply setting the number of channels equal to the total number of elements in the array.

Both `cv::getConvertElem()` and `cv::getConvertScaleElem()` take as arguments two types: `fromType` and `toType`. These types are specified using the integer constants (`cv::F32`, etc.).

In the case of `cv::getConvertScaleElem()`, the returned function takes two additional arguments, `alpha` and `beta`. These values are used by the converter function to rescale (`alpha`) and offset (`beta`) the input value before conversion to the desired type.

cv::idct()

```
void cv::idct(
  cv::InputArray  src,                    // Input array
  cv::OutputArray dst,                    // Result array
  int             flags,                  // for row-by-row
);
```

`cv::idct()` is just a convenient shorthand for the inverse discrete cosine transform. A call to `cv::idct()` is exactly equivalent to a call to `cv::dct()` with the arguments:

```
cv::dct( src, dst, flags | cv::DCT_INVERSE );
```

cv::idft()

```
void cv::idft(
  cv::InputArray  src,                    // Input array
  cv::OutputArray dst,                    // Result array
  int             flags       = 0,        // for row-by-row, etc.
  int             nonzeroRows = 0         // assume only this many entries are nonzero
);
```

`cv::idft()` is just a convenient shorthand for the inverse discrete Fourier transform. A call to `cv::idft()` is exactly equivalent to a call to `cv::dft()` with the arguments:

```
cv::dft( src, dst, flags | cv::DCT_INVERSE, outputRows );
```

> It is noteworthy that neither `cv::dft()` nor `cv::idft()` scales the output by default. So you will probably want to call `cv::idft()` with the `cv::DFT_SCALE` argument, that way, the transform and its "inverse" will be true inverse operations.

cv::inRange()

```
void cv::inRange(
  cv::InputArray  src,                    // Input Array
  cv::InputArray  upperb,                 // Array of upper bounds (inclusive)
  cv::InputArray  lowerb,                 // Array of lower bounds (inclusive)
  cv::OutputArray dst                     // Result array, cv::U8C1 type
);
```

$$dst_i = lowerb_i \le src_i \le upperb_i$$

When applied to a one-dimensional array, each element of `src` is checked against the corresponding elements of `upperb` and `lowerb`. The corresponding element of `dst` is set to `255` if the element in `src` is between the values given by `upperb` and `lowerb`;; otherwise, it is set to `0`.

However, in the case of multichannel arrays for `src`, `upperb`, and `lowerb`, however, the output is still a single channel. The output value for element i will be set to `255` if and only if the values for the corresponding entry in `src` all lay inside of the intervals implied for the corresponding channel in `upperb` and `lowerb`. In this sense, `upperb` and `lowerb` define an n-dimensional hypercube for each pixel and the corresponding value in `dst` is only set to true (`255`) if the pixel in `src` lies inside that hypercube.

cv::invert()

```
double cv::invert(                      // Return 0 if 'src' is singular
  cv::InputArray  src,                  // Input Array, m-by-n
  cv::OutputArray dst                   // Result array, n-by-m
  int             method = cv::DECOMP_LU  // Method for computing (pseudo) inverse
```

```
);
```

`cv::invert()` inverts the matrix in `src` and places the result in `dst`. The input array must be a floating-point type, and the result array will be of the same type. Because `cv::invert()` includes the possibility of computing pseudo-inverses, the input array need not be square. If the input array is n-by-m, then the result array will be m-by-n. This function supports several methods of computing the inverse matrix (see Table 3-24), but the default is Gaussian elimination. The return value depends on the method used.

Table 3-24: Possible values of method argument to `cv::invert()`

Value of method argument	Meaning
`cv::DECOMP_LU`	Gaussian elimination (LU decomposition)
`cv::DECOMP_SVD`	Singular value decomposition (SVD)
`cv::DECOMP_CHOLESKY`	Only for symmetric positive matrices

In the case of Gaussian elimination (`cv::DECOMP_LU`), the determinant of `src` is returned when the function is complete. If the determinant is `0`, inversion failed and the array `dst` is set to `0`s.

In the case of `cv::DECOMP_SVD`, the return value is the inverse condition number for the matrix (the ratio of the smallest to the largest eigenvalues). If the matrix `src` is singular, then `cv::invert()` in SVD mode will compute the pseudo-inverse. The other two methods (LU and Cholesky decomposition) require the source matrix to be square, nonsingular and positive.

`cv::log()`

```
void cv::log(
  cv::InputArray  src,
  cv::OutputArray dst
);
```

$$dst_i = \begin{cases} \log src_i & src_i \neq 0 \\ -C & else \end{cases}$$

`cv::log()` computes the natural log of the elements in `src1` and puts the results in `dst`. Source pixels that are less than or equal to zero are marked with destination pixels set to a large negative value.

`cv::LUT()`

```
void cv::LUT(
  cv::InputArray  src,
  cv::InputArray  lut,
  cv::OutputArray dst
);
```

$$dst_i = \text{lut}\left(src_i\right)$$

The function `cv::LUT()` performs a "lookup table transform" on the input in `src`. `cv::LUT()` requires the source array `src` to be 8-bit index values. The `lut` array holds the lookup table. This lookup table array should have exactly 256 elements, and may have either a single channel or, in the case of a multichannel `src` array, the same number of channels as the source array. The function `cv::LUT()` then fills the destination array `dst` with values taken from the lookup table `lut` using the corresponding value from `src` as an index into that table.

In the case where the values in `src` are signed 8-bit numbers, they are automatically offset by +128 so that their range will index the lookup table in a meaningful way. If the lookup table is multichannel (and the indices are as well), then the value in `src` is used as a multidimensional index into `lut`, and the result array `dst` will be single channel. If `lut` is one-dimensional, then the result array will be multichannel,

with each channel being separately computed from the corresponding index from `src` and the one-dimensional lookup table.

`cv::magnitude()`

```
void cv::magnitude(
  cv::InputArray  x,
  cv::InputArray  y,
  cv::OutputArray dst
);
```

$$dst_i = \sqrt{x_i^2 + y_i^2}$$

`cv::magnitude()` essentially computes the radial part of a Cartesian-to-polar conversion on a two-dimensional vector field. In the case of `cv::magnitude()`, this vector field is expected to be in the form of two separate single channel arrays. These two input array should have the same size. (If you have a single two-channel array, `cv::split()` will give you separate channels.) Each element in `dst` is computed from the corresponding elements of `x` and `y` as the Euclidian norm of the two (i.e., the square root of the sum of the squares of the corresponding values).

`cv::Mahalanobis()`

```
cv::Size cv::mahalanobis(
    cv::InputArray  vec1,
    cv::InputArray  vec2,
    cv::OutputArray icovar
);
```

`cvMahalanobis()` computes the value:

$$r_{mahalonobis} = \sqrt{\left(\vec{x} - \vec{\mu}\right)^T \Sigma^{-1} \left(\vec{x} - \vec{\mu}\right)}$$

The *Mahalanobis distance* is defined as the vector distance measured between a point and the center of a Gaussian distribution; it is computed using the inverse covariance of that distribution as a metric. See Figure 3-2. Intuitively, this is analogous to the z-score in basic statistics, where the distance from the center of a distribution is measured in units of the variance of that distribution. The Mahalanobis distance is just a multivariable generalization of the same idea.

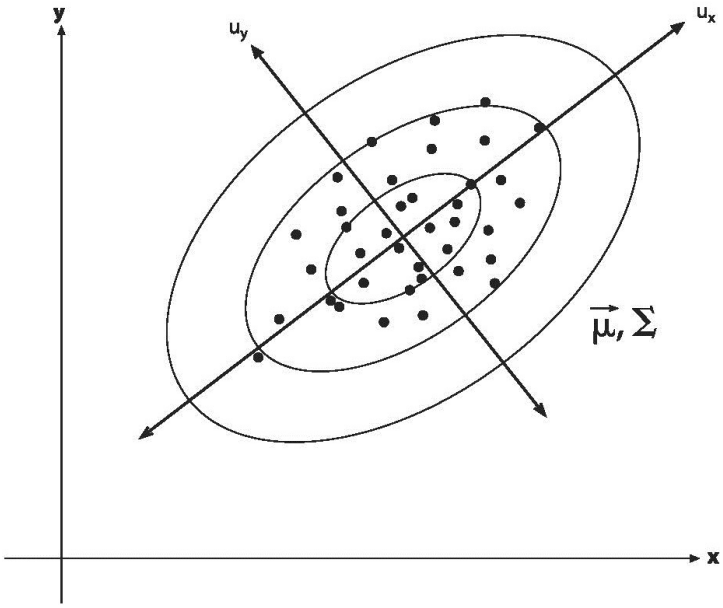


Figure 3-2: A distribution of points in two dimensions with superimposed ellipsoids representing Mahalanobis distances of 1.0, 2.0, and 3.0 from the distribution's mean

The vector `vec1` is presumed to be the point **x**, and the vector `vec2` is taken to be the distribution's mean.[32] That matrix `icovar` is the inverse covariance.

> This covariance matrix will usually have been computed with `cv::calcCovarMatrix()` (described previously) and then inverted with `cv::invert()`. It is good programming practice to use the `cv::DECOMP_SVD` method for this inversion because someday you will encounter a distribution for which one of the eigenvalues is `0`!

`cv::max()`

```
cv::MatExpr cv::max(
  const cv::Mat& src1,                  // First input Array
  const cv::Mat& src2                   // Second input array
);
MatExpr cv::max(                        // Return a matrix expression, not a matrix
  const cv::Mat&  src1,                 // First input array (first position)
  double          value                 // Scalar in second position
);
MatExpr cv::max(                        // Return a matrix expression, not a matrix
  double          value,                // Scalar in first position
  const cv::Mat&  src1                  // Input Array (second position)
);

void cv::max(
  cv::InputArray  src1,                 // First input Array
  cv::InputArray  src2,                 // Second input Array
  cv::OutputArray dst                   // Result array
);
void cv::max(
  const Mat&      src1,                 // First input Array
  const Mat&      src2,                 // Second input Array
  Mat&            dst                   // Result array
);
void cv::max(
  const Mat&      src1,                 // Input Array
  double          value,                // Scalar input
  Mat&            dst                   // Result array
);
```

$$dst_i = \max\left(\dagger \mathrm{src}_{1,\mathrm{i}}, \dagger src_{2,i}\right)$$

`cv::max()` computes the maximum value of each corresponding pair of pixels in the arrays `src1` and `src2`. It has two basic forms: those that return a matrix expression and those that compute a result and put it someplace you have indicated. In the three-argument form, in the case where one of the operands is a `cv::Scalar`, comparison with a multichannel array is done on a per-channel basis with the appropriate component of the `cv::Scalar`.

`cv::mean()`

```
cv::Scalar cv::mean(
  cv::InputArray  src,
```

[32] Actually, the Mahalanobis distance is more generally defined as the distance between any two vectors; in any case, the vector `vec2` is subtracted from the vector `vec1`. Neither is there any fundamental connection between `mat` in `cvMahalanobiscv::Mahalanobis()` and the inverse covariance; any metric can be imposed here as appropriate.

```
  cv::InputArray  mask = cv::noArray(),   // Optional mask, compute only where nonzero
);
```

$$N = \sum_{I,\ mask_I \neq 0} 1$$

$$mean_c = \frac{1}{N} \sum_{I,\ mask_I \neq 0} src_I$$

The function `cv::mean()` computes average value of all of the pixels in the input array `src` that are not masked out. The result is computed on a per-channel basis if `src` is multichannel.

`cv::meanStdDev()`

```
void cv::meanStdDev(
  cv::InputArray  src,
  cv::OutputArray mean,
  cv::OutputArray stddev,
  cv::InputArray  mask = cv::noArray(),   // Optional mask, compute only where nonzero

);
```

$$N = \sum_{I,\ mask_I \neq 0} 1$$

$$mean_c = \frac{1}{N} \sum_{I,\ mask_I \neq 0} src_I$$

$$stddev_c = \sqrt{\sum_{I,\ mask \neq 0} \left(src_{c,I} - mean_c \right)^2}$$

The function `cv::meanStdDev()` computes the average value of the pixels in the input array `src` not masked out, as well as their standard deviation. The mean and standard deviation are computed on a per-channel basis if `src` is multichannel.

> It is important to know that the standard deviation computed here is not the same as the covariance matrix. In fact, the standard deviation computed here is only the diagonal elements of the full covariance matrix. If you want to compute the full covariance matrix, you will have to use `cv::calcCovarMatrix()`.

`cv::merge()`

```
void cv::merge(
  const cv::Mat*  mv,                   // C-style array of arrays
  size_t          count,                // Number of arrays pointed to by 'mv'
  cv::OutputArray dst                   // Result array, contains all channels in 'mv'
);
void merge(
  const vector<cv::Mat>& mv,            // STL-style array of arrays
  cv::OutputArray dst                   // Result array, contains all channels in 'mv'
);
```

`cv::merge()` is the inverse operation of `cv::split()`. The arrays contained in `mv` are combined into the output array `dst`. In the case in which `mv` is a pointer to a C-style array of `cv::Mat` objects, the additional size parameter `count` must also be supplied.

cv::min()

```
cv::MatExpr cv::min(                    // Return a matrix expression, not a matrix
  const cv::Mat&  src1,                 // First input array
  const cv::Mat&  src2                  // Second input array
);
MatExpr cv::min(                        // Return a matrix expression, not a matrix
  const cv::Mat&  src1,                 // First input array (first position)
  double          value                 // Scalar in second position
);
MatExpr cv::min(                        // Return a matrix expression, not a matrix
  double          value,                // Scalar in first position
  const cv::Mat&  src1                  // Input Array (second position)
);

void cv::min(
  cv::InputArray  src1,                 // First input Array
  cv::InputArray  src2,                 // Second input Array
  cv::OutputArray dst                   // Result array
);
void cv::min(
  const Mat&      src1,                 // First input Array
  const Mat&      src2,                 // Second input Array
  Mat&            dst                   // Result array
);
void cv::min(
  const Mat&      src1,                 // Input Array
  double          value,                // Scalar input
  Mat&            dst                   // Result array
);
```

$$dst_i = \min\left(\dagger\mathrm{src}_{1,i}, \dagger src_{2,i}\right)$$

`cv::min()` computes the minimum value of each corresponding pair of pixels in the arrays `src1` and `src2` (or one source matrix and a single value). Note that the variants of `cv::min()` that return a value or return a matrix expression that can then be manipulated by OpenCV's matrix expression machinery.

In the three-argument form, in the case where one of the operands is a `cv::Scalar`, comparison with a multichannel array is done on a per-channel basis with the appropriate component of the `cv::Scalar`.

cv::minMaxIdx()

```
void cv::minMaxLoc(
  cv::InputArray src,                   // Input array, 1-dimensional only
  double*        minVal,                // min value goes here (in not NULL)
  double*        maxVal,                // min value goes here (in not NULL)
  cv::Point*     minLoc,                // location of min goes here (if not NULL)
  cv::Point*     maxLoc,                // location of max goes here (if not NULL)
  cv::InputArray mask = cv::noArray()   // search only non-zero values (if not NULL)

void cv::minMaxLoc(
  const SparseMat& src,                 // Input sparse array
  double*        minVal,                // min value goes here (in not NULL)
  double*        maxVal,                // min value goes here (in not NULL)
  cv::Point*     minIdx,                // C-style array, indices of min location
  cv::Point*     maxIdx,                // C-style array, indices of min location
);
```

This routine finds the minimal and maximal values in the array `src` and (optionally) returns their locations. The computed minimum and maximum values are placed in `minVal` and `maxVal`. Optionally, the locations of those extrema can also be returned, but only if the `src` array is two-dimensional. These locations will be written to the addresses given by `minLoc` and `maxLoc` (provided that these arguments

are non-NULL). Because these locations are of type cv::Point, this form of the function should only be used on two-dimensional arrays (i.e., matrices or images).

cv::minMaxLoc()

```
void cv::minMaxLoc(
  cv::InputArray src,                    // Input array
  double*        minVal,                 // min value goes here (in not NULL)
  double*        maxVal,                 // min value goes here (in not NULL)
  cv::Point*     minLoc,                 // location of min goes here (if not NULL)
  cv::Point*     maxLoc,                 // location of max goes here (if not NULL)
  cv::InputArray mask = cv::noArray()    // search only non-zero values (if present)

void cv::minMaxLoc(
  const SparseMat& src,                  // Input sparse array
  double*        minVal,                 // min value goes here (in not NULL)
  double*        maxVal,                 // min value goes here (in not NULL)
  cv::Point*     minIdx,                 // C-style array, indices of min location
  cv::Point*     maxIdx,                 // C-style array, indices of min location
);
```

This routine finds the minimal and maximal values in the array src and (optionally) returns their locations. The computed minimum and maximum values are placed in minVal and maxVal. Optionally, the locations of those extrema can also be returned, but only if the src array is two-dimensional. These locations will be written to the addresses given by minLoc and maxLoc (provided that these arguments are non-NULL). Because these locations are of type cv::Point, this form of the function should only be used on two-dimensional arrays (i.e., matrices or images).

cv::minMaxLoc() can also be called with a cv::SparseMat for the src array. In this case, the array can be of any number of dimensions and the minimum and maximum will be computed and their location returned. In this case, the locations of the extrema will be returned and placed in the C-style arrays minIdx and maxIdx. Both of those arrays, if provided, should have the same number of elements as the number of dimensions in the src array. In the case of cv::SparseMat, the minimum and maximum are only computed for what are generally referred to as nonzero elements in the source code. However, it is important to note that this terminology is slightly misleading, as what is really meant is: *elements that exist* in the sparse matrix representation in memory. In fact, there may, as a result of how the sparse matrix came into being and what has been done with it in the past, be elements that *exist* and are also zero. Such elements will be included in the computation of the minimum and maximum.

When working with multichannel arrays, there are several options. Natively, cv::minMaxLoc() does not support multichannel input. Primarily this is because this operation is ambiguous.

> If you want the minimum and maximum across all channels, you can use cv::reshape() to reshape the multichannel array into one giant single-channel array. If you would like the minimum and maximum for each channel separately, you can use cv::split() or cv::mixChannels() to separate the channels out and analyze them separately.

In both forms of cv::minMaxLoc, the arguments for the minimum or maximum value or location may be set to NULL, which turns off the computation for that argument.

cv::mixChannels()

```
void cv::mixChannels(
  const cv::Mat*         srcv,           // C-style array of matrices
  int                    nsrc,           // Number of elements in 'srcv'
  cv::Mat*               dstv,           // C-style array of target matrices
  int                    ndst,           // Number of elements in 'dstv'
  const int*             fromTo,         // C-style array of pairs, …from,to…
  size_t                 n_pairs         // Number of pairs in 'fromTo'
```

```
);

void cv::mixChannels(
  const vector<cv::Mat>& srcv,          // STL-style vector of matrices
  vector<cv::Mat>&       dstv,          // STL-style vector of target matrices
  const int*             fromTo,        // C-style array of pairs, …from,to…
  size_t                 n_pairs        // Number of pairs in 'fromTo'
);
```

There are many operations in OpenCV that are special cases of the general problem of rearranging channels from one or more images in the input, and sorting them into particular channels in one or more images in the output. Functions like `cv::split()`, `cv::merge()`, and (at least some cases of) `cv::cvtColor()` all make use of such functionality. Those methods do what they need to do by calling the much more general `cv::mixChannels()`. This function allows you to supply multiple arrays, each with potentially multiple channels, for the input, and the same for the output, and to map the channels from the input arrays into the channels in the output arrays in any manner you choose.

The input and output arrays can either be specified as C-style arrays of `cv::Mat` objects with an accompanying integer indicating the number of `cv::Mats`, or as an STL `vector<>` of `cv::Mat` objects. Output arrays must be pre-allocated with their size and number of dimensions matching those of the input arrays.

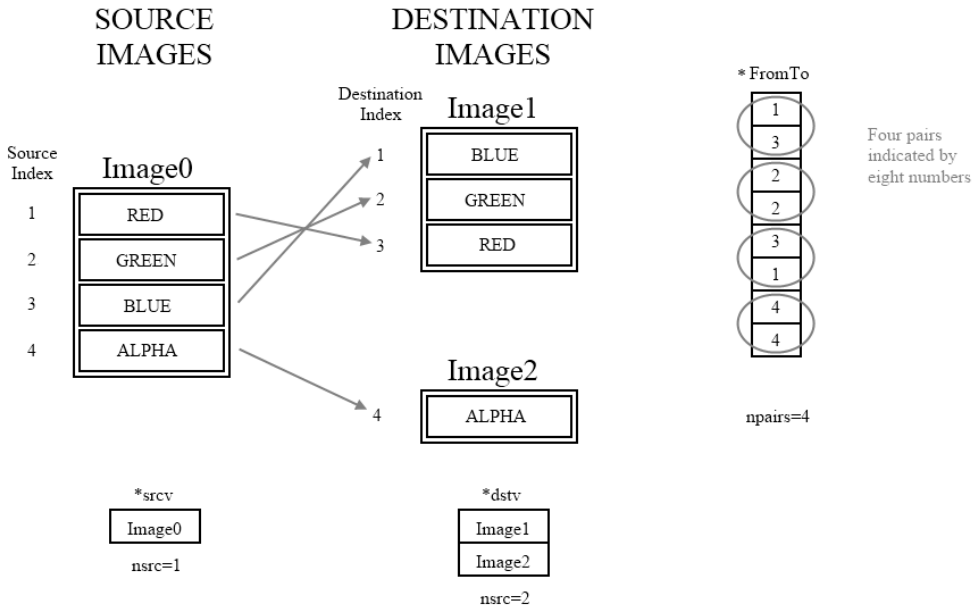


Figure 3-3: A single four-channel RGBA image is converted to one BGR and one Alpha-only image.

The mapping is controlled by the C-style integer array `fromTo`. This array can contain any number of integer pairs in sequence, with each pair indicating with its first value the source channel and with its second value the destination channel to which that should be copied. The channels are sequentially numbered starting at zero for the first image, then through the second image, and so on. (Figure 3-3). The total number of pairs is supplied by the argument `n_pairs`.

> Unlike most other functions in the post-version 2.1 library, `cv::mixChannels()` *does not* allocate the output arrays. They must be pre-allocated and have the same size and dimensionality as the input arrays.

cv::mulSpectrums()

```
doublevoid cv::mulSpectrums(
    cv::InputArray  arr1,                // First input array
    cv::InputArray  arr2,                // Second input array, same size as 'arr1'
    cv::OutputArray dst,                 // Result array, same size as 'arr1'
    int             flags,               // used to indicate that rows are independent
    bool            conj  = false        // If true, conjugate arr2 first
);
```

In many operations involving spectra (i.e., the results from `cv::dft()` or `cv::idft()`), one wishes to do a per-element multiplication that respects the packing of the spectra (real arrays), or their nature as complex variables. (See the description of `cv::dft()` for more details.) The input arrays may be one- or two-dimensional, with the second the same size and type as the first. If the input array is two-dimensional, it may either be taken to be a true two-dimensional spectrum, or an array of one-dimensional spectra (one per row). In the latter case, flags should be set to `cv::DFT_ROWS`; otherwise it can be set to `0`.

When the two arrays are complex, they are simply multiplied on an element-wise basis, but `cv::mulSpectrums()` provides an option to conjugate the second array elements before multiplication. For example, you would use this option to perform correlation (using the Fourier transform), but for convolution, you would use `conj=false`.

cv::multiply()

```
void cv::multiply(
  cv::InputArray  src1,                  // First input array
  cv::InputArray  src2,                  // Second input array
  cv::OutputArray dst,                   // Result array
  double          scale = 1.0,           // overall scale factor
  int             dtype = -1             // Output type for result array
);
```

$$dst_i = saturate\left(scale * † src1_i * src2_i\right)$$

`cv::multiply()` is a simple multiplication function; it multiplies the elements in `src1` by the corresponding elements in `src2` and puts the results in `dst`.

cv::mulTransposed()

```
void cv::mulTransposed(
  cv::InputArray  src1,                  // Input matrix
  cv::OutputArray dst,                   // Result array
  bool            aTa,                   // If true, transpose then multiply
  cv::InputArray  delta = cv::noArray(),  // subtract from 'src1' before multiplication
  double          scale = 1.0,           // overall scale factor
  int             dtype = -1             // Output type for result array

)
```

$$dst = \begin{cases} scale * \left(src - delta\right)^T \left(src - delta\right) & aTa = true \\ scale * \left(src - delta\right)\left(src - delta\right)^T & aTa = false \end{cases}$$

`cv::mulTransposed()` is used to compute the product of a matrix and its own transpose—useful, for example, in computing covariance. The matrix `src` should be two-dimensional and single-channel, but unlike `cv::GEMM()`, it is not restricted to the floating-point types. The result matrix will be the same type as the source matrix unless specified otherwise by `dtype`. If `dtype` is not negative (default), it should be either `cv::F32` or `cv::F64`; the output array `dst` will then be of the indicated type.

If a second input matrix `delta` is provided, that matrix will be subtracted from `src` before the multiplication. If no matrix is provided (i.e., `delta=cv::noArray()`), then no subtraction is done. The

array `delta` need not be the same size as `src`; if `delta` is smaller than `src`, `delta` is repeated (also called tiling, see `cv::repeat()`) in order to produce an array whose size matches the size of `src`. The argument `scale` is applied to the matrix after the multiplication is done. Finally, the argument `aTa` is used to select either the multiplication in which the transposed version of `src` is multiplied from the left (`aTa=true`) or from the right (`aTa=false`).

`cv::norm()`

```
double cv::norm(                          // Return computed norm in double precision
  cv::InputArray src1,                    // Input matrix
  int            normType = cv::NORM_L2,  // Type of norm to compute
  cv::InputArray mask     = cv::noArray() // include only non-zero values (if present)
);
double cv::norm(                          // Return computed norm of difference
  cv::InputArray src1,                    // Input matrix
  cv::InputArray src2,                    // Second input matrix
  int            normType = cv::NORM_L2,  // Type of norm to compute
  cv::InputArray mask     = cv::noArray() // include only non-zero values (if present)
);
double cv::norm(
  const cv::SparseMat& src,               // Input sparse matrix
  int                  normType = cv::NORM_L2, // Type of norm to compute
);
```

$$src1_{\infty,L1,L2}$$

$$src1 - src2_{\infty,L1,L2}$$

This function is used to compute the norm of an array (see Table 3-25) or a variety of distance norms (Table 3-26) between two arrays if two arrays are provided (see Table 3-26). The norm of a `cv::SparseMat` can also be computed, in which case zero-entries are ignored in the computation of the norm.

Table 3-25: Norm computed by `cv:norm()` *for different values of* `normType` *when* `arr2=NULL`

normType	Result
`cv::NORM_INF`	$src1_{\infty} = \max_i abs(\text{src1}_i)$
`cv::NORM_L1`	$src1_{L1} = \sum_i abs(\text{src1}_i)$
`cv::NORM_L2`	$src1_{L2} = \sum_i \text{src1}_i^2 \sqrt{\sum_i \text{src1}_i^2}$

If the second array argument `src2` is non-`NULL`, then the norm computed is a difference norm—that is, something like the distance between the two arrays.[33] In the first three cases (shown in Table 3-26) the norm is absolute; in the latter three cases it is rescaled by the magnitude of the second array `src2`.

Table 3-26: Norm computed by `cv::norm()` *for different values of* `normType` *when* `arr2` *is non-*`NULL`

normType	Result
`cv::NORM_INF`	$src1 - src2_{\infty} = \max_i abs(src1_i - src2_i)$

[33] At least in the case of the L2 norm, there is an intuitive interpretation of the difference norm as a Euclidean distance in a space of dimension equal to the number of pixels in the images.

`cv::NORM_L1`	$src1 - src2_{L1} = \sum_i abs\left(src1_i - src2_i\right)$
`cv::NORM_L2`	$src1 - src2_{L2} = \sum_i \left(src1_i - src2_i\right)^2$
`cv::NORM_RELATIVE_INF`	$\frac{src1 - src2_{\infty}}{src2_{\infty}}$
`cv::NORM_RELATIVE_L1`	$\frac{src1 - src2_{L1}}{src2_{L1}}$
`cv::NORM_RELATIVE_L2`	$\frac{src1 - src2_{L2}}{src2_{L2}}$

In all cases, `src1` and `src2` must have the same size and number of channels. When there is more than one channel, the norm is computed over all of the channels together (i.e., the sums in Table and Table 3-26 are not only over *x* and *y* but also over the channels).

`cv::normalize()`

```
void cv::normalize(
  cv::InputArray  src1,                  // Input matrix
  cv::OutputArray dst,                   // Result matrix
  double          alpha    = 1,          // first parameter (see Table 3-27)
  double          beta     = 0,          // second parameter (see Table 3-27)
  int             normType = cv::NORM_L2, // Type of norm to compute
  int             dtype = -1             // Output type for result array
  cv::InputArray  mask     = cv::noArray()// include only non-zero values (if present)
);
void cv::normalize(
  const cv::SparseMat& src,              // Input sparse matrix
  cv::SparseMat&       dst,              // Result sparse matrix
  double               alpha    = 1,     // first parameter (see Table 3-27)
  int                  normType = cv::NORM_L2, // Type of norm to compute
);
```

$$dst_{\infty,L1,L2} = \alpha$$

$$\min\left(dst\right) = \alpha, \dagger\dagger\dagger\dagger \max\left(dst\right) = \beta$$

As with so many OpenCV functions, `cv::normalize()` does more than it might at first appear. Depending on the value of `normType`, image `src` is normalized or otherwise mapped into a particular range in `dst`. The array `dst` will be the same size as `src`, and will have the same data type, unless the `rtype` argument is used. Optionally, `rtype` can be set to one of the OpenCV fundamental types (`cv::F32`, etc.) and the output array will be of that type. The exact meaning of this operation is dependent on the `normType` argument. The possible values of `normType` are shown in Table 3-27.

Table 3-27: Possible values of `normType` *argument to* `cv::normalize()`

norm_type	**Result**
`cv::NORM_INF`	$dst_{\infty} = \max_i abs\left(dst_i\right) = \alpha$

`cv::NORM_L1`

$$dst_{L1} = \sum_i abs\left(dst_i\right) = \alpha$$

`cv::NORM_L2`

$$dst_{L2} = \sum_i dst_i^2 = \sqrt{\sum_i dst_i^2} = \alpha$$

`cv::NORM_MINMAX`

Map into range [α , β]

In the case of the infinity norm, the array `src` is rescaled such that the magnitude of the absolute value of the largest entry is equal to `alpha`. In the case of the L1 or L2 norm, the array is rescaled so that the norm equals the value of `alpha`. If `normType` is set to `cv::MINMAX`, then the values of the array are rescaled and translated so that they are linearly mapped into the interval between `alpha` and `beta` (inclusive).

As before, if `mask` is non-`NULL` then only those pixels corresponding to nonzero values of the mask image will contribute to the computation of the norm—and only those pixels will be altered by `cv::normalize()`. Note that if the operation `dtype=cv::MINMAX` is used, the source array may not be `cv::SparseMat`. The reason for this is that the `cv::MIN_MAX` operation can apply an overall offset, and this would affect the sparsity of the array (specifically, a sparse array would become non-sparse as all of the zero elements became non-zero as a result of this operation).

`cv::perspectiveTransform()`

```
void cv::perspectiveTransform(
  cv::InputArray  src,                    // Input array, 2 or 3 channels
  cv::OutputArray dst,                    // Result array, size, type, as src1
  cv::InputArray  mtx                     // 3-by-3 or 4-by-4 transoform matrix
);
```

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix} \rightarrow \begin{bmatrix} x'/w' \\ y'/w' \\ z'/w' \end{bmatrix}$$

$$\begin{bmatrix} x' \\ y' \\ z' \\ w' \end{bmatrix} \rightarrow [mtx] \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

The `cv::perspectiveTransform()` function performs a plane-plane projective transform of a list of points (not pixels). The input array should be a two- or three-channel array, and the matrix `mtx` should be *3*-by-*3* or *4*-by-*4*, respectively, in the two cases. `cv::perspectiveTransform()` thus transforms each element of `src` by first regarding it as a vector of length `src.channels() + 1`, with the additional dimension (the projective dimension) set initially to `1.0`. This is also known as homogeneous coordinates. Each extended vector is then multiplied by `mtx` and the result is rescaled by the value of the (new) projective coordinate[34] (which is then thrown away as it is always `1.0` after this operation).

> Note again that this routine is for transforming a list of points, not an image as such. If you want to apply a perspective transform to an image, you are actually asking not to

[34] Technically, it is possible that after multiplying by `mtx`, the value of w' will be zero, corresponding to points projected to infinity. In this case, rather than dividing by zero, a value of `0` is assigned to the ratio.

transform the individual pixels, but rather to move them from one place in the image to another. This is the job of `cv::warpPerspective()`.

If you want to solve the inverse problem to find the most probable perspective transformation given many pairs of corresponding points, use `cv::getPerspectiveTransform()` or `cv::findHomography()`.

cv::phase()

```
void cv::phase(
  cv::InputArray  x,                    // Input array of x-components
  cv::InputArray  y,                    // Input array of y-components
  cv::OutputArray dst,                  // Output array of angles (radians)
);
```

$$dst_i = \text{atan2}\left(y_i, x_i\right)$$

`cv::phase()` computes the azimuthal (angle) part of a Cartesian-to-polar conversion on a two-dimensional vector field. This vector field is expected to be in the form of two separate single--channel arrays. These two input arrays should, of course, be of the same size. (If you happen to have a single two--channel array, a quick call to `cv::split()` will do just what you need.) Each element in `dst` is then computed from the corresponding elements of `x` and `y` as the arctangent of the ratio of the two.

cv::polarToCart()

```
void cv::polarToCart(
  cv::InputArray  magnitude,            // Input array of magnitudes
  cv::InputArray  angle,                // Input array of angles
  cv::OutputArray x,                    // Output array of x-components
  cv::OutputArray y,                    // Output array of y-components
  bool            angleInDegrees = false  // degrees (if true) radians (if false)
);
```

$$x_i = magnitude_i * \cos\left(angle_i\right)$$

$$y_i = magnitude_i * \sin\left(angle_i\right)$$

`cv::polarToCart()` computes a vector field in Cartesian $\left(x, y\right)$ coordinates from polar coordinates. The input is in two arrays, `magnitude` and `angle`, of the same size and type, specifying the magnitude and angle of the field at every point. The output is similarly two arrays that will be of the same size and type as the inputs, and which will contain the *x* and *y* projections of the vector at each point. The additional flag `angleInDegrees` will cause the angle array to be interpreted as angles in degrees, rather than in radians.

cv::pow()

```
void cv::pow(
  cv::InputArray  src,                  // Input array
  double          p,                    // power for exponentiation
  cv::OutputArray dst                   // Result array
);
```

$$dst_i = \begin{cases} src_i^p & p \in \mathbb{Z} \\ \left|src_i\right|^p & else \end{cases}$$

The function `cv::pow()` computes the element-wise exponentiation of an array by a given power `p`. In the case in which `p` is an integer, the power is computed directly. For non-integer `p`, the absolute value of

the source value is computed first, and then raised to the power `p` (so only real values are returned). For some special values of `p`, such as integer values, or $\pm 1/2$, special algorithms are used resulting in faster computation.

`cv::randu()`

```
template<typename _Tp> _Tp randu();       // Return random number of specific type

void cv::randu(
  cv::InputOutArray mtx,                  // All values will be randomized
  cv::InputArray    low,                  // minimum, 1-by-1 (Nc=1,4), or 1-by-4 (Nc=1)
  cv::InputArray    high                  // maximum, 1-by-1 (Nc=1,4), or 1-by-4 (Nc=1)
);
```

$$mtx_i \in \left[low_i, high_i \right)$$

There are two ways to call `cv::randu()`. The first method is to call the template form of `randu<>()`, which will return a random value of the appropriate type. Random numbers generated in this way are uniformly distributed[35] in the range from zero to the maximum value available for that type (for integers), and in the interval from 0.0 to 1.0 (not inclusive of 1.0) for floating-point types. This template form generates only single numbers[36].

The second way to call `cv::randu()` is to provide a matrix `mtx` that you wish to have filled with values, and two additional arrays that specify the minimum and maximum values for the range from which you would like a random number drawn for each particular array element. These two additional values, `low` and `high`, should be 1-*by*-1 with 1 or 4 channels, or 1-*by*-4 with a single channel; they may also be of type `cv::Scalar`. In any case, they are not the size of `mtx`, but rather the size of individual entries in `mtx`.

The array `mtx` is both an input and an output, in the sense that you must allocate the matrix so that `cv::randu()` will know the number of random values you need, and how they are to be arranged in terms of rows, columns, and channels.

`cv::randn()`

```
void cv::randn(
  cv::InputOutArray mtx,                  // All values will be randomized
  cv::InputArray    mean,                 // means, array is in channel-space
  cv::InputArray    stddev                // standard deviations, channel-space
);
```

$$mtx_i \sim N\left(mean_i, stddev_i \right)$$

The function `cv::randn()` fills a matrix `mtx` with random normally distributed values[37]. The parameters from which these values are drawn are taken from two additional arrays (`mean` and `stddev`) that specify the mean and standard deviation for the distribution from which you would like a random number drawn for each particular array element.

---

[35] Uniform distribution random numbers are generated using the Multiply-With-Carry algorithm [G. Marsaglia].

[36] In particular, this means that if you call the template form with a vector argument, such as: `cv::randu<Vec4f>`, the return value, though it will be of vector type, will be all zeros except for the first element.

[37] Gaussian-distribution random numbers are generated using the Ziggurat algorithm [G. Marsaglia, W. W. Tsang].

As with the array form of `cv::randu()`, every element of `mtx` is computed separately, and the arrays `mean` and `stddev` are in the channel-space for individual entries of `mtx`. Thus, if `mtx` were four channels, then `mean` and `stddev` would be 1-*by*-4 or 1-*by*-1 with 4 channels (or equivalently of type `cv::Scalar`).[38]

`cv::randShuffle()`

```
void cv::randShuffle(
  cv::InputOutArray mtx,                  // All values will be shuffled
  double            iterFactor = 1,       // Number of times to repeat shuffle
  cv::RNG*          rng        = NULL     // your own generator, if you like
);
```

`cv::randShuffle()` attempts to randomize the entries in a one-dimensional array by selecting random pairs of elements and interchanging their position. The number of such *swaps* is equal to the size of the array `mtx` multiplied by the optional factor `iterFactor`. Optionally, a random number generator can be supplied (for more on this, see "Random Number Generator (`cv::RNG`)"). If none is supplied, the default random number generator `theRNG()` will be used automatically.

`cv::reduce()`

```
void cv::reduce(
  cv::InputArray  src,                    // Input array, n-by-m must be 2-dimensional
  cv::OutputArray vec,                    // Output array, 1-by-m or n-by-1
  int             dim,                    // Reduction direction (1) to row, (0) to col
  int             reduceOp = cv::REDUCE_SUM,  // Reduce operation (see Table 3-)
  int             dtype = -1              // Output type for result array
);
```

Reduction is the systematic transformation of the input matrix `src` into a vector `vec` by applying some combination rule `reduceOp` on each row (or column) and its neighbor until only one row (or column) remains (see Table 3-28).[39] The argument `dim` controls how the reduction is done, as summarized in Table 3-29.

Table 3-28: Argument `reduceOp` in `cv::reduce()` selects the reduction operator

Value of op	Result
`cv::REDUCE_SUM`	Compute sum across vectors
`cv::REDUCE_AVG`	Compute average across vectors
`cv::REDUCE_MAX`	Compute maximum across vectors
`cv::REDUCE_MIN`	Compute minimum across vectors

Table 3-29: Argument `dim` in `cv::reduce()` controls the direction of the reduction

Value of dim	Result
`1`	Collapse to a single row
`0`	Collapse to a single column

`cv::reduce()` supports multichannel arrays of any type. Using `dtype`, you can specify an alternative type for `dst`.

---

[38] Note that `stddev` is not a square matrix; correlated number generation is not supported by `cv::randn()`.

[39] Purists will note that averaging is not technically a proper *fold* in the sense implied here. OpenCV has a more practical view of reductions and so includes this useful operation in `cvReduce`.

Using the `dtype` argument to specify a higher-precision format for `dst` is particularly important for `cv::REDUCE_SUM` and `cv::REDUCE_AVG`, where overflows and summation problems are possible.

cv::repeat()

```
void cv::repeat(
  cv::InputArray  src,                  // Input 2-dimensional array
  int             nx,                   // Copies in x-direction
  int             ny,                   // Copies in y-direction
  cv::OutputArray dst                   // Resukt array
);
cv::Mat cv::repeat(                     // Return result array
  cv::InputArray  src,                  // Input 2-dimensional array
  int             nx,                   // Copies in x-direction
  int             ny                    // Copies in y-direction
);
```

$$dst_{i,j} = src_{i\%src.rows,\dagger j\%src.cols}\ \dagger$$

This function copies the contents of `src` into `dst`, repeating as many times as necessary to fill `dst`. In particular, `dst` can be of any size relative to `src`. It may be larger or smaller, and it need not have an integer relationship between any of its dimensions and the corresponding dimensions of `src`.

`cv::repeat()` has two calling conventions. The first is the old-style convention, in which the output array is passed as a reference to `cv::repeat()`. The second actually creates and returns a `cv::Mat`, and is much more convenient when working with matrix expressions.

cv::scaleAdd()

```
void cv::scaleAdd(
  cv::InputArray  src1,                 // First input array
  double          scale,                // Scale factor applied to first array
  cv::InputArray  src2,                 // Second input array
  cv::OutputArray dst,                  // Result array
);
```

$$dst_i = scale * src1_i + src2_i$$

`cv::scaleAdd()` is used to compute the sum of two arrays `src1` and `src2` with a scale factor `scale` applied to the first before the sum is done. The results are placed in the array `dst`.

The same result can be achieved with the matrix algebra operation:

```
dst = scale * src1 + src2;
```

cv::setIdentity()

```
void cv::setIdentity(
  cv::InputOutputArray dst,             // Array to reset values
  const cv::Scalar&    value = cv::Scalar(1.0)  // Value to apply to diagonal elems
);
```

$$dst_{i,j} = \begin{cases} value & i = j \\ 0 & else \end{cases}$$

`cv::setIdentity()` sets all elements of the array to 0 except for elements whose row and column are equal; those elements are set to 1 (or to `value` if provided). `cv::setIdentity()` supports all data types and does not require the array to be square.

This can also be done using the `eye()` member function of the `cv::Mat` class. The use of `eye()` is often more convenient when one is working with matrix expressions.

```
Mat A( 3, 3, cv::F32 );
cv::setIdentity( A, s );
C = A + B;
```

For some other arrays `B` and `C`, and some scalar `s`, this is equivalent to:

```
C = s * cv::Mat::eye( 3, 3, cv::F32 ) + B;
```

cv::solve()

```
int cv::solve(
  cv::InputArray  lhs,                    // Left-hand side of system, n-by-n
  cv::InputArray  rhs,                    // Right-hand side of system, n-by-1
  cv::OutputArray dst,                    // Results array, will be n-by-1
  int             method = cv::DECOMP_LU  // Method for solver
);
```

The function `cv::solve()` provides a fast way to solve linear systems based on `cv::invert()`. It computes the solution to

$$C = \text{argmin}_X \dagger A \bullet X - B$$

where **A** is a square matrix given by `lhs`, **B** is the vector `rhs`, and *C* is the solution computed by `cv::solve()` for the best vector **X** it could find. That best vector **X** is returned in `dst`. The actual method used to solve this system is determined by the value of the `method` argument (Table 3-30). Only floating-point data types are supported. The function returns an integer value where a nonzero return indicates that it could find a solution.

Table 3-30: Possible values of `method` argument to `cv::solve()`

Value of method argument	Meaning
`cv::DECOMP_LU`	Gaussian elimination (LU decomposition)
`cv::DECOMP_SVD`	Singular value decomposition (SVD)
`cv::DECOMP_CHOLESKY`	For symmetric positive matrices
`cv::DECOMP_EIG`	Eigenvalue decomposition, symmetric matrices only
`cv::DECOMP_QR`	QR Factorization
`cv::DECOMP_NORMAL`	Optional additional flag, indicates that the normal equations are to be solved instead)

The methods `cv::DECOMP_LU` and `cv::DECOMP_CHOLESKY` cannot be used on singular matrices. If a singular `src1` is provided, both methods will exit and return `0` (a `1` will be returned if `src1` is nonsingular). `cv::solve()` can be used to solve over-determined linear systems using either QR decomposition (`cv::DECOMP_QR`) or singular value decomposition (`cv::DECOMP_SVD`) methods to find the least-squares solution for the given system of equations. Both of these methods can be used in case the matrix `src1` is singular.

Though the first five arguments in Table 3-30 are mutually exclusive, the last option `cv::DECOMP_NORMAL` may be combined with any of the first five (e.g., by logical-or: `cv_DECOMP_LU | cv::DECOMP_NORMAL`). If provided, then `cv::solve()` will attempt to solve the *normal equations*: $src1^T \bullet src1 \bullet dst = src1^T \bullet src2$ instead of the usual system $src1 \bullet dst = src2$.

cv::solveCubic()

```
int cv::solveCubic(
  cv::InputArray  coeffs,
  cv::InputArray  roots
```

```
);
```

Given a cubic polynomial in the form of a three- or four-element vector `coeffs`, `cv::solveCubic()` will compute the real roots of that polynomial. If `coeffs` has four elements, the roots of the following polynomial are computed:

$$\text{coeffs}_0 x^3 + \text{coeffs}_1 x^2 + \text{coeffs}_2 x + \text{coeffs}_3 = 0$$

If `coeffs` has only three elements, the roots of the following polynomial are computed:

$$x^3 + \text{coeffs}_0 x^2 + \text{coeffs}_1 x + \text{coeffs}_2 = 0$$

The results are stored in the array `roots`, which will have either one or three elements, depending on how may real roots the polynomial has.

> A word of warning about `cv::solveCubic()` and `cv::solvePoly()`: the order of the coefficients in the seemingly analogous input arrays `coeffs` is opposite in the two routines. In `cv::solveCubic()`, the highest order coefficients come first, while in `cv::solvePoly()` the highest order coefficients come last.

`cv::solvePoly()`

```
int cv::solvePoly (
  cv::InputArray  coeffs,
  cv::OutputArray roots                  // n complex roots (2-channels)
  int             maxIters = 300         // maximum iterations for solver
);
```

Given a polynomial of any order in the form of a vector of coefficients `coeffs`, `cv::solvePoly()` will attempt to compute the roots of that polynomial. Given the array of coefficients `coeffs`, the roots of the following polynomial are computed:

$$\text{coeffs}_n x^n + \text{coeffs}_{n-1} x^{n-1} + \cdots + \text{coeffs}_1 x + \text{coeffs}_0 = 0.$$

These roots are not guaranteed to be real. For an order-n polynomial (i.e., `coeffs` having $n+1$ elements), there will be n roots. As a result, the array `roots` will be returned in a two-channel (real, imaginary) matrix of doubles.

`cv::sort()`

```
void cv::sort(
  cv::InputArray  src,
  cv::InputArray  dst,
  int             flags
);
```

The OpenCV sort function is used for two-dimensional arrays. Only single-channel source arrays are supported. You should not think of this like sorting rows or columns in a spreadsheet; `cv::sort()` sorts every row or column *separately*. The result of the sort operation will be a new array `dst`, which is of the same size and type as the source array.

Sorting can be done on every row or on every column by supplying either the `cv::SORT_EVERY_ROW` or `cv::SORT_EVERY_COLUMN` flag. Sort can be in ascending or descending order, which is indicated by the `cv::SORT_ASCENDING` or `cv::SORT_DESCENDING` flags respectively. One flag from each of the two groups is required.

`cv::sortIdx()`

```
void cv::sortIdx(
  cv::InputArray  src,
  cv::InputArray  dst,
  int             flags
```

```
);
```

Similar to `cv::sort()`, `cv::sortIdx()` is used only for single-channel two-dimensional arrays. `cv::sortIdx()` sorts every row or column *separately*. The result of the sort operation is a new array `dst` of the same size as the source array, but which contains the integer indices of the sorted elements. For example, given an array `A`, a call to `cv::sortIdx( A, B, cv::SORT_EVERY_ROW | cv::SORT_DESCENDING )` would produce:

$$A = \begin{bmatrix} 0.0 & 0.1 & 0.2 \\ 1.0 & 1.1 & 1.2 \\ 2.0 & 2.1 & 2.2 \end{bmatrix} \qquad B = \begin{bmatrix} 2 & 1 & 0 \\ 2 & 1 & 0 \\ 2 & 1 & 0 \end{bmatrix}$$

In this toy case, every row was previously ordered from lowest to highest and sorting has indicated that this should be reversed.

`cv::split()`

```
void cv::split(
  const cv::Mat&   mtx,
  cv::Mat*         mv
);
void cv::split(
  const cv::Mat&   mtx,
  vector<Mat>&     mv                    // STL-style vector of n 1-channel cv::Mat's
);
```

The function `cv::split()` is a special, simpler case of `cv::mixChannels()`. `cv::split()` separates the channels in a multichannel array into multiple single-channel arrays. There are two ways for doing this: in the first, you supply a pointer to a C-style array of pointers to `cv::Mat` objects that `cv::split()` will use for the results of the split operation. In the second option, you supply an STL vector full of `cv::Mat` objects. If you use the C-style array, you need to make sure that the number of `cv::Mat` objects available is (at least) equal to the number of channels in `mtx`. If you use the STL vector form, `cv::split()` will handle the allocation of the result arrays for you.

`cv::sqrt()`

```
void cv::sqrt(
  cv::InputArray  src,
  cv::InputArray  dst
);
```

As a special case of `cv::pow()`, `cv::sqrt()` will compute the element-wise square root of an array. Multiple channels are processed separately.

There is such thing as a square root of a matrix, i.e., a matrix B whose relationship with some matrix A is that $BB = A$. If A is square and positive definite, then if B exists, it is unique.

If A can be diagonalized, then there is a matrix V (made from the eigenvectors of A as columns) such that $\dagger A \dagger = VDV^{-1}$, where D is a diagonal matrix. The square root of a diagonal matrix D is just the square roots of the elements of $\dagger D$. So to compute $A^{1/2}$, we simply use the matrix V and get:

$$A^{1/2} = VD^{1/2}V^{-1}$$

Math fans can easily verify that this expression is correct by explicitly squaring it:

$$\left(A^{1/2}\right)^2 = \left(VD^{1/2}V^{-1}\right)\left(VD^{1/2}V^{-1}\right) = VD^{1/2}V^{-1}VD^{1/2}V^{-1} = VDV^{-1} = A$$

In code, this would look something like[40]:

```
void matrix_square_root( const cv::Mat& A, cv::Mat& sqrtA ) {
  cv::Mat U, V, Vi, E;
  cv::eigen( A, E, U );
  V = U.T();
  cv::transpose( V, Vi ); // inverse of the orthogonal V
  cv::sqrt(E,  E);  //  assume  that  A  is  positively-defined,
otherwise its square root is complex-valued
  sqrtA = V * Mat::diag(E) * Vi;
}
```

---

cv::subtract()

```
void cv::subtract(
  cv::InputArray  src1,                  // First input array
  cv::InputArray  src2,                  // Second input array
  cv::OutputArray dst,                   // Result array
  cv::InputArray  mask  = cv::noArray(), // Optional mask, compute only where nonzero
  int             dtype = -1             // Output type for result array
)
```

$$dst_i = saturate\left(\dagger src1_i - src2_i\right)$$

`cv::subtract()` is a simple subtraction function: it subtracts all of the elements in `src2` from the corresponding elements in `src1` and puts the results in `dst`.

For simple cases the same result can be achieved with the matrix operation:

```
dst = src1 - src2;
```

Accumulation is also supported:

```
dst -= src1;
```

---

cv::sum()

```
cv::Scalar cv::sum(
  cv::InputArray  arr
);
```

$$sum_c = \sum_{i,j} arr_{ci,j}$$

`cv::sum()` sums all of the pixels in each of the channels of the array `arr`. The return value is of type `cv::Scalar`, so `cv::sum()` can accommodate multichannel arrays, but only up to four channels. The sum for each channel is placed in the corresponding component of the `cv::scalar` return value.

cv::trace()

```
cv::Scalar cv::trace(
  cv::InputArray  mat
```

---

[40] Here "something like" means that if you were really writing a responsible piece of code, you would do a lot of checking to make sure that the matrix you were handed was in fact what you thought it was (i.e., square). You would also probably want to check the return values of `cv::eigen()` and `cv::invert()` and think more carefully about the actual methods used for the decomposition and inversion, make sure the eigenvalues are positive before blindly calling `sqrt()` on them.

```
);
```

$$Tr\left(mat\right)_c = \sum_i mat_{ci,i}$$

The trace of a matrix is the sum of all of the diagonal elements. The trace in OpenCV is implemented on top of `cv::diag()`, so it does not require the array passed in to be square. Multichannel arrays are supported, but the trace is computed as a scalar so each component of the scalar will be the sum over each corresponding channel for up to four channels.

`cv::transform()`

```
void cv::transform(
  cv::InputArray  src,
  cv::OutputArray dst,
  cv::InputArray  mtx
);
```

$$dst_{c,i,j} = \sum_{c'} mtx_{c,c'} src_{c'i,j}$$

The function `cv::transform()` can be used to compute arbitrary linear image transforms. It treats a multichannel input array `src` as a collection of vectors in what you could think of as "channel space." Those vectors are then each multiplied by the "small" matrix `mtx`, to affect a transformation in this channel space.

The matrix `mtx` must have as many rows as there are channels in `src`, or that number plus one. In the second case, the channel space vectors in `src` are automatically extended by one and the value `1.0` is assigned to the extended element.

> The exact meaning of this transformation depends on what you are using the different channels for. If you are using the channels as color channels, then this transformation can be thought of as a linear color space transformation. (Transformation between RGB and YUV color spaces is an example of such a transformation. If you are using the channels to represent the $\left(x, y\right)$ or $\left(x, y, z\right)$ coordinates of points, then these transformations can be thought of as rotations (or other geometrical transformations) of those points.

`cv::transpose()`

```
void cv::transpose(
  cv::InputArray  src,                  // Input array, 2-dimensional, n-by-m
  cv::OutputArray dst,                  // Result array, 2-dimensional, m-by-n
);
```

`cv::transpose()` copies every element of `src` into the location in `dst` indicated by reversing the row and column indices. This function does support multichannel arrays; however, if you are using multiple channels to represent complex numbers, remember that `cv::transpose()` does not perform complex conjugation.

> This same result can be achieved with the matrix member function `cv::Mat::t()`. The member function has the advantage that it can be used in matrix expressions like:
>
> ```
> A = B + B.t();
> ```

Utility Functions

There are several operations provided by OpenCV that operate on single values (integers or floating-point numbers). These functions either serve some special function, or are fast implementations of particular

things that come up often enough in vision processing to justify implementing in a custom way for OpenCV.

Table 3-31 Utility and System Functions

Function	Description
`cv::alignPtr()`	Align pointer to given number of bytes
`cv::alignSize()`	Align buffer size to given number of bytes
`cv::allocate()`	Allocate a C-style array of objects
`cvCeil()`[41]	Round float number *x* to nearest integer not smaller than *x*
`cv::cubeRoot()`	Compute the cube root of a number
`cv::CV_Assert()`	Throw an exception if a given condition is not true
`CV_Error()`	Macro to build a `cv::Exception` (from a fixed string) and throw it
`CV_Error_()`	Macro to build a `cv::Exception` (from a formatted string) and throw it
`cv::deallocate()`	Deallocate a C-style array of objects
`cv::error()`	Indicate an error and throw an exception
`cv::fastAtan2()`	Calculate two-dimensional angle of a vector in degrees
`cv::fastFree()`	Deallocate a memory buffer
`cv::fastMalloc()`	Allocate an aligned memory buffer
`cvFloor()`[41]	Round float number *x* to nearest integer not larger than *x*
`cv::format()`	Create an STL string using `sprinf`-like formatting
`cv::getCPUTickCount()`	Get tick count from internal CPU timer
`cv::getNumThreads()`	Count number of threads currently used by OpenCV
`cv::getOptimalDFTSize()`	Compute the best size for an array that you plan to pass to `cv::DFT()`
`cv::getThreadNum()`	Get index of the current thread
`cv::getTickCount()`	Get tick count from generic source
`cv::getTickFrequency()`	Get number or ticks per second (see `cv::getTickCount()`)

[41] This function has something of a legacy interface. It is a C definition, not C++ (see core *../types_c.h*) where it is defined as an inline function. There are several others with a similar interface.

`cvIsInf()`[41]	Check if a floating-point number *x* is infinity
`cvIsNaN()`[41]	Check if a floating-point number *x* is "Not a Number"
`cvRound()`	Round float number *x* to the nearest integer
`cv::setNumThreads()`	Set number of threads used by OpenCV
`cv::setUseOptimized()`	Enables or disables the use of optimized code (SSE2, etc.)
`cv::useOptimized()`	Indicates status of optimized code enabling (see `cv::setUseOptimized()`)

cv::alignPtr()

```
template<T> T* cv::alignPtr(              // Return aligned pointer of type T*
  T*  ptr,                                // pointer, unaligned
  int n   = sizeof(T)                     // align to block size, a power of 2
);
```

Given a pointer of any type, this function computes an aligned pointer of the same type according to the following computation:

```
(T*)(((size_t)ptr + n+1) & -n)
```

> On some architectures, it is not even possible to read a multi-byte object from an address that is not evenly divisible by the size of the object (i.e.,.,., by 4 for a 32-bit integer). On architectures such as x86, this is handled for you automatically by the CPU by using multiple reads and assembling your value from those reads at the cost of a substantial penalty in performance.

cv::alignSize()

```
size_t cv::alignSize(                     // minimum size >='sz' divisible by 'n'
  size_t sz,                              // size of buffer
  int n   = sizeof(T)                     // align to block size, a power of 2
);
```

Given a number `n` (typically a return value from `sizeof()`), and a size for a buffer `sz`, `cv::alignSize()` computes the size that this buffer should be in order to contain an integer number of objects of size n, i.e., the minimum number that is greater or equal to `sz` yet divisible by `n`. The following formula is used:

```
(sz + n-1) & -n
```

cv::allocate()

```
template<T> T* cv::allocate(              // Return pointer to allocated buffer
  size_t sz                               // size of buffer, multiples of sizeof(T)
);
```

The function `cv::allocate()` functions similarly to the array form of `new`, in that it allocates a C-style array of `n` objects of type `T`, calls the default constructor for each object, and returns a pointer to the first object in the array.

cv::deallocate()

```
template<T> void cv::deallocate(
  T*     ptr,                             // Pointer to buffer to free
  size_t sz                               // size of buffer, multiples of sizeof(T)
);
```

The function `cv::deallocate()` functions similarly to the array form of `delete`, in that it deallocates a C-style array of `n` objects of type `T`, and calls the destructor for each object. `cv::deallocate()` is used to deallocate objects allocated with `cv::allocate()`. The number of elements `n` passed to `cv::deallocate()` must be the same as the number of objects originally allocated with `cv::allocate()`.

cv::fastAtan2()

```
float cv::fastAtan2(                    // Return value is 32-bit float
  float y,                              // y input value (32-bit float)
  float x                               // x input value (32-bit float)
);
```

This function computes the arctangent of an `x,y` pair and returns the angle from the origin to the indicated point. The result is reported in degrees ranging from `0.0` to `360.0`, inclusive of `0.0` but not inclusive of `360.0`.

cvCeil()

```
int cvCeil(                             // Return the smallest int >= x
  float x                               // input value (32-bit float)
);
```

Given a floating-point number `x`, `cvCeil()` computes the smallest integer not smaller than `x`. If the input value is outside of the range representable by a 32-bit integer, the result is undefined.

cv::cubeRoot()

```
float cv::cubeRoot(                     // Return value is 32-bit float
  float x                               // input value (32-bit float)
);
```

This function computes the cubed root of the argument x. Negative values of `x` are handled correctly (i.e., the return value is negative).

cv::CV_Assert() and CV_DbgAssert()

```
// example
CV_Assert( x!=0 )
```

`CV_Assert()` is a macro that will test the expression passed to it and, if that expression evaluates to `False` (or `0`), it will throw an exception. The `CV_Assert()` macro is always tested. Alternatively, you can use `CV_DbgAssert()`, which will only be tested in debug compilations.

cv::CV_Error() and CV_Error_()

```
// example
CV_Error( ecode, estring )
CV_Error_( ecode, fmt, ... )
```

The macro `CV_Error()` allows you to pass in an error code `ecode` and a fixed C-style character string `estring`, which it then packages up into a `cv::Exception` and passes that to `cv::error()` to be handled. The variant macro `CV_Error_()` is used if you need to construct the message string on the fly. `CV_Error_()` accepts the same `ecode` as `CV_Error()`, but then expects a `sprintf()` style format string followed by a variable number of arguments as would be expected by `sprintf()`.

cv::error()

```
void cv::error(
  const cv::Exception& ex               // Exception to be thrown
);
```

This function is mostly called from `CV_Error()` and `CV_Error_()`. If your code is compiled in a non-debug build, it will throw the exception `ex`. If your code is compiled in a debug build, it will deliberately provoke a memory access violation so that the execution stack and all of the parameters will be available for whatever debugger you are running.

You will probably not call `cv::error()` directly, but rather rely on the macros `CV_Error()` and `CV_Error_()` to throw the error for you. These macros take the information you want displayed in the exception and package it up for you and pass the resulting exception to `cv::error()`.

cv::fastFree()

```
void cv::fastFree(
  void* ptr                                 // Pointer to buffer to be freed
);
```

This routine deallocates buffers that were allocated with `cv::fastMalloc()` (covered next).

cv::fastMalloc()

```
void* cv::fastMalloc(                       // Pointer to allocated buffer
  size_t size                               // Size of buffer to allocate
);
```

`cv::FastMalloc()` works just like the `malloc()` you are familiar with, except that it is often faster, and it does buffer size alignment for you. This means that if the buffer size passed is more than 16 bytes, the returned buffer will be aligned to a 16 byte boundary.

cvFloor()

```
int cvFloor(                                // Return the largest int <= x
  float x                                   // input value (32-bit float)
};
```

Given a floating-point number `x`, `cv::Floor()` computes the largest integer not larger than `x`. If the input value is outside of the range representable by a 32-bit integer, the result is undefined.

cv::format()

```
string cv::format(                          // Return STL-string
  const char* fmt,                          // formatting string, as sprintf()
  ...                                       // vargs, as sprintf()
);
```

This function is essentially the same as `sprintf()` from the standard library, but rather than requiring a character buffer from the caller, it constructs an STL string object and returns that. It is particularly handy for formatting error messages for the `Exception()` constructor (which expects STL strings as argument).

cv::getCPUTickCount()

```
int64 cv::getCPUTickCount( void );          // long int CPU for tick count
```

This function reports the number of CPU ticks on those architectures that have such a construct (including, but not limited to, x86 architectures). It is important to know, however, that the return value of this function can be very difficult to interpret on many architectures. In particular, because on a multi-core system a thread can be put to sleep on one core and wake up on another, the difference between the results to two subsequent calls to `cv::getCPUTickCount()` can be misleading, even completely meaningless. Therefore, unless you are certain you know what you are doing, it is best to use `cv::getTickCount()` for timing measurements.[42] This function is best for tasks like initializing random number generators.

cv::getNumThreads()

```
int cv::getNumThreads( void );              // number of threads allocated to OpenCV
```

Return the current number of threads being used by OpenCV.

---

[42] Of course, if you *really do know* what you are doing, then there is no more accurate way to get detailed timing information than from the CPU timers themselves.

cv::getOptimalDFTSize()

```
int cv::getOptimalDFTSize( int n );       // best size array to use for dft, >= n
```

When making calls to `cv::dft()`, the algorithm used by OpenCV to compute the transform is extremely sensitive to the size of the array passed to `cv::dft()`. The preferred sizes do obey a rule for their generation, but that rule is sufficiently complicated that it is (at best) an annoyance to compute the correct size to which to pad your array every time. The function `cv::getOptimalDFTSize()` takes as argument the size of the array you would have passed to `cv::dft()`, and returns the size of the array you should pass to `cv::dft()`. The usage of this information is to just create a larger array into which you can copy your data and pad out the rest with zeros.

cv::getThreadNum()

```
int cv::getThreadNum( void );             // int, id of this particular thread
```

If your OpenCV library was compiled with OpenMP support, it will return the index (starting from zero) of the currently executing thread.

cv::getTickCount()

```
int64 cv::getTickCount( void );           // long int CPU for tick count
```

This function returns a tick count relative to some architecture-dependent time. The rate of ticks is also architecture and operating system dependent, however; the time per tick can be computed by `cv::getTickFrequency()` (reviewed next).below). This function is preferable to `cv::getCPUTickCount()` for most timing applications, as it is not affected by low-level issues such as which core your thread is running on and automatic throttling of CPU frequency (which most modern processors do for power management reasons).

cv::getTickFrequency()

```
double cv::getTickFrequency( void );      // Tick frequency in seconds as 64-bit
```

When using `cv::getTickCount()` for timing analysis, the exact meaning of a tick is, in general, architecture dependent. The function `cv::getTickFrequency()` computes the conversion between clock time (i.e., seconds) and abstract "ticks."".

> Thus to compute the time required for some specific thing to happen (such as a function to execute), one need only call `cv::getTickCount()` before and after the function call, subtract the results, and divide by the value of `cv::getTickFrequency()`.

cvIsInf()

```
int cvIsInf( double x );              // return 1 if x is IEEE754 "infinity"
```

The return value of `cvIsInf()` is one if `x` is plus or minus infinity and zero otherwise. The infinity test is the test implied by the IEEE754 standard.

cvIsNaN()

```
int cvIsNan( double x );              // return 1 if x is IEEE754 "Not a number"
```

The return value of `cvIsNaN()` is one if `x` is "not a number" and zero otherwise. The NaN test is the test implied by the IEEE754 standard.

cvRound()

```
int cvRound( double x );              // Return integer nearest to 'x'
```

Given a floating-point number `x`, `cvRound()` computes the integer closest to `x`. If the input value is outside of the range representable by a 32-bit integer, the result is undefined.

cv::setNumThreads()

```
void cv::setNumThreads( int nthreads );   // Set number of threads OpenCV can use
```

When OpenCV is compiled with OpenMP support, this function sets the number of threads that OpenCV will use in parallel OpenMP regions. The default value for the number of threads is the number of logical cores on the CPU (i.e., if we have four cores each with two hyper-threads, there will be eight threads by default). If `nthreads` is set to `0`, the number of threads will be returned to this default value.

`cv::setUseOptimized()`

```
void cv::setUseOptimized( bool on_off );  // If false, turn off optimized routines
```

Though early versions of OpenCV relied on outside libraries (such as IPP, the Intel Performance Primitives library) for access to high performance optimizations such as SSE2 instructions, later versions have increasingly moved to containing that code in the OpenCV itself. By default the use of these optimized routines is enabled, unless you specifically disabled it when you built your installation of the library. However, you can turn the use of these optimizations on and off at any time with `cv::setUseOptimized()`.

> The test of the global flag for optimizations usage is done at a relatively high level inside of the OpenCV library functions. The implication of this is that you should not call `cv::setUseOptimized()` while any other routines might be running (on any threads). You should make sure to call this routine when you can be certain you know what is and what is not running, preferably from the very top level of your application.

`cv::useOptimized()`

```
bool cv::useOptimized( void );            // return true if optimizations are enabled
```

At any time, you can check the state of the global flag, which enables the use of high performance optimizations (see `cv::setUseOptimized()`) by calling `cv::useOptimized()`. `True` will be returned only if these optimizations are currently enabled; otherwise, this function will return `False`.

Objects That Do Stuff

As the OpenCV library has evolved, it has become increasingly common to introduce new objects which encapsulate functionality that is too complicated to be associated with a single function and which, if implemented as a set of functions, would cause the overall function space of the library to become too cluttered.

As a result, new functionality is often represented by an associated new object type, which can be thought of as a "machine" that does whatever this function is. Most of these machines have an overloaded `operator()`, which officially makes them *function objects* or *functors*. If you are not familiar with this programming idiom, the important idea is that unlike "normal" functions, function objects are created, and can maintain state information inside them. As a result, they can be set up with whatever data or configuration they need, and they are "asked" to perform services through either common member functions, or by calling them as functions (usually via the overloaded `operator()`[43]).

Principal Component Analysis (cv::PCA)

Principal component analysis is the process of analyzing a distribution in many dimensions and extracting from that distribution the particular subset of dimensions that carry the most information. The dimensions computed by PCA are not necessarily the basis dimensions in which the distribution was originally specified. Indeed, one of the most important aspects of PCA is the ability to generate a new basis in which

[43] Here the word "usually" means "usually when people program function objects," but does not turn out to mean "usually" for the OpenCV library. There is a competing convention in the OpenCV library which uses the overloaded `operator()` to load the configuration, and a named member to provide the fundamental service of the object. This convention is substantially less canonical in general, but quite common in the OpenCV library.

the axes of the new basis can be ordered by their importance[44]. These basis vectors will turn out to be the eigenvectors of the covariance matrix for the distribution as a whole, and the corresponding eigenvalues will tell us about the extent of the distribution in that dimension.

We are now in a position to explain why PCA is handled by one of these function objects. Given a distribution once, the PCA object can compute and retain this new basis. The big advantage of the new basis is that the basis vectors that correspond to the large eigenvalues carry most of the information about the objects in the distribution. Thus, without losing much accuracy, we can throw away the less informative dimensions. This dimension reduction is called a KLT Transform.[45] Once you have loaded a sample distribution and the principal components are computed, you might want to use that information to do various things, such as applying the KLT transform to new vectors. By making the PCA functionality a function object, it can "remember" what it needs to know about the distribution you gave it, and thereafter use that information to provide the "service" of transforming new vectors on demand.

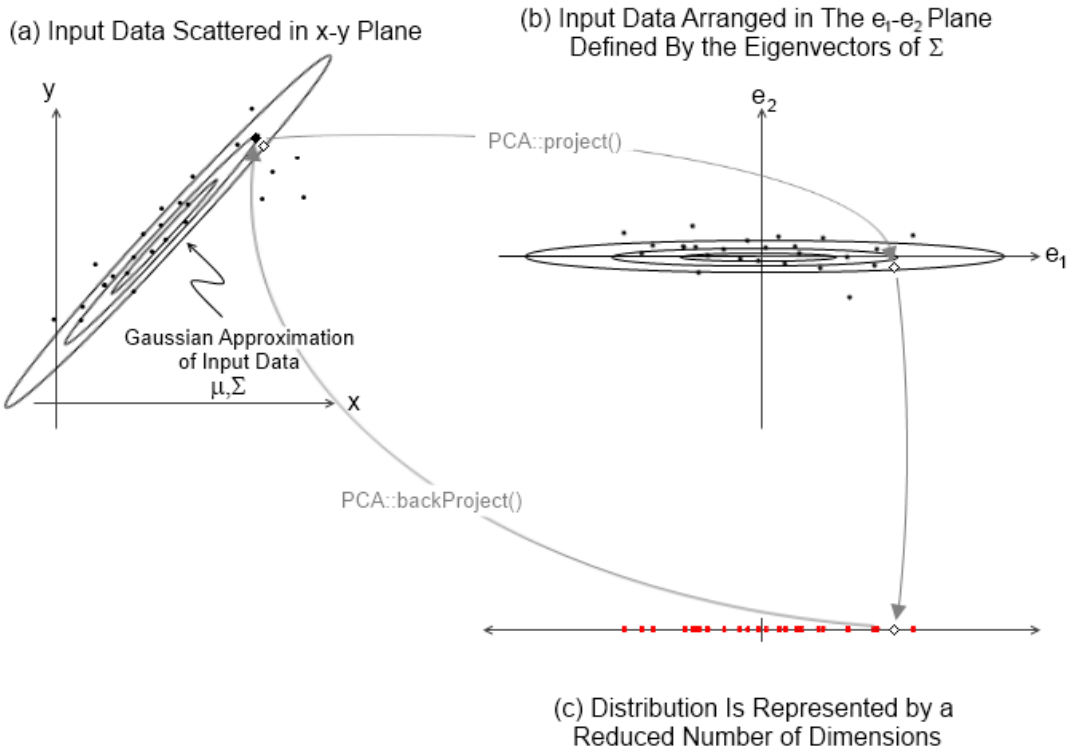


Figure 3-4. (a) Input data is characterized by a Gaussian approximation; (b) the data is projected into the space implied by the eigenvectors of the covariance of that approximation; (c)the data is projected by the KLT projection to a space defined only by the most "useful" of the eigenvectors; superimposed: a new data point (white diamond) is projected to the reduced dimension space by `cv::PCA::project()`*; that same point is brought back to the original space (black diamond) by* `cv::PCA::backProject()`

`cv::PCA::PCA()`

```
PCA::PCA();
```

[44] You might be thinking to yourself "hey, this sounds like machine learning, what is it doing in this chapter?" This is not a bad question. In modern computer vision, machine learning is becoming increasingly intrinsic to an ever -growing list of algorithms. For this reason, component capabilities, such as PCA and SVD, are increasingly considered "building blocks."

[45] KLT stands for "Karhunen-Loeve Transform,"", so that phrase "KLT Transformation" is a bit of a malapropism. It is however at least as often said one way as the other.

```
PCA::PCA(
  cv::InputArray data,                    // Data, as rows or cols in 2d array
  cv::InputArray mean,                    // average, if known, 1-by-n or n-by-1
  int            flags,                   // Are vectors rows or cols of 'data'
  int            maxComponents = 0        // Max dimensions to retain
);
```

The PCA object has a default constructor, `cv::PCA()`, which simply builds the PCA object and initializes the empty structure. The second form executes the default construction, then immediately proceeds to pass its arguments to `PCA::operator()()` (discussed next).

`cv::PCA::operator()()`

```
PCA::operator()(
  cv::InputArray data,                    // Data, as rows or cols in 2d array
  cv::InputArray mean,                    // average, if known, 1-by-n or n-by-1
  int            flags,                   // Are vectors rows or cols of 'data'
  int            maxComponents = 0        // Max dimensions to retain
);
```

The overloaded `operator()()` for `PCA` builds the model of the distribution inside of the PCA object. The argument `data` is an array containing all of the samples that constitute the distribution. Optionally, `mean`, a second array that contains the mean value in each dimension, can be supplied. (`mean` can either be n-by-1 or 1-by-n.) The data can be arranged as an n-by-D (n rows of samples each of D dimensions) or D-by-n array (n columns of samples each of D dimensions). The argument `flags` is currently used only to specify the arrangement of the data in `data` and `mean`. In particular, `flags` can be set either to `cv::PCA_DATA_AS_ROW` or `cv::PCA_DATA_AS_COL`, to indicate that either data is n-by-D and mean is n-by-1 or data is D-by-n and mean is 1-by-n respectively. The final argument `maxComponents` specifies the maximum number of components (dimensions) that PCA should retain. By default, all of the components are retained.

> Any subsequent call to `cv::PCA::operator()()` will overwrite the internal representations of the eigenvectors and eigenvalues, so you can recycle a PCA object whenever you need to (i.e., you don't have to reallocate a new one for each new distribution you want to handle if you no longer need the information about the previous distribution).

`cv::PCA::project()`

```
cv::Mat PCA::project(                     // Return results, as a 2d matrix
  cv::InputArray  vec                     // points to project, rows or cols, 2d
) const;

void PCA::project(
  cv::InputArray  vec                     // points to project, rows or cols, 2d
  cv::OutputArray result                  // Result of projection, reduced space
) const;
```

Once you have loaded your reference distribution with `cv::PCA::operator()()`, you can start asking the PCA object to do useful things for you like compute the KLT projection of some set of vectors onto the basis vectors computed by the principal component analysis. The function `cv::PCA::project()` has two forms, the first returns a matrix containing the results of the projections, while the second writes the results to a matrix you provide. The advantage of the first form is that you can use it in matrix expressions.

The argument `vec` contains the input vectors. `vecvec` is required to have the same number of dimensions and the same "orientation" as the `data` array that was passed to PCA when the distribution was first analyzed (i.e., if your data was columns when you called `cv::PCA::operator()()`, `vec` should also have the data arranged into columns).

The returned array will have the same number of objects as `vec` with the same orientation, but the dimensionality of each object will be whatever was passed to `maxComponents` when the PCA object was first configured with `cv::PCA::operator()()`.

`cv::PCA::backProject()`

```
cv::Mat PCA::backProject(                // Return results, as a 2d matrix
  cv::InputArray  vec                    // Result of projection, reduced space
} const;

void PCA::backProject(
  cv::InputArray  vec                    // Result of projection, reduced space
  cv::OutputArray result                 // "reconstructed" vectors, full dimension
) const;
```

The `cv::PCA::backProject()` function performs the reverse operation of `cv::PCA::project()`, with the analogous restrictions on the input and output arrays. The argument `vec` contains the input vectors, which this time are from the projected space. They will have the same number of dimensions as you specified with `maxComponents` when you configured the PCA object and the same "orientation" as the `data` array that was passed to PCA when the distribution was first analyzed (i.e., if your data was columns when you called `cv::PCA::operator()`, `vec` should also have the data arranged into columns).

The returned array will have the same number of objects as `vec` with the same orientation, but the dimensionality of each object will be the dimensionality of the original data you gave to the PCA object when the PCA object was first configured with `cv::PCA::operator()()`).

If you did not retain all of the dimensions when you configured the PCA object in the beginning, the result of back projecting vectors, which are themselves projections of some vector $\vec{x}$ from the original data space, will not be equal to $\vec{x}$. Of course, the difference should be small, even if the number of components retained was much smaller than the original dimension of $\dagger\vec{x}$, as this is the very point of using PCA in the first place.

Singular Value Decomposition (cv::SVD)

The class `cv::SVD` is similar to `cv::PCA` above in that it is the same kind of function object. Its purpose, however, is quite different. The singular value decomposition is essentially a tool for working with non-square, ill-conditioned, or otherwise poorly-behaved matrices such those encountered when solving under-determined linear systems.

Mathematically, the singular value decomposition (SVD) is the decomposition of an m-by-n matrix **A** into the form:

$$A = U \cdot W \cdot V^T$$

where **W** is a diagonal matrix and **U** and **V** are m-by-n and n-by-n (unitary) matrices. Of course, the matrix **W** is also an m-by-n matrix, so here "diagonal" means that any element whose row and column numbers are not equal is necessarily `0`.

`cv::SVD()`

```
SVD::SVD();
SVD::SVD(
  cv::InputArray A,                    // Linear system, array to be decomposed
  int            flags = 0             // what to construct, can A can scratch
);
```

The SVD object has a default constructor, `cv::SVD()`, which simply builds the SVD object and initializes the empty structure. The second form basically executes the default construction, then immediately proceeds to pass its arguments to `cv::SVD::operator()()` (discussed next).

cv::SVD::operator()()

```
SVD::& SVD::operator() (
  cv::InputArray A,                    // Linear system, array to be decomposed
  int            flags = 0             // what to construct, can A be scratch
);
```

The operator `cv::SVD::operator()()` passes to the cv::SVD object the matrix to be decomposed. The matrix **A**, as described earlier, is decomposed into a matrix **U**, a matrix **V** (actually the transpose of **V**, which we will call **Vt**), and a set of singular values (which are the diagonal elements of the matrix **W**).

The flags can be any one of `cv::SVD::MODIFY_A`, `cv::SVD::NO_UV`, or `cv::SVD::FULL_UV`. The latter two are mutually exclusive, but either can be combined with the first. The flag `cv::SVD::MODIFY_A` indicates that it is OK to modify the matrix **A** when computing. This speeds up computation a bit and saves some memory. It is more important when the input matrix is already very large. The flag `cv::SVD::NO_UV` tells `cv::SVD` to not explicitly compute the matrices **U** and **Vt**, while the flag `cv::SVD::FULL_UV` indicates that not only would you like **U** and **Vt** computed, but that you would like them to be represented as full size square orthogonal matrices.

cv::SVD::compute()

```
void SVD::compute(
  cv::InputArray  A,                   // Linear system, array to be decomposed
  cv::OutputArray W,                   // Output array 'W', singular values
  cv::OurputArray U,                   // Output array 'U', left singular vectors
  cv::OutputArray Vt,                  // Output array 'Vt', right singular vectors
  int             flags = 0            // what to construct, and if A can be scratch
);
```

This function is an alternative to using `cv::SVD::operator()()` to decompose the matrix **A**. The primary difference is that the matrices **W**, **U**, and **Vt** are stored in the user-supplied arrays, rather than being kept internally. The flags supported are exactly those supported by `cv::SVD::operator()()`.

cv::SVD::solveZ()

```
void SVD::solveZ(
  cv::InputArray  A,                   // Linear system, array to be decomposed
  cv::OutputArray z                    // One possible solution (unit length)
);
```

$$\dagger\vec{z} = \dagger \mathbf{argmin}_{\vec{x}\cdot\vec{x}=1} A\bullet\vec{x}$$

Given an under-determined (singular) linear system, `cv::SVD::solveZ()` will (attempt to) find a unit length solution of $A\bullet\vec{x} = 0$ and place the solution in the array `y`. Because the linear system is singular, however, it may have no solution, or it may have an infinite family of solutions. `cv::SVD::solveZ()` will find a solution, if one exists. If no solution exists, then the return value $\vec{y}$ will be a vector that minimizes $\dagger A\bullet\vec{x}$, even if this is not, in fact, zero.

cv::SVD::backSubst()

```
void SVD::backSubst(
  cv::InputArray  b,                   // Right-hand side of linear system
  cv::OutputArray x                    // Found solution to linear system
);

void SVD::backSubst(
  cv::InputArray  W,                   // Output array 'W', singular values
  cv::InputArray  U,                   // Output array 'U', left singular vectors
```

```
  cv::InputArray  Vt,                    // Output array 'Vt', right singular vectors
  cv::InputArray  b,                     // Right hand side of linear system
  cv::OutputArray x                      // Found solution to linear system
);
```

Assuming that the matrix **A** has been previously passed to the `cv::SVD` object (and thus decomposed into **U**, **W**, and **Vt**), the first form of `cv::SVD::backSubst()` attempts to solve the system:

$$\left(UWV^T\right)\vec{x} = \vec{b}\ .$$

The second form does the same thing, but expects the matrices **W**, **U**, and **Vt** to be passed to it as arguments. The actual method of computing `dst` is to evaluate the following expression:

$$\overrightarrow{x} = V_t^T \cdot diag\left(W\right)^{-1} \cdot U^T \cdot \overrightarrow{b} \sim A^{-1} \cdot \overrightarrow{b}.$$

This method produces a *pseudo-solution* for an over-determined system, which is the best solution in the sense of minimizing the least-squares error.[46] Of course, it will also exactly solve a correctly determined linear system.

> In practice, it is relatively rare that you would want to use `cv::SVD::backSubst()` directly. This is because you can do precisely the same thing by calling `cv::solve()` and passing the `cv::DECOMP_SVD` method flag—which is a lot easier. It is only in the less common case in which you need to solve many different systems with the same *left-hand* side (**x**) that you would be better off calling `cv::SVD::backSubst()` directly. (As opposed to solving the *same* system many times with different *righthand* sides (**b**), which you might just as well do with `cv::solve()`.

Random Number Generator (`cv::RNG`)

The random number generator object RNG holds the state of a pseudorandom sequence that generates random numbers. The benefit of doing things this way is that you can conveniently maintain multiple streams of pseudorandom numbers.

> When programming large systems, it is a good programming practice to use separate random number streams in different modules of the code. In this way, the removal of one module does not change the behavior of the streams in the other modules.

Once created, the random number generator provides the "service" of generating random numbers on demand, drawn from either a uniform or a Gaussian distribution. The generator uses the *Multiply with Carry* algorithm (G. Marsaglia) for uniform distributions and the *Ziggurat* algorithm (G. Marsaglia and W. W. Tsang) for the generation of numbers from a Gaussian distribution.

`cv::theRNG()`

```
cv::RNG& theRNG( void );               // Return a random number generator
```

The function `cv::theRNG()` returns the default random number generator for the thread from which it was called. OpenCV automatically creates one instance of `cv::RNG` for each thread in execution. This is the same random number generator that is implicitly accessed by functions like `cv::randu()` or `cv::randn()`. Those functions are convenient and just as quick if you want a single number, or to initialize an array, but if you have a loop of your own that needs to generate a lot of random numbers, you are better off grabbing a reference to a random number generator (in this case, the default generator, but

[46] The object $diag(\boldsymbol{W})^{-1}$ is a matrix whose diagonal elements λ_i^* are defined in terms of the diagonal elements λ_i of $\boldsymbol{W}$ by $\lambda_i^* = \lambda_i^{-1}$ for $\lambda_i \geq \varepsilon$. This value ε is the *singularity threshold*, a very small number that is typically proportional to the sum of the diagonal elements of $\boldsymbol{W}$ (i.e., $\varepsilon_o \sum_i \lambda_i$).

you could use your own instead) and using `RNG::operator T()` to get your random numbers (more on that operator follows).

`cv::RNG()`

```
cv::RNG::RNG( void );
cv::RNG::RNG( uint64 state );              // create using the seed 'state'
```

You can create an RNG object with either the default constructor, or you can pass it a 64-bit unsigned integer that it will use as the seed of the random number sequence. If you call the default constructor (or pass 0 to the second variation) the generator will initialize with a standardized value.[47]

`cv::RNG::operator T()`

```
cv::RNG::operator uchar();
cv::RNG::operator schar();
cv::RNG::operator ushort();
cv::RNG::operator short int();
cv::RNG::operator int();
cv::RNG::operator unsigned();
cv::RNG::operator float();
cv::RNG::operator double();
```

This is really a set of different methods that return a new random number from `cv::RNG` of some specific type. Each of these is an overloaded cast operator, so in effect you cast the RNG object to whatever type you want:

Example 3-2: Using the default random number generator we generate a pair of integers and a pair of floating-point numbers; the style of the cast operation is up to you, this example shows both the `int(x)` *and the* `(int)x` *forms*

```
cv::RNG rng = cv::theRNG();
cout << "An integer:      " << (int)rng   << endl;
cout << "Another integer: " << int(rng)   << endl;
cout << "A float:         " << (float)rng << endl;
cout << "Another float:   " << float(rng) << endl;
```

When integer types are generated, they will be generated across the entire range of available values (using the MWC algorithm described earlier and thus uniformly). When floating-point types are generated, they will always be in the range from the interval [0.0, 1.0).[48]

`cv::RNG::operator()`

```
unsigned int cv::RNG::operator()();       // Return random value from 0-UINT_MAX
unsigned int cv::RNG::operator()( unsigned int N );  // Return value from 0-(N-1)
```

When generating integer random numbers, the overloaded `operator()()` allows a convenient way to just grab another one. In essence, calling `my_rng()` is equivalent to calling `(unsigned int)my_rng`. The somewhat more interesting form of `cv::RNG::operator()()` takes an integer argument `N`. This form returns a random unsigned integer *modulo* `N` (using the MWC algorithm described earlier and thus uniformly). Thus, the range of integers returned by `my_rng( N )` is then the range of integers from `0` to `N-1`.

`cv::RNG::uniform()`

```
int        cv::RNG::uniform( int a,    int b    );       // Return value from a-(b-1)
```

---

[47] This "standard value" is not zero because, for that value, many random number generators, including the ones used by RNG, will return nothing but zeros thereafter. Currently, this standard value is $2^{32} - 1$.

[48] In case this notation is not familiar, the designation of an interval using square brackets, [], '[' indicates that this limit is inclusive, and the designation using parentheses, (), '(' indicates that this limit is noninclusiveinclusive. Thus the notation [0.0,1.0) means an interval from 0.0 to 1.0 inclusive of 0.0 but not inclusive of 1.0.

```
float  cv::RNG::uniform( float a,  float b  );       // Return value in range [a,b)
double cv::RNG::uniform( double a, double b );       // Return value in range [a,b)
```

This function allows you to generate a random number uniformly (using the MWC algorithm) in the interval [a, b).

> It is important to note that the C++ compiler does not consider the return value of a function when determining which of multiple similar forms to use, only the arguments. As a result, if you call: `float x = my_rng.uniform(0,1)` you are going to get `0.f`, because `0` and `1` are integers and the only integer in the interval [`0`, `1`) is `0`. If you want a floating-point number, you should use something like `my_rng.uniform(0.f,1.f)`, and for a double, use `my_rng.uniform(0.,1.)`. Of course, explicit casting of the arguments also works.

`cv::RNG::gaussian()`

```
double  cv::RNG::gaussian( double sigma ); // Gaussian number, zero mean, std-dev='sigma'
```

This function allows you to generate a random number from a zero-mean Gaussian distribution (using the Ziggurat algorithm) with standard deviation `sigma`.

`cv::RNG::fill()`

```
void  cv::RNG::fill(
  InputOutputArray mat,                  // Input array, values will be overwritten
  int              distType,             // Type of distribution (Gaussian or uniform)
  InputArray       a,                    // min (uniform) or mean (Gaussian)
  InputArray       b                     // max (uniform) or std-deviation (Gaussian)
 );
```

The `cv::RNG::fill()` algorithm fills a matrix `mat` of up to four channels with random numbers drawn from a specific distribution. That distribution is selected by the `distType` argument, which can be either `cv::RNG::UNIFORM` or `cv::RNG::NORMAL`. In the case of the uniform distribution, each element of `mat` will be filled with a random value generated from the interval $\dagger\, mat_{c,i} \in \left[a_c, b_c \right)$. In the case of the Gaussian (`cv::RNG::NORMAL`) distribution, each element is generated from a distribution with mean taken from `a` and standard deviation taken from `b`: $\dagger\, mat_{c,i} \in N\left(a_c, b_c \right)$. It is important to note that the arrays `a` and `b` are not of the dimension of `mat`, they are n_c-by-1 or 1-by-n_c where n_c is the number of channels in `mat` (i.e., there is not a separate distribution for each element of mat, `a` and `b` specify one distribution, not one distribution for every element of `mat`.)

> If you have a multichannel array, then you can generate individual entries in "channel-space" from a multivariate distribution simply by giving the appropriate mean and standard deviation for each channel in the input arrays `a` and `b`. This distribution, however, will be drawn from a distribution with only zero entries in the off-diagonal elements of its covariance matrix. (This is because each element is generated completely independently of the others.) If you need to draw from a more general distribution, the easiest method is to generate zero-mean values using an identity covariance matrix with `cv::RNG::fill()`, and then transform (translate and rotate) back to your original basis using `cv::transform()`.

Summary

In this chapter, we introduced some frequently encountered basic data structures. In particular, we looked at the all-important OpenCV array structure `cv::Mat`, which can contain matrices, images, and multidimensional arrays.

Exercises

In the following exercises, you may need to refer to the manual at http://docs.opencv.org/ .

1. Find and open *…/opencv/modules/core/include/opencv2/core/core.hpp*. Read through and find the many conversion helper functions.
   a) Choose a negative floating-point number. Take its absolute value, round it, and then take its ceiling and floor.[49]
   b) Generate some random numbers.
   c) Create a floating-point `cv::Point2f` and convert it to an integer `cv::Point2i`.
   d) Convert a `cv::Point2i` to a `CvPoint2f`.
2. This exercise will accustom you to the idea of many functions taking matrix types. Create a two-dimensional matrix with three channels of type byte with data size 100-by-100. Set all the values to 0.
   a) Draw a circle in the matrix using the `cv::circle()` function:

```
void circle(
  cv::Mat&          img,                // Image to be drawn on
  cv::Point         center,             // Location of circle center
  int               radius,             // Radius of circle
  const cv::Scalar& color,              // Color, RGB form
  int               thickness = 1,      // Thickness of line
  int               lineType  = 8,      // Connectedness, 4 or 8
  int               shift     = 0       // Bits of radius to treat as fraction
);
```

   b) Display this image using methods described in Chapter 2.

1. Create a two-dimensional matrix with three channels of type byte with data size 100-by-100, and set all the values to 0. Use the element access member: `m.at<cv::Vec3f>` to point to the middle ("green") channel. Draw a green rectangle between (20, 5) and (40, 20).
2. Create a three-channel RGB image of size 100-by-100. Clear it. Use pointer arithmetic to draw a green square between (20, 5) and (40, 20).
3. Practice using the block access methods (Table 3-16). Create a 210-by-210 single-channel byte image and zero it. Within the image, build a pyramid of increasing values using the submatrix constructor and the `cv::Range` object. That is: the outer border should be 0, the next inner border should be 20, the next inner border should be 40, and so on until the final innermost square is set to value 200; all borders should be 10 pixels wide. Display the image.
4. Use multiple image objects for one image. Load an image that is at least 100-by-100. Create two additional image objects using the first object and the submatrix constructor. Create the new images with width at 20 and the height at 30, but with their origins located at pixel at (5, 10) and (50, 60), respectively. Logically invert the two images using the 'not' logical inversion operator. Display the loaded image, which should have two inverted rectangles within the larger image.
5. Add an `CV_DbgAssert( condition )` to the code of question 4 that will be triggered by a condition in the program. Build the code in debug, run it and see the assert being triggered. Now build it in release mode and see that the condition is not triggered.
6. Create a mask using `cv::compare()`. Load a real image. Use `cv::split()` to split the image into red, green, and blue images.
   a) Find and display the green image.
   b) Clone this green plane image twice (call these clone1 and clone2).

---

[49] Remember that `cvFloor()` and `cvCeil()` are legacy functions and are found in *…/opencv/modules/core/include/opencv2/core/types_c.h*.

c) Find the green plane's minimum and maximum value.

d) Set clone1's values to *thresh = (unsigned char)((maximum - minimum)/2.0)*.

e) Set `clone2` to 0 and use `cv::compare(green_image, clone1, clone2, cv::CMP_GE)`. Now `clone2` will have a mask of where the value exceeds thresh in the green image.

f) Finally, compute the value: `green_image = green_image - thresh/2` and display the results. (Bonus: assign this value to `green_image` only where `clone2` is non-zero.)

4

Graphical User Interface

HighGUI: Portable Graphics Toolkit

The OpenCV functions that allow us to interact with the operating system, the filesystem[1], and hardware such as cameras are collected into a sub-library called HighGUI (which stands for "high-level graphical user interface"). HighGUI allows us to open windows, to display images, to read and write graphics-related files (both images and video), and to handle simple mouse, pointer, and keyboard events. We can also use it to create other useful doodads—like sliders, for example—and then add them to our windows. If you are a GUI guru in your window environment of choice, then you might find that much of what HighGUI offers is redundant. Yet, even so, you might find that the benefit of cross-platform portability is itself a tempting morsel.

From our initial perspective, the HighGUI library in OpenCV can be divided into three parts: the hardware part, the filesystem part, and the GUI part. We will take a moment to overview what is in each part before we really dive in.

The hardware part is primarily concerned with the operation of cameras. In most operating systems, interaction with a camera is a tedious and painful task. HighGUI allows an easy way to query a camera and retrieve its latest image. It hides all of the nasty stuff, and that keeps us happy.

The filesystem part is concerned primarily with loading and saving images. One nice feature of the library is that it allows us to read video using the same methods we would use to read a camera. We can therefore abstract ourselves away from the particular device we're using and get on with writing interesting code. In a similar spirit, HighGUI provides us with a (relatively) universal pair of functions to load and save still images. These functions simply rely on the filename extension and automatically handle all of the decoding or encoding that is necessary.

The third part of HighGUI is the window system (or GUI). The library provides some simple functions that allow us to open a window and throw an image into that window. It also allows us to register and respond to mouse and keyboard events on that window. These features are most useful when trying to get off of the ground with a simple application. Tossing in some slider bars, we find ourselves able to prototype a surprising variety of applications using only the HighGUI library. If we want to link to Qt, we can even get a little more functionality.[2]

---

[1] Some lower level file system operations are located in the core module as well.

[2] This is Qt the cross-platform widget toolkit. We will talk more about how this works later in this chapter.

As we proceed in this chapter, we will not treat these three segments separately; rather, we will start with some functions of highest immediate utility and work our way to the subtler points thereafter. In this way, you will learn what you need to get going as soon as possible.

At the end of the chapter, we will consider additional functions that will allow us to mark up images by drawing lines or other shapes on them. These functions are primarily provided for debugging and validating code, but will turn out to be broadly useful. Finally, we will look at more advanced data persistence for OpenCV data types that will allow us to store and reload additional data types other than just images.

Working with Image Files

OpenCV provides special functions for the loading and saving of images that deal, either directly or implicitly, with the complexities associated with compression and decompression of that image data. These functions are different from the more universal XML/YML-based functions discussed in the previous chapter in several respects. The primary distinction is that because these functions are designed for actual images, as opposed to general arrays of data, they rely heavily on existing backends for compression and decompression, handling all of the common file formats, each in the manner required by that file type. Most of these compression and decompression schemes have been developed with the idea that it is possible to lose some information without degrading the visual experience of the image. Clearly such lossy compression schemes are not a good idea for arrays of nonimage data. Less obviously, artifacts introduced by lossy compressions schemes can also cause headaches for computer vision algorithms as well. In many cases, algorithms will find and respond to compression artifacts that are completely invisible to us humans.

The key difference to remember is that the loading and saving functions we will discuss here are really an interface to the resources for handling image files that are already present in your operating system or its available libraries. The XML/YML data persistence mentioned here is entirely intrinsic to OpenCV.

Loading and Saving Images

The most common tasks we will need to accomplish are the loading and the saving of files from disk. The easiest way to do this is with the high-level functions `cv::imread()` and `cv::imwrite()`. These functions handle the complete task of decompression and compression as well as the actual interaction with the filesystem.

Reading Files with `cv::imread()`

Obviously the first thing to do is to learn how to get an image out of the filesystem and into our program. The function that does this is `cv::imread()`:

```
cv::Mat cv::imread(
  const string& filename,                       // Input filename
  int           flags   = cv::LOAD_IMAGE_COLOR  // Flags set how to interpret file
);
```

When opening an image, `cv::imread()` does not look at the file extension. Instead, `cv::imread()` analyzes the first few bytes of the file (aka its *signature* or "magic sequence") and determines the appropriate codec using that. The second argument `flags` can be set to one of several values. By default, `flags` is set to `cv::IMREAD_COLOR`. This value indicates that images are to be loaded as three-channel images with 8 bits per channel. In this case, even if the image is actually grayscale in the file, the resulting image in memory will still have three channels, with all of the channels containing identical information. Alternatively, if `flags` is set to `cv::IMREAD_GRAYSCALE`, the image will be loaded as grayscale, regardless of the number of channels in the file. The final option is to set `flags` to

cv::IMREAD_ANYCOLOR. In this case, the image will be loaded "as is," with the result being three-channel if the file is color, and one-channel if the file is grayscale.[3]

In addition to the color-related flags, cv::imread() supports the flag cv::IMREAD_ANYDEPTH, which indicates that if an input image channels have more than 8 bits, that it should be loaded without conversion (i.e., the allocated array will be of the type indicated in the file.)

Table 4-1: Parameters accepted by cv::imread()

Parameter ID	Meaning	Default
cv::IMREAD_COLOR	*Always load to three-channel array.*	yes
cv::IMREAD_GRAYSCALE	*Always load to single-channel array.*	no
cv::IMREAD_ANYCOLOR	*Channels as indicated by file (up to three).*	no
cv::IMREAD_ANYDEPTH	*Allow loading of more than 8-bit depth.*	no
cv::IMREAD_UNCHANGED	*Equivalent to combining:* cv::LOAD_IMAGE_ANYCOLOR | cv::LOAD_IMAGE_UNCHANGED	no

cv::imread() does not give a runtime error when it fails to load an image; it simply returns an empty cv::Mat (i.e., empty()==true).

Writing Files with `cv::imwrite()`

The obvious complementary function to cv::imread() is cv::imwrite(), which takes three arguments:

```
int cv::imwrite(
  const string&      filename,           // Input filename
  cv::InputArray     image,              // Image to write to file
  const vector<int>& params  = vector<int>() // (Optional) for parameterized formats
);
```

The first argument gives the filename, whose extension is used to determine the format in which the file will be stored. The second argument is the image to be stored. The third argument is used for parameters that are accepted by the particular file type being used for the write operation. The params argument expects an STL vector of integers, with those integers being a sequence of parameter IDs followed by the value to be assigned to that parameter (i.e., alternating between the parameter ID and the parameter value). For the parameter IDs, there are aliases provided by OpenCV, as listed in *Table 4-2*.

Table 4-2: Parameters accepted by cv::imwrite()

Parameter ID	Meaning	Range	Default
cv::IMWRITE_JPG_QUALITY	*JPEG quality*	0–100	95

[3] At the time of writing, "as is" does not support the loading of a fourth channel for those file types that support alpha channels. In this case, the fourth channel will currently just be ignored and the file will be treated as if it had only three channels.

cv::IMWRITE_PNG_COMPRESSION	*PNG compression (higher values mean more compression)*	0-9	3
cv::IMWRITE_PXM_BINARY	*Use binary format for PPM, PGM, or PBM files*	0 or 1	1

The `cv::imwrite()` function will store only 8-bit single- or three-channel images for most file formats. Backends for flexible image formats like PNG, TIFF, or JPEG 2000 allow storing 16-bit or even float formats and some allow four-channel images (BGR plus alpha) as well. The return value will be `1` if the save was successful and should be `0` if the save was not.[4]

A Note about Codecs

Remember, however, that `cv::imwrite()` is intended for images, and relies heavily on software libraries intended for handling image file types. These libraries are generically referred to as *codecs* ("*co*-mpression and *dec*-ompression librarie-*s*"). Your operating system will likely have many codecs available, with (at least) one being available for each of the different common file types.

OpenCV comes with the codecs you will need for some file formats (JPEG, PNG, and TIFF), and on Microsoft Windows as well as on Apple Mac OS X, these should be available on installation. On Linux and other Unix-like operating systems, OpenCV will try to use codecs supplied with the OS image. In this case, you will need to ensure you have installed the relevant packages with their development files (i.e., install *libjpeg-dev* in addition to *libjpeg* on Linux).

> In Apple's Mac OS X, it is also possible to use the native image readers from Mac OS X. If you do this, however, you should be aware that Mac OS X has an embedded native color management, and so the colors that appear in the loaded file may be different than if you had used the codecs included with the OpenCV package. This is a good example of how the reliance on external elements by `cv::imread()` and `cv::imwrite()` can have important and sometimes unexpected consequences.

Compression and Decompression

As already mentioned, the `cv::imread()` and `cv::imwrite` functions are high-level tools that handle a number of separate things necessary to ultimately get your image written to or read from the disk. In practice, it is often useful to be able to handle some of those subcomponents individually and, in particular, to be able to compress or decompress an image in memory (using the codecs we just reviewed).

Compressing Files with `cv::imencode()`

Images can be compressed directly from OpenCV's array types. In this case, the result will not be an array type, but rather a simple character buffer. This should not be surprising, as the resulting object is now in some format that is meaningful only to the codec that compressed it, and will (by construction) not be the same size as the original image.

```
void cv::imencode(
  const string&      ext,                // Extension specifies codec
  cv::InputArray     img,                // Image to be encoded
  vector<uchar>&     buf,                // Encoded file bytes go here
  const vector<int>& params  = vector<int>() // (Optional) for parameterized formats
```

[4] The reason we say "should" is that, in some OS environments, it is possible to issue save commands that will actually cause the operating system to throw an exception. Normally, however, a zero value will be returned to indicate failure.

```
);
```

The first argument to `cv::imencode()` is the file extension `ext`, represented as a string, which is associated with the desired compression. Of course, no file is actually written, but the extension is not only an intuitive way to refer to the desired format, it is the actual key used by most operating systems to index the available codecs. The next argument `img` is the image to be compressed, and the argument following that `buf` is the character array into which the compressed image will be placed. The final argument `params` is used to specify any parameters which may be required (or desired) for the specific compression codec to be used. The possible values for `params` are those listed earlier in Table 4-2 in the `cv::imwrite()`.

Uncompressing Files with `cv::imdecode()`

```
cv::Mat cv::imdecode(
  cv::InputArray buf,                    // Encoded file bytes are here
  int            flags = cv::LOAD_IMAGE_COLOR // Flags set how to interpret file
);
```

Just as `cv::imencode()` allows us to compress an image into a character buffer, `cv::imdecode()` allows us to decompress from a character buffer into an image array. `cv::imdecode()` takes just two arguments, the first being the buffer[5] `buf` and the second being the `flags` arguments, which takes the same options as the flags used by `cv::imread()` (see *Table 4-1*). As was the case with `cv::imread()`, `cv::imdecode()` does not need a file extension (as `cv::imencode()` did) because it can deduce the correct codec to use from the first bytes of the compressed image in the buffer.

Just as `cv::imread()` returns an empty (`cv::Mat::empty()==true`) array if it cannot read the file it is given, `cv::imdecode()` returns an empty array if the buffer it is given is empty, contains invalid or unusable data, and so on.

Working with Video

When working with video, we must consider several functions, including (of course) how to read and write video files. You will also probably find yourself thinking shortly thereafter about how to actually play back such files on the screen—either for debugging, or as the final output of our program; we'll get to that next,

Reading Video with the `cv::VideoCapture` Object

The first thing we need is the `cv::VideoCapture` object. This is another one of those "objects that do stuff" we encountered in the previous chapter. This object contains the information needed for reading frames from a camera or video file. Depending on the source, we use one of three different calls to create a `cv::VideoCapture` object:

```
cv::VideoCapture::VideoCapture(
  const string&      filename,           // Input filename
);
cv::VideoCapture::VideoCapture(
  int device                             // Video Capture device id
);
cv::VideoCapture::VideoCapture();
```

In the case of the first constructor, we can simply give a filename for a video file (.MPG, .AVI, etc.) and OpenCV will open the file and prepare to read it. If the open is successful and we are able to start reading frames, the member function `cv::VideoCapture::isOpen()` will return `true`.

[5] You should not be surprised that the input `buf` is not type `vector<int>&`, as it was with `cv::imencode()`. Recall that the type `cv::InputArray` covers many possibilities and `vector<>` is one of them.

A lot of people don't always check these sorts of things, thinking that nothing will go wrong. Don't do that here. The returned value of `cv::VideoCapture::isOpen()` will be `false` if for some reason the file could not be opened (e.g., if the file does not exist), but that is not the only possible cause. The constructed object will also not be ready to be used if the codec with which the video is compressed is not known. As with the image codecs, you will need to have the appropriate library already residing on your computer in order to successfully read the video file. This is why it is always important for your code to check the return value of `cv::VideoCapture::isOpen()`, because even if it works on one machine (where the needed codec DLL or shared library is available), it might not work on another machine (where that codec is missing). Once we have an open `cv::VideoCapture` object, we can begin reading frames and do a number of other things. But before we get into that, let's take a look at how to capture images from a camera.

The variant of `cv::VideoCapture::VideoCapture()` that takes an integer `device` argument works very much like the `string` version we just discussed, except without the headache from the codecs.[6] In this case, we give an *identifier* that indicates a camera we would like to access and how we expect the operating system to talk to that camera. For the camera, this is just an identification number that is zero (0) when we only have one camera and increments upward when there are multiple cameras on the same system. The other part of the identifier is called the *domain* of the camera and indicates (in essence) what type of camera we have. The domain can be any of the predefined constants shown in Table 4-3.

Table 4-3: Camera "domain" indicates where HighGUI should look for your camera

Camera capture constant	Numerical value
`cv::CAP_ANY`	0
`cv::CAP_MIL`	100
`cv::CAP_VFW`	200
`cv::CAP_V4L`	200
`cv::CAP_V4L2`	200
`cv::CAP_FIREWIRE`	300
`cv::CAP_IEEE1394`	300
`cv::CAP_DC1394`	300
`cv::CAP_CMU1394`	300
`cv::CAP_QT`	500
`cv::CAP_DSHOW`	700
`cv::CAP_PVAPI`	800
`cv::CAP_OPENNI`	900
`cv::CAP_ANDROID`	1000
…	

When we construct the `device` argument for `cv::VideoCapture::VideoCapture()`, we pass in an identifier that is just the sum of the domain index and the camera index. For example:

```
cv::VideoCapture capture( cv::CAP_FIREWIRE );
```

In this example, `cv::VideoCapture::VideoCapture()` will attempt to open the first (i.e., number-zero) FireWire camera. In most cases, the domain is unnecessary when we have only one camera; it is sufficient to use `cv::CAP_ANY` (which is conveniently equal to 0, so we don't even have to type that in). One last useful hint before we move on: on some platforms, you can pass `-1` to

[6] Of course, to be completely fair, we should probably confess that the headache caused by different codecs has been replaced by the analogous headache of determining which cameras are (or are not) supported on our system.

`cv::VideoCapture::VideoCapture()`, which will cause OpenCV to open a window that allows you to select the desired camera.

Your last option is to create the capture object without providing any information about what is to be opened.

```
cv::VideoCapture cap();
cap.open( "my_video.avi" );
```

In this case, the capture object will be there, but not ready for use until you explicitly open the source you want to read from. This is done with the `cv::VideoCapture::open()` method which, like the `cv::VideoCapture` constructor, can take either an STL string or a device ID as arguments. In either case, `cv::VideoCapture::open()` will have exactly the same effect as calling the `cv::VideoCapture` constructor with the same argument.

Reading Frames with `cv::VideoCapture::read()`

```
bool cv::VideoCapture::read(
  cv::Mat& image                          // Image into which to read data
);
```

Once you have a `cv::VideoCapture` object, you can start reading frames. The easiest way to do this is to call `cv::VideoCapture::read()`, which will simply go to the open file represented by `cv::VideoCapture` and get the next frame, inserting that frame into the provided array `image`. This action will automatically "advance" the video capture object such that a subsequent call to `cv::VideoCapture::read()` will return the next frame, and so on.

If the read was not successful (e.g., if you have reached the end of your file), then this function call will return `false` (otherwise it will return `true`). Similarly, the array object you supplied to the function will also be empty.

Reading Frames with `cv::VideoCapture::operator>>()`

```
cv::VideoCapture& cv::VideoCapture::operator>>(
  cv::Mat& image                          // Image into which to read data
);
```

In addition to using the read method of `cv::VideoCapture`, you can use the overloaded function `cv::VideoCapture::operator>>()` (i.e., the "stream read" operator) to read the next frame from your video capture object. In this case, `cv::VideoCapture::operator>>()` behaves exactly the same as `cv::VideoCapture::read()`, except that because it is a stream operator, it returns a reference to the original `cv::VideoCapture::read()` object, whether or not it succeeded. In this case, you must check that the return array is not empty.

Reading Frames with `cv::VideoCapture::grab()` and `cv::VideoCapture::retrieve()`

Instead of taking images one at a time from your camera or video source and decoding them as you read them, it is possible to break this process down into a *grab* phase, which is a little more than a memory copy, and a *retrieve* phase, which handles the actual decoding of the grabbed data.

```
bool cv::VideoCapture::grab( void );
bool cv::VideoCapture::retrieve(
  cv::Mat& image,                         // Image into which to read data
  int      channel = 0                    // Used for multihead devices
);
```

The `cv::VideoCapture::grab()` function copies the currently available image to an internal buffer that is invisible to the user. Why would you want OpenCV to put the frame somewhere you can't access it? The answer is that this grabbed frame is unprocessed, and `grab()` is designed simply to get it onto the computer (typically from a camera) as quickly as possible.

> There are many reasons to grab and retrieve separately rather than together as would be the case in calling `cv::VideoCapture::read()`. The most common situation

arises when there are multiple cameras (e.g., with stereo imaging). In this case, it is important to have frames that are separated in time by the minimum amount possible (ideally they would be simultaneous for stereo imaging). Therefore, it makes the most sense to first grab all the frames and then come back and decode them after you have them all safely in your buffers.

As was the case with `cv::VideoCapture::read()`, `cv::VideoCapture::grab()` returns `true` only if the grab was successful.

Once you have grabbed your frame, you can call `cv::VideoCapture::retrieve()`, which handles the decoding and the allocation and copying necessary to return the frame to you as a `cv::Mat` array. The function `cv::VideoCapture::retrieve()` functions analogously to `cv::VideoCapture::read()`, except that it operates from the internal buffer to which `cv::VideoCapture::grab()` copies frames. The other important difference between `cv::VideoCapture::read()` and `cv::VideoCapture::retrieve()` is the additional argument `channel`. The `channel` argument is used when the device being accessed natively has multiple "heads" (i.e., multiple imagers). This is typically the case for devices designed specifically to be stereo imagers, as well as slightly more exotic devices such as the *Kinect*.[7] In these cases, the value of channel will indicate which image from the device is to be retrieved. In these cases, you would call `cv::VideoCapture::grab()` just once and then call `cv::VideoCapture::retrieve()` as many times as needed to retrieve all of the images in the camera (each time with a different value for `channel`).

Camera Properties: `cv::VideoCapture::get()` and `cv::VideoCapture::set()`

Video files contain not only the video frames themselves, but also important metadata, which can be essential for handling the files correctly. When a video file is opened, that information is copied into the `cv::VideoCapture` object's internal data area. It is very common to want to read that information from the `cv::VideoCapture` object, and sometimes also useful to write to that data area ourselves. The `cv::VideoCapture::get()` and `cv::VideoCapture::set()` functions allow us to perform these two operations:

```
double cv::VideoCapture::get(
  int    propid                          // Property identifier (see Table 4-4)
);

bool cv::VideoCapture::set(
  int    propid                          // Property identifier (see Table 4-4)
  double value                           // Value to which to set the property
);
```

The routine `cv::VideoCapture::get()` accepts any of the property IDs shown in Table 4-4[8].

Table 4-4: Video capture properties used by cv::VideoCapture::get() and cv::VideoCapture::set()

Video capture property	Camera Only	Meaning
`cv::CAP_PROP_POS_MSEC`		Current position in video file (milliseconds) or video

[7] The currently supported multihead cameras are Kinect and Videre; others may be added later.

[8] It should be understood that not all of the properties recognized by OpenCV will be recognized or handled by the "backend" behind the capture. For example, the capture mechanisms operating behind the scenes on Android, Firewire on Linux (via dc1394), Quicktime, or a Kinect (via OpenNI) are all going to be very different, and not all of them will offer all of the services implied by this long list of options. Expect this list to grow as well, as new system types make new options possible.

		capture timestamp
`cv::CAP_PROP_POS_FRAMES`		Zero-based index of next frame
`cv::CAP_PROP_POS_AVI_RATIO`		Relative position in the video (range is 0.0 to 1.0)
`cv::CAP_PROP_FRAME_WIDTH`		Width of frames in the video
`cv::CAP_PROP_FRAME_HEIGHT`		Height of frames in the video
`cv::CAP_PROP_FPS`		Frame rate at which the video was recorded
`cv::CAP_PROP_FOURCC`		Four character code indicating codec
`cv::CAP_PROP_FRAME_COUNT`		Total number of frames in a video file
`cv::CAP_PROP_FORMAT`		Format of the Mat objects returned (e.g., CV::U8C3)
`cv::CAP_PROP_MODE`		Indicates capture mode, values are specific to video backend being used (i.e., DC1394, etc.)
`cv::CAP_PROP_BRIGHTNESS`	✓	Brightness setting for camera (when supported)
`cv::CAP_PROP_CONTRAST`	✓	Contrast setting for camera (when supported)
`cv::CAP_PROP_SATURATION`	✓	Saturation setting for camera (when supported)
`cv::CAP_PROP_HUE`	✓	Hue setting for camera (when supported)
`cv::CAP_PROP_GAIN`	✓	Gain setting for camera (when supported)
`cv::CAP_PROP_EXPOSURE`	✓	Exposure setting for camera (when supported)
`cv::CAP_PROP_CONVERT_RGB`	✓	If nonzero, captured images will be converted to have three channels
`cv::CAP_PROP_WHITE_BALANCE`	✓	White balance setting for camera (when supported)
`cv::CAP_PROP_RECTIFICATION`	✓	Rectification flag for stereo cameras (DC1394-2.x only)

Most of these properties are self-explanatory. `POS_MSEC` is the current position in a video file, measured in milliseconds. `POS_FRAME` is the current position in frame number. `POS_AVI_RATIO` is the position given as a number between `0.0` and `1.0`. (This is actually quite useful when you want to position a trackbar to allow folks to navigate around your video.) `FRAME_WIDTH` and `FRAME_HEIGHT` are the dimensions of the individual frames of the video to be read (or to be captured at the camera's current settings). `FPS` is specific to video files and indicates the number of frames per second at which the video was captured; you will need to know this if you want to play back your video and have it come out at the right speed. `FOURCC` is the four-character code for the compression codec to be used for the video you are currently reading (more on these shortly). `FRAME_COUNT` should be the total number of frames in the video, but this figure is not entirely reliable.

All of these values are returned as type `double`, which is perfectly reasonable except for the case of `FOURCC` (FourCC) [FourCC85]. Here you will have to recast the result in order to interpret it, as shown in Example 4-1.

Example 4-1. Unpacking a four-character code to identify a video codec

```
cv::VideoCapture cap( "my_video.avi" );
double           f      = cap.get( cv::CAP_PROP_FOURCC );
char*            fourcc = (char*) (&f);
```

For each of these video capture properties, there is a corresponding `cv::VideoCapture::set()` function that will attempt to set the property. These are not all meaningful things to do; for example, you should not be setting the `FOURCC` of a video you are currently reading. Attempting to move around the video by setting one of the position properties will sometimes work, but only for some video codecs (we'll have more to say about video codecs in the next section).

Writing Video with the cv::VideoWriter Object

The other thing we might want to do with video is writing it out to disk. OpenCV makes this easy; it is essentially the same as reading video but with a few extra details.

Just as we did with the `cv::VideoCapture` device for reading, we must first create a `cv::VideoWriter` device before we can write out our video. The video writer has two constructors; one is a simple default constructor that just creates an uninitialized video object that we will have to open later, and the other has all of the arguments necessary to actually set up the writer.

```
cv::VideoWriter::VideoWriter(
  const string& filename,                // Input filename
  int           fourcc,                  // codec, use CV_FOUR_CC() macro
  double        fps,                     // Frame rate (storred in output file)
  cv::Size      frame_size,              // Size of individual images
  bool          is_color  = true         // if false, you can pass gray frames
);
```

You will notice that the video writer requires a few extra arguments relative to the video reader. In addition to the filename, we have to tell the writer what codec to use, what the frame rate is, and how big the frames will be. Optionally, we can tell OpenCV that the frames are already in color (i.e., three-channel). If you set `isColor` to `false`, you can pass grayscale frames and the video writer will handle them correctly.

As with the video reader, you can also create the video writer with a default constructor, and then configure the writer with the `cv::VideoWriter::open()` method, which takes the same arguments as the full constructor.

```
cv::VideoWriter out();
out.open(
  "my_video.mpg",
  CV_FOUR_CC('D','I','V','X'),  // MPEG-4 codec
  30.0,                         // fps
  cv::Size( 640, 480 ),
  true                          // expect only color frames
);
```

Here, the codec is indicated by its four-character code. (For those of you who are not experts in compression codecs, they all have a unique four-character identifier associated with them). In this case, the `int` that is named `fourcc` in the argument list for `cv::VideoWriter::VideoWriter()` is actually the four characters of the `fourcc` packed together. Since this comes up relatively often, OpenCV provides a convenient macro `CV_FOURCC(c0,c1,c2,c3)` that will do the bit packing for you.

Once you have given your video writer all the information it needs to set itself up, it is always a good idea to ask it if it is ready to go. This is done with the `cv::VideoWriter::isOpened()` method, which will return true if you are good to go. If it returns false, this could mean that you don't have write permission to the directory for the file you indicated, or (most often) that the codec you specified is not available.

> The codecs available to you will depend on your operating system installation and the additional libraries you have installed. For portable code, it is especially important to be able to gracefully handle the case in which the desired codec is not available on any particular machine.

Writing Frames with `cv::VideoWriter::write()`

Once you have verified that your video writer is ready to write, you can write frames by simply passing your array to the writer:

```
cv::VideoWriter::write(
  const Mat& image                      // Image to write as next frame
);
```

This image must have the same size as the size you gave to the writer when you configured it in the first place. If you told the writer that the frames would be in color, this must also be a three-channel image. If you indicated to the writer (via `isColor`) that the images is just one channel, you should supply a single-channel (grayscale) image.

Writing Frames with cv::VideoWriter::operator<<()

The video writer also supports the idiom of the overloaded output stream operator (`operator<<()`). In this case, once you have your video writer open, you can write images to the video stream in the same manner you would write to `cout` or a file `ofstream` object:

```
my_video_writer << my_frame;
```

Working with Windows

The HighGUI toolkit provides some rudimentary built-in features for creating windows, displaying images in those windows, and for making some user interaction possible with those windows. The native OpenCV graphical user interface (GUI) functions have been part of the library for a very long time, and have the advantage of being stable, portable,[9] and very easy to use.

On the other hand, the native GUI features have the disadvantage of being not particularly complete. As a result, there has been an effort to modernize the GUI portion of HighGUI, and to add a number of useful new features. This was done by converting from "native" interfaces to the use of Qt. Qt is a cross-platform toolkit, and so new features can be implemented only once in the library, rather than once each for each of the native platforms. Needless to say, this has the result of making development of the Qt interface more attractive, and so it does more stuff, and will probably grow in the future, leaving the features of the native interface to become static legacy code.

In this section, we will first take a look at the native functions, and then move on to look at the differences, and particularly the new features, offered by the Qt-based interface.

HighGUI Native Graphical User Interface

This section describes the core interface functions that are part of OpenCV and require no external toolkit support. Some of them, when you are using the Qt backend, will also behave somewhat differently or have some additional options, but we will put those details off until the next section.

The native HighGUI user input tools support only two basic interactions. Specifically, mouse clicks on the image area can be responded to, and a simple track bar can be added. These basic functions are usually sufficient for basic mock-ups and debugging, but hardly ideal for end-user facing applications. For that, you will (at least) want to use the Qt-based interface or some other UI toolkit.

The main advantages of the native tools are that they are fast and easy to use, and don't require you to install any additional libraries.

Creating a Window with `cv::namedWindow()`

First, we want to be able to create a window and show an image on the screen using HighGUI. The function that does the first part for us is `cv::namedWindow()`. The function expects a name for the new window and one flag. The name appears at the top of the window, and the name is also used as a handle for

[9] They are "portable" because they make use of native window GUI tools on various platforms. This means X11 on Linux, Cocoa on MacOSX, and the raw Win32 API on Microsoft Windows machines. However, this portability only extends to those platforms for which there is an implementation in the library. There exist platforms on which OpenCV can be used for which there are no available implementations of the HighGUI library (e.g., Android).

the window that can be passed to other HighGUI functions.[10] The `flag` argument indicates if the window should autosize itself to fit an image we put into it. Here is the full prototype:

```
int cv::namedWindow(
  const string&  name,                  // Handle used to identify window
  int            flags = 0              // Used to tell window to autosize
);
```

For now, the only valid options available for `flags` are to set it to `0` (it's the default value), which indicates that users are to be able (and required) to resize the window, or to set it to `cv::WINDOW_AUTOSIZE`.[11] If `cv::WINDOW_AUTOSIZE` is set, then HighGUI resizes the window to fit automatically whenever a new image is loaded, but users cannot resize the window.

Once we create a window, we usually want to put something inside it. But before we do that, let's see how to get rid of the window when it is no longer needed. For this, we use `cv::destroyWindow()`, a function whose only argument is a string: the name given to the window when it was created.

```
int cv::destroyWindow(
  const string&  name,                  // Handle used to identify window
);
```

Drawing an Image with `cv::imshow()`

Now we are ready for what we really want to do, and that is to load an image and to put it into the window where we can view it and appreciate its profundity. We do this via one simple function, `cv::imshow()`:

```
void cv::imshow(
  const string&  name,                  // Handle used to identify window
  cv::InputArray image                  // Image to display in window
);
```

The first argument is the name of the window within which we intend to draw. The second argument is the image to be drawn.

Updating a Window and `cv::waitKey()`

The function `cv::waitKey()` has two functions. The first is the one from which it derives its name, which is to wait for some specified (possibly indefinite) amount of time for a key-press on the keyboard, and to return that key value when it is received. `cv::waitKey()` accepts a key-press from any open OpenCV window (but will not function if no such window exists).

```
int cv::waitKey(
  int delay = 0                         // Milliseconds until giving up (0='never')
);
```

`cv::waitKey()` takes a single argument `delay`, which is the amount of time (in milliseconds) which it will wait for a key-press before returning automatically. If the delay is set to `0`, `cv::waitKey()` will wait indefinitely for a key-press. If no key-press comes before `delay` milliseconds has passed, `cv::waitKey()` will return `-1`.

The second, less obvious function of `cv::waitKey()` is that it provides an opportunity for any open OpenCV window to be updated. This means that if you do not call `cv::waitKey()`, your image may

[10] In OpenCV, windows are referenced by name instead of by some unfriendly (and invariably OS-dependent) "handle." Conversion between handles and names happens under the hood of HighGUI, so you needn't worry about it.

[11] Later in this chapter, we will look at the (optional) Qt-based backend for HighGUI. If you are using that backend, there are more options available for `cv::namedWindow()` as well as other functions.

never be drawn in your window, or your window may behave strangely (and badly) when moved, resized, or uncovered.[12]

An Example Displaying an Image

Let's now put together a simple program that will display an image on the screen. We can read a filename from the command line, create a window, and put our image in the window in 15 lines (including comments!). This program will display our image as long as we want to sit and appreciate it, and then exit when the `ESC`-key (ASCII value of `27`) is pressed.

Example 4-2. A simple example of creating a window and displaying an image in that window

```
int main( int argc, char** argv ) {

  // Create a named window with the name of the file
  cv::namedWindow( argv[1], 1 );

  // Load the image from the given filename
  cv::Mat = cv::imread( argv[1] );

  // Show the image in the named window
  cv::imshow( argv[1], img );

  // Idle until the user hits the "Esc" key
  while( true ) {
    if( cv::waitKey( 100 ) == 27 ) break;
  }

  // Clean up and don't be piggies
  cv::destroyWindow( argv[1] );

  exit(0);
}
```

For convenience, we have used the filename as the window name. This is nice because OpenCV automatically puts the window name at the top of the window, so we can tell which file we are viewing (see Figure 4-1). Easy as cake.

[12] What this sentence really means is that `cv::waitKey()` is the *only* function in HighGUI that can fetch and handle events. This means that if it is not called periodically, no normal event processing will take place. As a corollary to this, if HighGUI is being used within an environment that takes care of event processing, then you may not need to call `cv::waitKey()`.

Figure 4-1: A simple image displayed with `cv::imshow()`

Before we move on, there are a few other window-related functions you ought to know about. They are:

```
void cv::moveWindow( const char* name, int x, int y );
void cv::destroyAllWindows( void );
int  cv::startWindowThread( void );
```

`cv::moveWindow()` simply moves a window on the screen so that its upper-left corner is positioned at `x,y`. `cv::destroyAllWindows()` is a useful cleanup function that closes all of the windows and deallocates the associated memory.

On Linux and MacOS, `cv::startWindowThread()` tries to start a thread that updates the window automatically and handles resizing and so forth. A return value of `0` indicates that no thread could be started—for example, because there is no support for this feature in the version of OpenCV that you are using. Note that, if you do not start a separate window thread, OpenCV can react to user interface actions only when it is explicitly given time to do so (this happens when your program invokes `cv::waitKey()`).

Mouse Events

Now that we can display an image to a user, we might also want to allow the user to interact with the image we have created. Since we are working in a window environment and since we already learned how to capture single keystrokes with `cv::waitKey()`, the next logical thing to consider is how to "listen to" and respond to mouse events.

Unlike keyboard events, mouse events are handled by a more typical callback mechanism. This means that, to enable response to mouse clicks, we must first write a callback routine that OpenCV can call whenever a mouse event occurs. Once we have done that, we must register the callback with OpenCV, thereby informing OpenCV that this is the correct function to use whenever the user does something with the mouse over a particular window.

Let's start with the callback. For those of you who are a little rusty on your event-driven program lingo, the *callback* can be any function that takes the correct set of arguments and returns the correct type. Here, we must be able to tell the function to be used as a callback exactly what kind of event occurred and where it occurred. The function must also be told if the user was pressing such keys as Shift or Alt when the mouse event occurred. A pointer to such a function is called a `cv::MouseCallback`. Here is the exact prototype that your callback function must match:

```
void your_mouse_callback(
  int   event,                          // Event type (see Table 4-5)
  int   x,                              // x-location of mouse event
  int   y,                              // y-location of mouse event
  int   flags,                          // More details on event (see Table 4-6)
```

```
  void* param                          // Parameters from cv::setMouseCallback()
);
```

Now, whenever your function is called, OpenCV will fill in the arguments with their appropriate values. The first argument, called the `event`, will have one of the values shown in Table 4-5.

Table 4-5: Mouse event types

Event	Numerical value
`cv::EVENT_MOUSEMOVE`	0
`cv::EVENT_LBUTTONDOWN`	1
`cv::EVENT_RBUTTONDOWN`	2
`cv::EVENT_MBUTTONDOWN`	3
`cv::EVENT_LBUTTONUP`	4
`cv::EVENT_RBUTTONUP`	5
`cv::EVENT_MBUTTONUP`	6
`cv::EVENT_LBUTTONDBLCLK`	7
`cv::EVENT_RBUTTONDBLCLK`	8
`cv::EVENT_MBUTTONDBLCLK`	9

The second and third arguments will be set to the `x` and `y` coordinates of the mouse event. It is worth noting that these coordinates represent the pixel coordinates in the image independent of the size of the window.[13]

The fourth argument, called `flags`, is a bit field in which individual bits indicate special conditions present at the time of the event. For example, `cv::EVENT_FLAG_SHIFTKEY` has a numerical value of `16` (i.e., the fifth bit, or `1<<4`) and so, if we wanted to test whether the shift key were down, we could simply compute the bitwise AND of `flags & cv::EVENT_FLAG_SHIFTKEY`. Table 4-6 shows a complete list of the flags.

Table 4-6: Mouse event flags

Flag	Numerical value
`cv::EVENT_FLAG_LBUTTON`	1
`cv::EVENT_FLAG_RBUTTON`	2
`cv::EVENT_FLAG_MBUTTON`	4
`cv::EVENT_FLAG_CTRLKEY`	8
`cv::EVENT_FLAG_SHIFTKEY`	16
`cv::EVENT_FLAG_ALTKEY`	32

The final argument is a void pointer that can be used to have OpenCV pass additional information in the form of a pointer to whatever kind of structure you need.[14]

[13] In general, this is not the same as the pixel coordinates of the event that would be returned by the OS. This is because OpenCV is concerned with telling you where *in the image* the event happened, not *in the window* (to which the OS typically references mouse event coordinates).

[14] A common situation in which you will want to use the `param` argument is when the callback itself is a static member function of a class. In this case, you will probably find yourself wanting to pass the `this` pointer and so indicate which class object instance the callback is intended to affect.

Next, we need the function that registers the callback. That function is called `cv::setMouseCallback()`, and it requires three arguments.

```
void cv::setMouseCallback(
  const string&     windowName,           // Handle used to identify window
  cv::MouseCallback on_mouse,             // Callback function
  void*             param     = NULL      // Additional parameters for callback fn.
);
```

The first argument is the name of the window to which the callback will be attached. Only events in that particular window will trigger this specific callback. The second argument is your callback function. Finally, the third `param` argument allows us to specify the `param` information that should be given to the callback whenever it is executed. This is, of course, the same `param` we were just discussing with the callback prototype.

In Example 4-3, we write a small program to draw boxes on the screen with the mouse. The function `my_mouse_callback()` responds to mouse events, and it uses the event to determine what to do when it is called.

Example 4-3. Toy program for using a mouse to draw boxes on the screen

```
#include <opencv2/opencv.hpp>

// Define our callback which we will install for
// mouse events
//
void my_mouse_callback(
   int event, int x, int y, int flags, void* param
);

Rect box;
bool drawing_box = false;

// A litte subroutine to draw a box onto an image
//
void draw_box( cv::Mat& img, cv::Rect box ) {
  cv::rectangle(
    img,
    box.tl(),
    box.br(),
    cv::Scalar(0x00,0x00,0xff)    /* red */
  );
}

void help() {
  std::cout << "Call: ./ch4_ex4_1\n" <<
    " shows how to use a mouse to draw regions in an image." << std::endl;
}

int main( int argc, char** argv ) {

  help();
  box = cv::Rect(-1,-1,0,0);
  cv::Mat image(200, 200, CV::U8C3), temp;
  image.copyTo(temp);

  box   = cv::Rect(-1,-1,0,0);
  image = cv::Scalar::all(0);

  cv::namedWindow( "Box Example" );

  // Here is the crucial moment where we actually install
  // the callback. Note that we set the value of 'params' to
```

```
  // be the image we are working with so that the callback
  // will have the image to edit.
  //
  cv::setMouseCallback(
    "Box Example",
    my_mouse_callback,
    (void*)&image
  );

  // The main program loop. Here we copy the working image
  // to the temp image, and if the user is drawing, then
  // put the currently contemplated box onto that temp image.
  // Display the temp image, and wait 15ms for a keystroke,
  // then repeat.
  //
  for(;;) {

    image.copyTo(temp);
    if( drawing_box ) draw_box( temp, box );
    cv::imshow( "Box Example", temp );

    if( cv::waitKey( 15 ) == 27 ) break;
  }

  return 0;
}

// This is our mouse callback. If the user
// presses the left button, we start a box.
// when the user releases that button, then we
// add the box to the current image. When the
// mouse is dragged (with the button down) we
// resize the box.
//
void my_mouse_callback(
   int event, int x, int y, int flags, void* param
) {

  cv::Mat& image = *(cv::Mat*) param;

  switch( event ) {
    case cv::EVENT_MOUSEMOVE: {
      if( drawing_box ) {
        box.width  = x-box.x;
        box.height = y-box.y;
      }
    }
    break;
    case cv::EVENT_LBUTTONDOWN: {
      drawing_box = true;
      box = cv::Rect( x, y, 0, 0 );
    }
    break;
    case cv::EVENT_LBUTTONUP: {
      drawing_box = false;
      if( box.width < 0  ) {
        box.x += box.width;
        box.width *= -1;
      }
      if( box.height < 0 ) {
        box.y += box.height;
        box.height *= -1;
      }
```

```
      draw_box( image, box );
    }
    break;
  }
}
```

Sliders, Trackbars, and Switches

HighGUI provides a convenient slider element. In HighGUI, sliders are called *trackbars*. This is because their original (historical) intent was for selecting a particular frame in the playback of a video. Of course, once added to HighGUI, people began to use trackbars for all of the usual things one might do with a slider as well as many unusual ones (we'll discuss some of these in the next section, "No Buttons").

As with the parent window, the slider is given a unique name (in the form of a character string) and is thereafter always referred to by that name. The HighGUI routine for creating a trackbar is:

```
int cv::createTrackbar(
  const string&        trackbarName,      // Handle used to identify trackbar, label
  const string&        windowName,        // Handle used to identify window
  int*                 value,             // Slider position gets put here
  int                  count,             // Total counts for slider at far right
  cv::TrackbarCallback onChange    = NULL,// Callback function (optional)
  void*                param       = NULL // Additional parameters for callback fn.
);
```

The first two arguments are the name for the trackbar itself and the name of the parent window to which the trackbar will be attached. When the trackbar is created, it is added to either the top or the bottom of the parent window;[15] it will not occlude any image that is already in the window, rather it will make the window slightly bigger. The name of the trackbar will appear as a "label" for the trackbar. As with the location of the trackbar itself, the exact location of this label depends on the operating system, but most often it is immediately to the left.

Figure 4-2: A simple application displaying an image; this window has two trackbars attached named "Trackbar0" and "Trackbar1"

[15] Whether it is added to the top or bottom depends on the operating system, but it will always appear in the same place on any given platform.

The next two arguments are `value`, a pointer to an integer that will be set automatically to the value to which the slider has been moved, and `count`, a numerical value for the maximum value of the slider.

The last argument is a pointer to a callback function that will be automatically called whenever the slider is moved. This is exactly analogous to the callback for mouse events. If used, the callback function must have the form specified by `cv::TrackbarCallback`, which means that it must match the following prototype:

```
void your_trackbar_callback(
  int   pos,                          // Trackbar slider position
  void* param = NULL                  // Parameters from cv::setTrackbarCallback()
);
```

This callback is not actually required, so if you don't want a callback, then you can simply set this value to `NULL`. Without a callback, the only effect of the user moving the slider will be the value of `*value` being updated when the slider is moved. (Of course, if you don't have a callback, you will be responsible for polling this value if you are going to respond to it being changed.)

The final argument to `cv::createTrackbar()` is `params`, which can be any pointer. This pointer will be passed to your callback as its `params` argument whenever the callback is executed. This is very helpful, among other things, for allowing you to handle trackbar events without having to introduce global variables.

Finally, here are two more routines that will allow you to programmatically read or set the value of a trackbar just by using its name:

```
int cv::getTrackbarPos(
  const string& trackbarName,         // Handle used to identify trackbar, label
  const string& windowName,           // Handle used to identify window
);

void cv::setTrackbarPos(
  const string& trackbarName,         // Handle used to identify trackbar, label
  const string& windowName,           // Handle used to identify window
  int   pos                           // Trackbar slider position
);
```

These functions allow you to read or set the value of a trackbar from anywhere in your program.

No Buttons

Unfortunately, the native interface in HighGUI does not provide any explicit support for buttons. It is thus common practice, among the particularly lazy,[16] to instead use sliders with only two positions. Another option that occurs often in the OpenCV samples in *…/opencv/samples/c/* is to use keyboard shortcuts instead of buttons (see, e.g., the *floodfill* demo in the OpenCV source-code bundle).

Switches are just sliders (trackbars) that have only two positions, "on" (1) and "off" (0) (i.e., `count` has been set to `1`). You can see how this is an easy way to obtain the functionality of a button using only the available trackbar tools. Depending on exactly how you want the switch to behave, you can use the trackbar callback to automatically reset the button back to 0 (as in Example 4-4; this is something like the standard behavior of most GUI "buttons") or to automatically set other switches to 0 (which gives the effect of a "radio button").

[16] For the less lazy, the common practice was to compose the image you are displaying with a "control panel" you have drawn and then use the mouse event callback to test for the mouse's location when the event occurs. When the (x, y) location is within the area of a button you have drawn on your control panel, the callback is set to perform the button action. In this way, all "buttons" are internal to the mouse event callback routine associated with the parent window. Really however, if you need this kind of functionality now, it is probably just best to use the Qt backend.

Example 4-4. Using a trackbar to create a "switch" that the user can turn on and off; this program plays a video, and uses the switch to create a pause functionality

```
// An example program in which the user can draw boxes on the screen.
//
#include <opencv2/opencv.hpp>
#include <iostream>

using namespace std;

//
// Using a trackbar to create a "switch" that the user can turn on and off.
// We make this value global so everyone can see it.
//
int g_switch_value = 1;
void switch_off_function() { cout << "Pause\n"; }; //YOU COULD DO MORE WITH THESE FUNCTIONS
void switch_on_function()  { cout << "Run\n"; };

// This will be the callback that we give to the trackbar.
//
void switch_callback( int position, void* ) {
  if( position == 0 ) {
    switch_off_function();
  } else {
    switch_on_function();
  }
}

void help() {
    cout << "Call: my.avi" << endl;
    cout << "Shows putting a pause button in a video." << endl;
}

int main( int argc, char** argv ) {

  cv::Mat frame; // To hold movie images
  cv::VideoCapture g_capture;
  help();
  if( argc < 2 || !g_capture.open( argv[1] ) ){
    cout << "Failed to open " << argv[1] << " video file\n" << endl;
    return -1;
  }

  // Name the main window
  //
  cv::namedWindow( "Example", 1 );

  // Create the trackbar. We give it a name,
  // and tell it the name of the parent window.
  //
  cv::createTrackbar(
    "Switch",
    "Example",
    &g_switch_value,
    1,
    switch_callback
  );

  // This will just cause OpenCV to idle until
  // someone hits the "Escape" key.
  //
  for(;;) {
      if( g_switch_value ) {
```

```
            g_capture >> frame;
            if( frame.empty() ) break;
            cv::imshow( "Example", frame);
        }
        if( cv::waitKey(10)==27 ) break;
    }

    return 0;
}
```

You can see that this will turn on and off just like a light switch. In our example, whenever the trackbar "switch" is set to 0, the callback executes the function `switch_off_function()`, and whenever it is switched on, the `switch_on_function()` is called.

Working with the Qt Backend

As we described earlier, the trend in the development of the HighGUI portion of OpenCV is to rely increasingly on a separate library for GUI functionality. This makes sense, as it is not OpenCV's purpose to reinvent that particular wheel, and there are plenty of great GUI toolkits out there that are well maintained and which evolve and adapt to the changing times under their own development teams.

From the perspective of the OpenCV library, there is a great advantage to using such an outside toolkit. Functionality is gained, and at the same time, development time is reduced (which would otherwise be taken away from the core goal of the library).

However, an important clarification is that using HighGUI with the Qt backend is not the same as using Qt directly (we will explore this possibility briefly at the end of this chapter). The HighGUI interface is still the HighGUI interface; it simply uses Qt behind the scenes in place of the various native libraries. One side effect of this is that it is not that convenient to extend the Qt interface. If you want more than HighGUI gives you, you are still pretty stuck with writing your own window layer. On the bright side, the Qt interface gives you a lot more to work with, and so perhaps you will not find that extra level of complexity necessary as often.

Getting Started

If you have built your OpenCV installation with Qt support on,[17] opening a window will automatically open a window with two new features. These are the *Toolbar* and the *Status Bar* (see Figure 4-3). These objects come up complete, with all of the contents you see in the figure. In particular, the Toolbar contains buttons which will allow you to pan (the first four arrow buttons), zoom (the next four buttons), save the current image (the ninth button), and a final button which can be used to pop up a properties window (more on this last one a little later).

[17] This means that when you configured the build with `cmake`, you used the `-D WITH_QT=ON` option.

Figure 4-3: This image is displayed with the Qt interface enabled; it shows the Toolbar, the Status Bar, and a text Overlay (in this case, containing the name of the image)

The lower Status Bar in Figure 4-3 contains information about what is under your mouse at any given moment. The x-y location is displayed, as well as the RGB value of the pixel currently pointed at.

All of this you get "for free" just for compiling your code with the Qt interface enabled. If you have compiled with Qt and you *do not* want these decorations, then you can simply add the `cv::GUI_NORMAL` flag when you call `cv::namedWindow()` and they will disappear.[18]

The Actions Menu

As you can see, when you create the window with `cv::GUI_EXTENDED`, you will see a range of buttons in the Toolbar. An alternative to the toolbar, which will always be available whether you use `cv::GUI_EXTENDED` or not, is the *pop-up menu*. This pop-up menu contains the same options as the toolbar and can be made to appear at any time by right-clicking on the image.

[18] There is also a flag CV_GUI_EXTENDED, which (in theory) creates these decorations, but its numerical value is 0x00, so it is the default behavior anyhow.

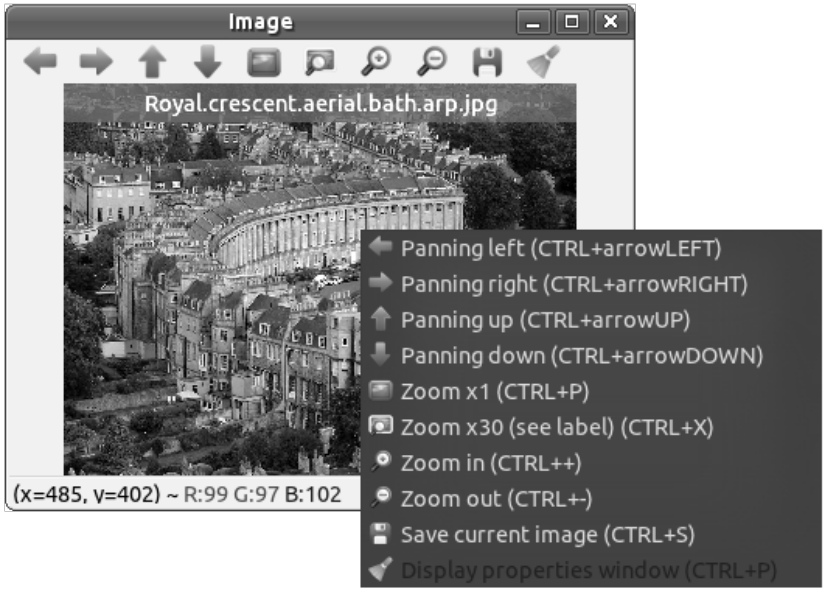


Figure 4-4. Here the Qt extended UI window is shown with the pop-up menu, which provides the same options as the Toolbar (along with an explanation of the buttons and their keyboard shortcuts)

The Text Overlay

Another option provided by the Qt GUI is the ability to put a short-lived banner across the top of the image you are displaying. This banner is called the Overlay, and appears with a shaded box around it for easy reading. This is an exceptionally handy feature if you just want to throw some simple information like the frame number or frame rate on a video, or a filename of the image you are looking at. You can display an overlay on any window, even if you are using `cv::GUI_NORMAL`.

```
int cv::displayOverlay(
  const string& name,                    // Handle used to identify window
  const string& text,                    // Text you want to display
  int           delay                    // Milliseconds to show text (0='forever')
);
```

The function `cv::displayOverlay()` takes three arguments. The first is the name of the window you want the overlay to appear on. The second argument is whatever text you would like to appear in the window. One word of warning here, the text has a fixed size, so if you try to cram too much stuff in there, it will just overflow.[19] By default, the text is always center justified. The third and final argument, `delay`, is the amount of time (in milliseconds) that the overlay will stay in place. If `delay` is set to `0`, then the overlay will stay in place indefinitely (or at least until you write over it with another call to `cv::displayOverlay()`). In general, if you call `cv::displayOverlay()` before the delay timer for a previous call is expired, the previous text is removed and replaced with the new text, and the timer is reset to the new delay value regardless of what is left in the timer before the new call.

Writing Your Own Text into the Status Bar

In addition to the Overlay, you can also write text into the Status Bar. By default, the Status Bar contains information about the pixel over which your mouse pointer is located (if any). You can see in Figure 4-3

[19] You can, however, insert new lines. So, for example, if you were to give the `text` string `"Hello\nWorld"`, then the word "Hello" would appear on the first (top) line, and the word "World" would appear on a second line right below "Hello."

that the status bar contains an x-y location and the RGB color value of the pixel that was under the pointer when the figure was made. You can replace this text with your own text with the `cv::displayStatusBar()` method:

```
int cv::displayStatusBar(
  const string& name,                    // Handle used to identify window
  const string& text,                    // Text you want to display
  int           delay                    // Milliseconds to show text (0='forever')
);
```

Unlike `cv::displayOverlay()`, `cv::displayStatusBar()` can only be used on windows that were created with the `cv::GUI_EXTENDED` flag (i.e., ones that have a Status Bar in the first place). When the `delay` timer is expired (if you didn't set it to `0`), then the default x-y and RGB text will re-appear in the Status Bar.

The Control Panel

You may have noticed in the earlier figures that there is a last button on the Toolbar, corresponding to a last option (which is "darkened" in Figure 4-4) on the pop-up menu which we have not really discussed yet. This option opens up an entirely new window called the *Control Panel*. The control panel is a convenient place to put trackbars and buttons (the Qt GUI does support buttons) that you don't want in your face all of the time. It is important to remember, however, that there is just one Control Panel *per application*, and you do not really *create* it, you just *configure* it.

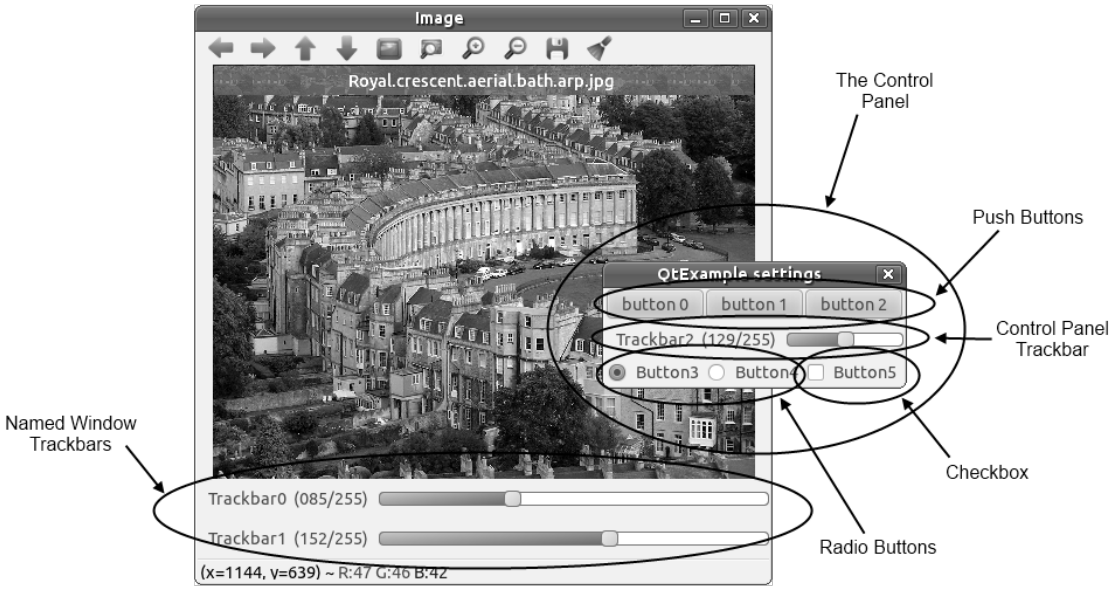


Figure 4-5: In this image, we have added two trackbars to the main window; we also show the Control Panel with three pushbuttons, a trackbar, two radio buttons, and a checkbox

The Control Panel will not be available unless some trackbars or buttons have been assigned to it (more on how to do this momentarily). If it is available, then the Control Panel can be made to appear by pressing the "Display Properties Window" button on the Toolbar (the one on the far right), the identical button on the Action Menu, or pressing `Ctrl+P` while your mouse is over any window.

Trackbars Revisited

In the previous section on the HighGUI native interface, we saw that it was possible to add trackbars to windows. The trackbars in 4 were created using the same `cv::createTrackbar()` command we saw

earlier. The only real difference between the trackbars in the figure, and the ones we created earlier, is that they are prettier (compare with the trackbars using the non-Qt interface in Figure 4-2).

The important new concept in the Qt interface, however, is that we can also create trackbars in the Control Panel. This is done simply by creating the trackbar as you normally would, but by specifying an empty string as the window name to which the trackbar is to be attached.

```
int contrast = 128;
cv::createTrackbar( "Contrast:", "", &contrast, 255, on_change_contrast );
```

For example, this fragment would create a trackbar in the Control Panel that would be labeled "Contrast:", and whose value would start out at `128`, with a maximum value of `255`. Whenever the slider is adjusted, the callback `on_change_contrast()` will be called.

Creating Buttons with `cv::createButton()`

One of the most helpful new capabilities provided by the Qt interface is the ability to create buttons. This includes normal push-buttons, radio style (mutually exclusive) buttons, and checkboxes. All buttons created are always located in the Control Panel.

All three styles of buttons are created with the same method:

```
int cv::createButton(
  const string&      buttonName,          // Handle used to identify trackbar, label
  cv::ButtonCallback onChange     = NULL, // Callback for button event
  void*              params,              // (Optional) parameters for button event
  int                buttonType   = cv::PUSH_BUTTON,  // PUSH_BUTTON or RADIOBOX
  int                initialState = 0     // Start state of the button
);
```

The button expects a name `buttonName` that will appear on or next to the button. If you like, you can neglect this and simply provide an empty string, in which case the button name will be automatically generated in a serialized manner (e.g., "button 0," "button 1," etc.). The second argument `onChange` is a callback that will be called whenever the button is pressed. The prototype for such a callback must match the declaration for `cv::ButtonCallback`, which is:

```
void your_button_callback(
  int   state,                          // Identifies button event type
  void* params                          // Paramaters from cv::createButton()
);
```

When your callback is called as a result of someone pressing a button, it will be given the value `state`, which is derived from what just happened to the button. The pointer `param` that you gave to `cv::createButton()` will also be passed to your callback, filling its `param` argument.

The `buttonType` argument can take one of three values: `cv::PUSH_BUTTON`, `cv::RADIOBOX`, or `cv::CHECKBOX`. The first corresponds to your standard button—you press it, it calls your callback. In the case of the checkbox, the value will be 1 or 0 depending on whether the box was checked or unchecked. The same is true for a radio button, except that when you click a radio button, the callback is called both for the button you just clicked, and for the button that is now unclicked (as a result of the mutex nature of radio buttons). All buttons in the same row (see *button bars* below) are assumed to be part of the same mutex group.

When buttons are created, they are automatically organized into *button bars*. A button bar is a group of buttons that occupies a "row" in the Control Panel. Consider the following code, which generated the Control Panel you see in Figure 4-5.

```
  cv::namedWindow( "Image", cv::GUI_EXPANDED );
  cv::displayOverlay( "Image", file_name, 0 );
  cv::createTrackbar( "Trackbar0", "Image", &mybar0, 255 );
  cv::createTrackbar( "Trackbar1", "Image", &mybar1, 255 );

  cv::createButton( "", NULL, NULL, cv::PUSH_BUTTON );
  cv::createButton( "", NULL, NULL, cv::PUSH_BUTTON );
```

```
cv::createButton( "", NULL, NULL, cv::PUSH_BUTTON );
cv::createTrackbar( "Trackbar2", "", &mybar1, 255 );
cv::createButton( "Button3", NULL, NULL, cv::RADIOBOX, 1 );
cv::createButton( "Button4", NULL, NULL, cv::RADIOBOX, 0 );
cv::createButton( "Button5", NULL, NULL, cv::CHECKBOX, 0 );
```

You will notice that Trackbar0 and Trackbar1 are created in the window called "Image," while Trackbar2 is created in an unnamed window (the Control Panel). The first three `cv::createButton()` calls are not given a name for the button, and you can see in Figure 4-5 the automatically assigned names are placed onto the buttons. You will also notice in Figure 4-5 that the first three buttons are in one row, while the second group of three is on another. This is because of the trackbar.

Buttons are created one after another, each to the right of its predecessor, until (unless) a trackbar is created. Because a trackbar consumes an entire row, it is given its own row below the buttons. If more buttons are created, they will appear on a new row thereafter.[20]

Text and Fonts

Just as the Qt interface allowed for much prettier trackbars and other elements, Qt also allows for much prettier and more versatile text. To write text using the Qt interface, you must first create a `CvFont` object,[21] which you then use whenever you want to put some text on the screen. Fonts are created using the `cv::fontQt()` function:

```
CvFont fontQt(                          // Return OpenCV font characerization struct
  const string& fontName,               // e.g., "Times"
  int           pointSize,              // Size of font, using "point" system.
  cv::Scalar    color    = cv::Scalar::all(0), // BGR color as a scalar (no alpha)
  int           weight   = cv::FONT_NORMAL,    // Font weights, 1-100 or see Table 4-7
  int           spacing  = 0            // Space between individual characters
);
```

The first argument to `cv::fontQt()` is the system font name. This might be something like "Times." If your system does not have an available font with this name, then a default font will be chosen for you. The second argument `pointSize` is the size of the font (i.e., `12`="12 point", `14`="14 point", etc.) You may set this to `0`, in which case a default font size (typically 12 point) will be selected for you.

The argument color can be set to any `cv::Scalar` and will set the color in which the font is drawn; the default value is black. `weight` can take one of several pre-defined values, or any integer you like between `1` and `100`. The pre-defined aliases and their values are shown in Table 4-7.

Table 4-7: Pre-defined aliases for Qt-font weight and their associated values

Camera capture constan	Numerical value
`cv::FONT_LIGHT`	25
`cv::FONT_NORMAL`	50
`cv::FONT_DEMIBOLD`	63
`cv::FONT_BOLD`	75
`cv::FONT_BLACK`	87

The final argument is spacing, which controls the spacing between individual characters. It can be negative or positive.

[20] Unfortunately, there is no "carriage return" for button placement.

[21] You will notice that the name of this object is `CvFont` rather than what you might expect: `cv::Font`. This is a legacy to the old pre-C++ interface. `CvFont` is a `struct`, and is not in the `cv::` namespace.

Once you have your font, you can put text on an image (and thus on the screen)[22] with `cv::addText()`.

```
void cv::addText(
  cv::Mat&      image,                  // Image onto which to write
  const string& text,                   // Text to write
  cv::Point     location,               // Location of lower-left corner of text
  CvFont*       font                    // OpenCV font characerization struct
);
```

The arguments to `cv::addText()` are just what you would expect, the `image` to write on, the `text` to write, where to write it, and the `font` to use—with the latter being a font you defined using `cv::fontQt`. The `location` argument corresponds to the lower-left corner of the first character in text (or, more precisely, the beginning of the *baseline* for that character).

Setting and Getting Window Properties

Many of the state properties of a window set at creation are queryable, and many of those can be changed (set) even after the window's creation.

```
void   cv::setWindowProperty(
  const string& name,                   // Handle used to identify window
  int           prop_id,                // Identifies window property (see Table 4-8)
  double        prop_value              // Value to which to set property
);

double cv::getWindowProperty(
  const string& name,                   // Handle used to identify window
  int           prop_id                 // Identifies window property (see Table 4-8)
);
```

To get a window property, you need only call `cv::getWindowProperty()` and supply the `name` of the window and the property ID (`prop_id` argument) of the property you are interested in (see Table 4-8).

Table 4-8: Gettable and settable window properties

Property name	Description
`cv::WIND_PROP_FULL_SIZE`	Set to either cv::WINDOW_FULLSCREEN for full screen window, or to cv::WINDOW_NORMAL for regular window.
`cv::WIND_PROP_AUTOSIZE`	Set to either cv::WINDOW_AUTOSIZE to have the window automatically size to the displayed image, or cv::WINDOW_NORMAL to have the image size to the window.
`cv::WIND_PROP_ASPECTRATIO`	Set to either cv::WINDOW_FREERATIO to allow the window to have any aspect ratio (as a result of user resizing) or cv::WINDOW_KEEPRATIO to only allow user resizing to affect absolute size (and not aspect ratio).

[22] It is important to notice here that `cv::addText()` is somewhat unlike all of the rest of the functions in the Qt interface (though not inconsistent with the behavior of its non-Qt analog `cv::putText()`). Specifically, `cv::addText()` does not put text in or on a *window*, but rather in an *image*. This means that you are actually changing the pixel values of the image—which is different than what would happen if, for example, you were to use `cv::displayOverlay()`.

Saving and Recovering Window State

The Qt interface also allows the state of windows to be saved and restored. This can be very convenient, as it includes not only the location and size of your windows, but also the state of all of the trackbars and buttons. The interface state is saved with the `cv::saveWindowParameters()` function, which takes a single argument indicating the window to be saved:

```
void cv::saveWindowParameters(
  const string& name                    // Handle used to identify window
);
```

Once the state of the window is saved, it can be restored with the complementary `cv::loadWindowParameters()` function:

```
void cv::loadWindowParameters(
  const string& name                    // Handle used to identify window
);
```

The real magic here is that the load command will work correctly even if you have quit and restarted your program. The nature of how this is done is not important to us here, but one detail you should know is that the state information, wherever it is saved, is saved under a key that is constructed from the executable name. So if you should change the name of the executable (though you can change its location), the state will not restore.

Interacting with OpenGL

On many contexts, it is very useful to be able to use OpenGL to render synthetic images and display them on top of camera of other processed images. The HighGUI and Qt GUI's provide a convenient way to use OpenGL to perform that rendering right on top of the whatever image you are already showing.[23] This can be extremely effective for visualizing and debugging robotic or augmented-reality applications, or anywhere in which you are trying to generate a three-dimensional model from your image and want to see the resulting model visualized on top of the original incoming image.

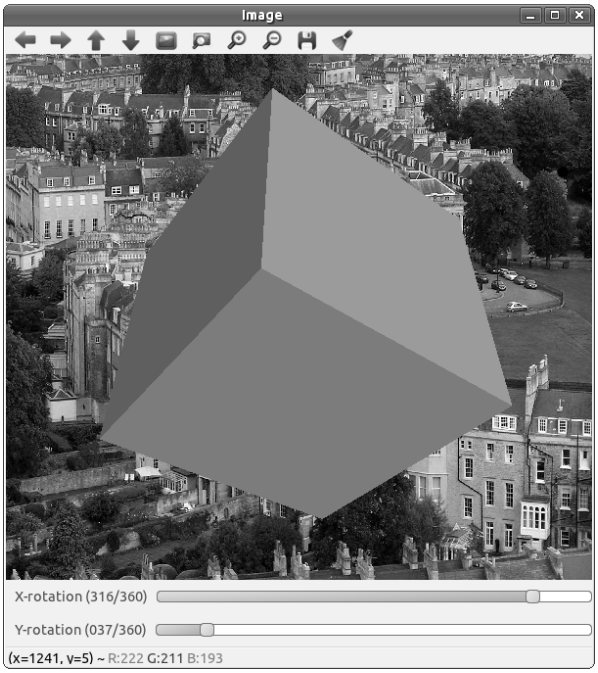


Figure 4-6. Here OpenGL is used to render a cube on top of our previous image

The basic concept is very simple; you create a callback that is an OpenGL draw function, and register it with the interface. From there, OpenCV takes care of the rest. The callback is then called every time the window is drawn (which includes whenever you call `cv::imshow()`, as you would with successive frames of a video stream). Your callback should match the prototype for `cv::OpenGLCallback()`, which means that it should be something like the following:

```
void your_opengl_callback(
  void* params             // (Optional) Parameters from cv::createOpenGLCallback()
);
```

Once you have your callback, you can configure the OpenGL interface with `cv::setOpenGlDrawCallback()`:

```
void cv::setOpenGlDrawCallback(
  const string&      windowName,          // Handle used to identify window
  cv::OpenGLCallback callback,            // OpenGL callback routine
  void*              params    = NULL     // (Optional) parameters for callback
);
```

As you can see, there is not much one needs to really set things up. In addition to specifying the name of the window on which the drawing will be done and supplying the `callback` function, you have a third

[23] By way of reminder, in order to use these commands, you will need to have built OpenCV with the cmake flag –D WITH_OPENGL=ON.

argument `params`, which allows you to specify a pointer that will be passed to `callback` whenever it is called.

> It is probably worth calling out here explicitly that none of this sets up the camera, lighting, or other aspects of your OpenGL activities. Internally there is a wrapper around your OpenGL callback that will set up the projection matrix using a call to `gluPerspective()`. If you want anything different (which you almost certainly will), you will have to clear and configure the projection matrix at the beginning of your callback.

In Figure 4-6, we have taken a simple example code from the OpenCV documentation which draws a cube in OpenGL, but we have replaced the fixed rotation angles in that cube with variables (`rotx` and `roty`) which we have made the values of the two sliders in our earlier examples. Now the user can rotate the cube with the sliders while enjoying the beautiful scenery behind.

Example 4-5: Slightly modified code from the OpenCV documentation that draws a cube every frame; this modified version uses the global variables `rotx` and `roty` that are connected to the sliders in Figure 4-6

```
void on_opengl( void* param )
{
  glMatrixModel( GL_MODELVIEW );
  glLoadIdentity();

  glTranslated( 0.0, 0.0, -1.0 );

  glRotatef( rotx, 1, 0, 0 );
  glRotatef( roty, 0, 1, 0 );
  glRotatef( 0, 0, 0, 1 );

  static const int coords[6][4][3] = {
    { { +1, -1, -1 }, { -1, -1, -1 }, { -1, +1, -1 }, { +1, +1, -1 } },
    { { +1, +1, -1 }, { -1, +1, -1 }, { -1, +1, +1 }, { +1, +1, +1 } },
    { { +1, -1, +1 }, { +1, -1, -1 }, { +1, +1, -1 }, { +1, +1, +1 } },
    { { -1, -1, -1 }, { -1, -1, +1 }, { -1, +1, +1 }, { -1, +1, -1 } },
    { { +1, -1, +1 }, { -1, -1, +1 }, { -1, -1, -1 }, { +1, -1, -1 } },
    { { -1, -1, +1 }, { +1, -1, +1 }, { +1, +1, +1 }, { -1, +1, +1 } }
  };

  for (int i = 0; i < 6; ++i) {
    glColor3ub( i*20, 100+i*10, i*42 );
    glBegin(GL_QUADS);
    for (int j = 0; j < 4; ++j) {
      glVertex3d(0.2 * coords[i][j][0], 0.2 * coords[i][j][1], 0.2 * coords[i][j][2]);
    }
    glEnd();
  }
}
```

As was mentioned above, whenever you call `cv::imshow()`, a call will be generated to your OpenGL callback. You can also induce a call directly to that callback by calling `cv::updateWindow()`.[24]

Integrating OpenCV with Full GUI Toolkits

Even OpenCV's built-in Qt interface is still just a handy way of accomplishing some simple tasks that come up often while developing code or exploring algorithms. When it comes time to actually build an end

[24] You will really use `cv::updateWindow()` only when you are using OpenGL. The utility of `cv::updateWindow()` is precisely that it induces an OpenGL draw event (and thus calls your callback which you would have set up with `cv::setOpenGLDrawCallback()`).

user facing application, neither the native UI nor the Qt-based interface are going to do it. In this section, we will (very) briefly explore some of the issues and techniques for working with OpenCV and two existing toolkits: wxWidgets and Qt.

Clearly there are countless UI toolkits out there, and we would not want to waste time digging into each of them. Having said that, it is useful to explore how to handle the issues which will arise if you want to use OpenCV with a more fully featured toolkit.

The primary issue is how to convert OpenCV images to the form that the toolkit expects for graphics, and to know which widget or component in the toolkit is going to do that display work for you. From there, you don't need much else that is specific to OpenCV. Notably, you will not need or want the features of the UI toolkits we have covered in this chapter.

An Example of OpenCV and Qt

Here we will show an example of using the actual Qt toolkit to write a program that reads a video file and displays it on the screen. There are several subtleties, some of which have to do with how one uses Qt, and others to do with OpenCV. Of course, we will focus on the latter, but it is worth taking a moment to notice how the former affect our current goal.

Below is the top-level code for our example; it just creates a Qt application and adds our `QMoviePlayer` widget. Everything interesting will happen inside that object.

Example 4-6. An example program ch4_qt.cpp, which takes a single argument indicating a video file; that video file will be replayed inside of a Qt object which we will define, called `QMoviePlayer`

```
#include <QApplication>
#include <QLabel>
#include <QMoviePlayer.hpp>
int main( int argc, char* argv[] ) {

  QApplication app( argc, argv );

  QMoviePlayer mp;
  mp.open( argv[1] );
  mp.show();

  return app.exec();
}
```

The interesting stuff is in the `QMoviePlayer` object. Let's take a look at the header file which defines that object:

4-6 cont. The QMoviePlayer object header file QMoviePlayer.hpp

```
#include "ui_QMoviePlayer.h"
#include <opencv2/opencv.hpp>
#include <string>

using namespace std;

class QMoviePlayer : public QWidget {

  Q_OBJECT;

  public:
  QMoviePlayer( QWidget *parent = NULL );
  virtual ~QMoviePlayer() {;}

  bool open( string file );

  private:
  Ui::QMoviePlayer  ui;
  cv::VideoCapture m_cap;
```

```
  QImage  m_qt_img;
  cv::Mat m_cv_img;
  QTimer* m_timer;

  void paintEvent( QPaintEvent* q );
  void _copyImage( void );

  public slots:
  void nextFrame();

};
```

There is a lot going on here. The first thing that happens is the inclusion of the file *ui_QMoviePlayer.h*. This file was automatically generated by the Qt Designer. What matters here is that it is just a `QWidget` that contains nothing but a `QFrame` called `frame`. The member `ui::QMoviePlayer` is that interface object which is defined in *ui_QMoviePlayer.h*.

In this file, there is also a `QImage` called `m_qt_img` and a `cv::Mat` called `m_cv_img`. These will contain the Qt and OpenCV representations of the image we are getting from our video. Finally, there is a `QTimer`, which is what will take the place of `cv::waitKey()`, allowing us to replay the video frames at the correct rate. The remaining functions will become clear as we look at their actual definitions in `QMoviePlayer.cpp`.

4-6 cont. The QMoviePlayer object source file: QMoviePlayer.cpp

```
#include "QMoviePlayer.hpp"
#include <QTimer>
#include <QPainter>

QMoviePlayer::QMoviePlayer( QWidget *parent )
 : QWidget( parent )
{
  ui.setupUi( this );
}
```

The top-level constructor for `QMoviePlayer` just calls the setup function, which was automatically built for us for the UI member.

4-6 cont. Open.

```
bool QMoviePlayer::open( string file ) {

  if( !m_cap.open( file ) ) return false;

  // If we opened the file, set up everything now:
  //
  m_cap.read( m_cv_img );
  m_qt_img = QImage(
    QSize( m_cv_img.cols, m_cv_img.rows ),
    QImage::Format_RGB888
  );
  ui.frame->setMinimumSize( m_qt_img.width(), m_qt_img.height() );
  ui.frame->setMaximumSize( m_qt_img.width(), m_qt_img.height() );
  _copyImage();

  m_timer = new QTimer( this );
  connect(
    m_timer,
    SIGNAL( timeout() ),
    this,
    SLOT( nextFrame() )
  );
  m_timer->start( 1000. / m_cap.get( cv::CAP_PROP_FPS ) );
```

```
  return true;

}
```

When an `open()` call is made on the `QMoviePlayer`, several things have to happen. The first is that the `cv::VideoCapture` member object `m_cap` needs to be opened. If that fails, we just return. Next we read the first frame into our OpenCV image member `m_cv_img`. Once we have this, we can set up the Qt image object `m_qt_img`, giving it the same size as the OpenCV image. Next we resize the frame object in the UI element to be the same size as the incoming images as well.

We will look at the call to `QMoviePlayer::_copyImage()` in a moment; this is going to handle the very important process of converting the image we have already captured into `m_cv_img` onto the Qt image `m_qt_img`, which we are actually going to have Qt paint onto the screen for us.

The last thing we do in `QMoviePlayer::open()` is to set up a `QTimer` which, when it "goes off," will call the function `QMoviePlayer::nextFrame()` (which will, not surprisingly, get the next frame). The call to `m_timer->start()` is how we both start the timer running, and indicate that it should go off at the correct rate implied by `cv::CAP_PROP_FPS` (i.e., 1,000 milliseconds divided by the frame rate).

4-6 cont. cv::Mat to QImage.

```
void QMoviePlayer::_copyImage( void ) {

  // Copy the image data into the Qt QImage
  //
  cv::Mat cv_header_to_qt_image(
    cv::Size(
      m_qt_img.width(),
      m_qt_img.height()
    ),
    cv::U8C3,
    m_qt_img.bits()
  );
  cv::cvtColor( m_cv_img, cv_header_to_qt_image, cv::BGR2RGB );
}
```

The function `QMoviePlayer::_copyImage()` is responsible for copying the image from the buffer `m_cv_img` into the Qt image buffer `m_qt_img`. The way we do this shows off a nice feature of the `cv::Mat` object. We first define a `cv::Mat` object called `cv_hreader_to_qt_image`. When we define that object, we actually tell it what area to use for its data area, and hand it the data area for the Qt `QImage` object `m_qt_img.bits()`. We then call `cv::cvtColor` to do the copying, which handles the subtlety that OpenCV prefers BGR ordering, while Qt prefers RGB.

4-6 cont. nextFrame.

```
void QMoviePlayer::nextFrame() {

  // Nothing to do if capture object is not open
  //
  if( !m_cap.isOpened() ) return;

  m_cap.read( m_cv_img );
  _copyImage();

  this->update();
}
```

The `QMoviePlayer::nextFrame()` function actually handles the reading of subsequent frames. Recall that this routine is called whenever the `QTimer` expires. It reads the new image into the OpenCV buffer, calls `QMoviePlayer::_copyImage()` to copy it into the Qt buffer, and then makes an update call on the `QWidget` that this is all part of (so that Qt knows that something has changed).

4-6 cont. paintEvent.

```
void QMoviePlayer::paintEvent( QPaintEvent* e ) {

  QPainter painter( this );
  painter.drawImage( QPoint( ui.frame->x(), ui.frame->y()), m_qt_img );

}
```

Last, but not least, is the function `QMoviePlayer::paintEvent()`. This is the function that is called by Qt whenever it is necessary to actually draw the `QMoviePlayer` widget. This function just creates a `QPainter` and tells it to draw the current Qt image `m_qt_img` (starting at the corner of the screen).

An Example of OpenCV and wxWidgets

In this example, we will use a different cross-platform toolkit, wxWidgets. The wxWidgets toolkit is similar in many ways to Qt in terms of its GUI components, but of course, it is in the details that things tend to become difficult. As with the Qt example, we will have one top-level file that basically puts everything in place and a code and header file pair that define an object which encapsulates our example task of playing a video. This time our object will be called `WxMoviePlayer`, and we will build it based on the UI classes provided by wxWidgets.

Example 4-7: An example program ch4_wx.cpp, which takes a single argument indicating a video file; that video file will be replayed inside of a wxWidgets object which we will define called `WxMoviePlayer`

```
#include "wx/wx.h"
#include "WxMoviePlayer.hpp"

// Application class, the top level object in wxWidgets
//
class MyApp : public wxApp {
  public:
    virtual bool OnInit();
};

// Behind the scenes stuff to create a main() function and attach MyApp
//
DECLARE_APP( MyApp );
IMPLEMENT_APP( MyApp );

// When MyApp is initialized, do these things.
//
bool MyApp::OnInit() {

  wxFrame* frame = new wxFrame( NULL, wxID_ANY, wxT("ch4_wx") );
  frame->Show( true );
  WxMoviePlayer* mp = new WxMoviePlayer(
    frame,
    wxPoint( -1, -1 ),
    wxSize( 640, 480 )
  );
  mp->open( wxString(argv[1]) );
  mp->Show( true );

  return true;

}
```

The structure here is a little more complicated in appearance than with the Qt example, but the content is very similar. The first thing we do is create a class definition for our application, which we derive from the library class wxApp. The only thing different about our class is that it will overload the `MyApp::OnInit()` function with our own content. After declaring `class MyApp`, we call two macros `DECLARE_APP()` and `IMPLEMENT_APP()`. In short, these are creating the `main()` function,

and installing an instance of MyApp as "the application." The last thing we do in our main program is to actually fill out the function `MyApp::OnInit()`. This is going to get called when our program starts: this functions largely as the equivalant to `main()` from our point of view. This function creates the window (called a "frame" in wxWidgets), and an instance of our WxMoviePlayer object in that frame. It then calls the open method on the WxMoviePlayer and hands it the name of the movie file we want to open.

Of course, all of the interesting stuff is happening inside of the WxMoviePlayer object. Here is the header file for that object:

The WxMoviePlayer object header file WxMoviePlayer.hpp

```
#include "opencv2/opencv.hpp"

#include "wx/wx.h"
#include <string>

#define TIMER_ID 0

using namespace std;

class WxMoviePlayer : public wxWindow {

  public:
    WxMoviePlayer(
      wxWindow*      parent,
      const wxPoint& pos,
      const wxSize&  size
    );
    virtual ~WxMoviePlayer() {};
    bool open( wxString file );

  private:

    cv::VideoCapture m_cap;
    cv::Mat          m_cv_img;
    wxImage          m_wx_img;
    wxBitmap         m_wx_bmp;
    wxTimer*         m_timer;
    wxWindow*        m_parent;

    void _copyImage( void );

    void OnPaint( wxPaintEvent& e );
    void OnTimer( wxTimerEvent& e );
    void OnKey(   wxKeyEvent&   e );

  protected:
    DECLARE_EVENT_TABLE();
};
```

The important things to notice in the above declaration are the following. The `WxMoviePlayer` object is derived from `wxWindow`, which is the generic class used by wxWidgets for just about anything that will be visible on the screen. We have three event-handling methods `OnPaint()`, `onTimer()`, and `OnKey()`. These will handle drawing, getting a new image from the video, and closing the file with the ESC key, respectively. Finally, you will notice that there is an object of type `wxImage` and an object of type `wxBitmap`, in addition to the OpenCV `cv:Mat` type image. In wxWidgets, bitmaps (which are operating system dependant) are distinguished from "images" (which are device independent representations of image data). The exact role of these two will be clear shortly as we look at the code file WxMoviePlayer.cpp.

The WxMoviePlayer object source file WxMoviePlayer.cpp

```
#include "WxMoviePlayer.hpp"

BEGIN_EVENT_TABLE( WxMoviePlayer, wxWindow )
  EVT_PAINT( WxMoviePlayer::OnPaint )
  EVT_TIMER( TIMER_ID, WxMoviePlayer::OnTimer )
  EVT_CHAR( WxMoviePlayer::OnKey )
END_EVENT_TABLE()
```

The first thing we do is to set up the callbacks that will be associated with individual events. This is done by macros provided by the wxWidgets framework.[25]

```
WxMoviePlayer::WxMoviePlayer(
  wxWindow*      parent,
  const wxPoint& pos,
  const wxSize&  size
) : wxWindow( parent, -1, pos, size, wxSIMPLE_BORDER ) {
  m_timer         = NULL;
  m_parent        = parent;
}
```

When the movie player is created, its timer element is NULL (we will set that up when we actually have a video open). We do take note of the parent of the player, however. (In this case, that parent will be the wxFrame we created to put it in.) We will need to know who the parent frame is when it comes time to closing the application in response to the ESC key.

```
void WxMoviePlayer::OnPaint( wxPaintEvent& event ) {
  wxPaintDC dc( this );

  if( !dc.Ok() ) return;

  int x,y,w,h;
  dc.BeginDrawing();
    dc.GetClippingBox( &x, &y, &w, &h );
    dc.DrawBitmap( m_wx_bmp, x, y );
  dc.EndDrawing();

  return;
}
```

The `WxMoviePlayer::OnPaint()` routine is called whenever the window needs to be repainted on screen. You will notice that when we execute `WxMoviePlayer::OnPaint()`, the information we need to actually do the painting is assumed to be in `m_wx_bpm`, the `wxBitmap` object. Because the wxBitmap is the system dependant representation, it is already prepared to be copied to the screen. The next two methods, `WxMoviePlayer::_copyImage()` and `WxMoviePlayer::open()` will show how it got created in the first place.

```
void WxMoviePlayer::_copyImage( void ) {
  m_wx_bmp = wxBitmap( m_wx_img );

  Refresh( FALSE ); // indicate that the object is dirty
  Update();
}
```

[25] The astute reader will notice that the keyboard event is "hooked up" to the WxMoviePlayer widget and not to the top-level application or the frame (as was the case for the Qt example, and is the case for HighGUI). There are various ways to accomplish this, but wxWidgets really prefers your keyboard events to be bound locally to visible objects in your UI, rather than globally. Since this is a simple example, we chose to just do the easiest thing and bind the keyboard events directly to the movie player.

The `WxMoviePlayer::_copyImage()` method is what is going to get called whenever a new image is read from the `cv::VideoCapture` object. It does not appear to do much, but actually a lot is going on in its short body. First and foremost is the construction of the `wxBitmap m_wx_bmp` from the `wxImage m_wx_img`. The constructor is handling the conversion from the abstract representation used by wxImage (which, we will see, looks very much like the representation used by OpenCV) to the device and system specific representation used by your particular machine. Once that copy is done, a call to `Refresh()` indicates that the widget is "dirty" and needs redrawing, and the subsequent call to `Update()` actually indicates that the time for that redrawing is now.

```
bool WxMoviePlayer::open( wxString file ) {
  if( !m_cap.open( std::string( file.mb_str() ) )) {
    return false;
  }

  // If we opened the file, set up everything now:
  //
  m_cap.read( m_cv_img );

  m_wx_img = wxImage(
    m_cv_img.cols,
    m_cv_img.rows,
    m_cv_img.data,
    TRUE  // static data, do not free on delete()
  );

  _copyImage();

  m_timer = new wxTimer( this, TIMER_ID );
  m_timer->Start( 1000. / m_cap.get( cv::CAP_PROP_FPS ) );

  return true;
}
```

The `WxMoviePlayer::open()` method also does several important things. The first is to actually open the `cv::VideoCapture` object, but there is a lot more to be done. Next, an image is read off of the player, and it is used to create a `wxImage` object which "points at" the OpenCV `cv::Mat` image. This is the opposite philosophy to the one we used in the Qt example: in this case, it turns out to be a little more convenient to create the `cv::Mat` first and have it own the data, and the GUI toolkit's image object second and have it be just a header to that existing data. Next, we call `WxMoviePlayer::_copyImage()`, and that function converts the OpenCV image `m_cv_img` into the native bitmap for us.

Finally, we create a `wxTimer` object and tell it to wake us up every few milliseconds—with that number being computed from the FPS reported by the `cv::VideoCapture` object. Whenever that timer expires, a `wxTimerEvent` is generated and passed to `WxMoviePlayer::OnTimer()`, which you will recall to be the handler of such events.

```
void WxMoviePlayer::OnTimer( wxTimerEvent& event ) {
  if( !m_cap.isOpened() ) return;
  m_cap.read( m_cv_img );
  cv::cvtColor( m_cv_img, m_cv_img, cv::BGR2RGB );
  _copyImage();
}
```

That handler does not do too much; primarily it just iterates the reading of a new frame from the video and the conversion of that frame from BGR to RGB for display, and then calls our `WxMoviePlayer::_copyImage()`, which makes the next bitmap for us.

```
void WxMoviePlayer::OnKey( wxKeyEvent& e ) {
  if( e.GetKeyCode() == WXK_ESCAPE ) m_parent->Close();
}
```

Finally, we have our handler for any key-presses. It simply checks to see if that key was the Escape key, and if so, closes the program. It is worth noting that we do not close the WxMoviePlayer object, but rather the parent frame. Closing the frame is the same as closing the window any other way; it shuts down the application.

An Example of OpenCV and the Windows Template Library

In this example, we will use the native windows GUI API.[26] The Windows Template Library, WTL, is a very thin C++ wrapper around the raw Win32 API. WTL applications are structured similarly to MFC, in that there is an application/document-view structure. For the purposes of this sample, we will start by running the WTL Application Wizard from within Visual Studio, and creating a new "SDI Application" and under "User Interface Features" ensuring "Use a View Window" is selected (it should be, by default).

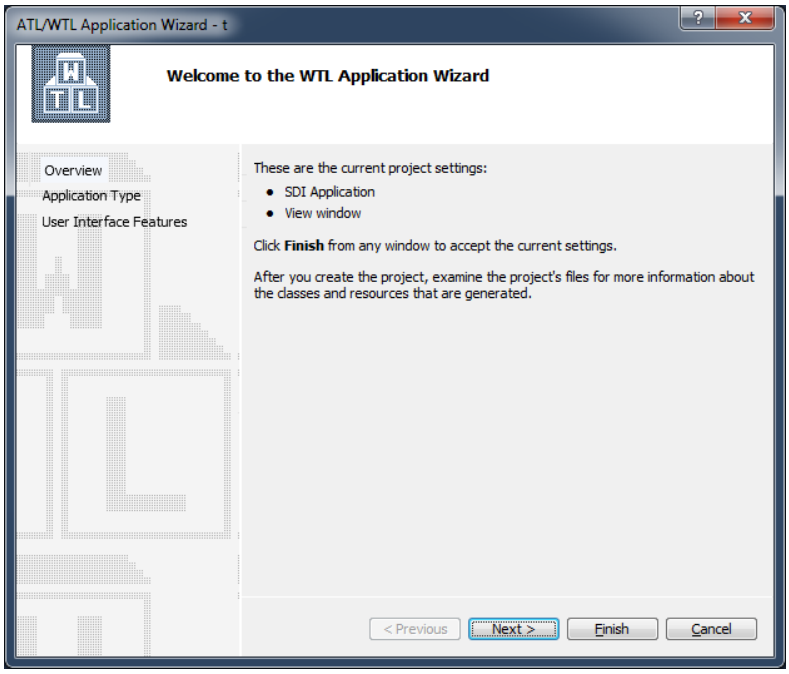


Figure 4-7: The WTL Application Wizard

The exact file names generated will depend on the name you give your project. For this example, the project is named *OpenCVTest*, and we will mostly be working in the `COpenCVTestView` class.

Example 4-8: An example header file for our custom View class

```
class COpenCVTestView : public CWindowImpl<COpenCVTestView> {

public:
  DECLARE_WND_CLASS(NULL)

  bool OpenFile(std::string file);
  void _copyImage();


  BOOL PreTranslateMessage(MSG* pMsg);

  BEGIN_MSG_MAP(COpenCVTestView)
    MESSAGE_HANDLER(WM_ERASEBKGND, OnEraseBkgnd)
    MESSAGE_HANDLER(WM_PAINT, OnPaint)
```

[26] Special thanks to Sam Leventer, who is the original author of this WTL example code.

```
    MESSAGE_HANDLER(WM_TIMER, OnTimer)
  END_MSG_MAP()

// Handler prototypes (uncomment arguments if needed):
//  LRESULT MessageHandler(
//    UINT    /*uMsg*/,
//    WPARAM  /*wParam*/,
//    LPARAM  /*lParam*/,
//    BOOL&   /*bHandled*/
//  );
//  LRESULT CommandHandler(
//    WORD    /*wNotifyCode*/,
//    WORD    /*wID*/,
//    HWND    /*hWndCtl*/,
//    BOOL&   /*bHandled*/
//  );
//  LRESULT NotifyHandler(
//    int     /*idCtrl*/,
//    LPNMHDR /*pnmh*/,
//    BOOL&   /*bHandled*/
//  );

  LRESULT OnPaint(
    UINT    /*uMsg*/,
    WPARAM  /*wParam*/,
    LPARAM  /*lParam*/,
    BOOL&   /*bHandled*/
  );
  LRESULT OnTimer(
    UINT    /*uMsg*/,
    WPARAM  /*wParam*/,
    LPARAM  /*lParam*/,
    BOOL&   /*bHandled*/
  );
  LRESULT OnEraseBkgnd(
    UINT    /*uMsg*/,
    WPARAM  /*wParam*/,
    LPARAM  /*lParam*/,
    BOOL&   /*bHandled*/
  );

private:
  cv::VideoCapture m_cap;
  cv::Mat          m_cv_img;

  RGBTRIPLE*       m_bitmapBits;
};
```

The structure here is very similar to the preceding wx example. The only change outside of the view code is for the `Open` menu item handler, which will be in your `CMainFrame` class. It will need to call into the view class to open the video:

```
LRESULT CMainFrame::OnFileOpen(
  WORD /*wNotifyCode*/,
  WORD /*wID*/,
  HWND /*hWndCtl*/,
  BOOL& /*bHandled*/
) {
    WTL::CFileDialog dlg(TRUE);
  if (IDOK == dlg.DoModal(m_hWnd)) {
    m_view.OpenFile(dlg.m_szFileName);
  }
```

```
  return 0;
}

bool COpenCVTestView::OpenFile(std::string file) {

  if( !m_cap.open( file ) ) return false;

  // If we opened the file, set up everything now:
  //
  m_cap.read( m_cv_img );

  // could create a DIBSection here, but lets just allocate memory for the raw bits
  m_bitmapBits = new RGBTRIPLE[m_cv_img.cols * m_cv_img.rows];

  _copyImage();

  SetTimer(0, 1000.0f / m_cap.get( cv::CAP_PROP_FPS ) );

  return true;
}


void COpenCVTestView::_copyImage() {

  // Copy the image data into the bitmap
  //
  cv::Mat cv_header_to_qt_image(
    cv::Size(
      m_cv_img.cols,
      m_cv_img.rows
    ),
    CV::U8C3,
    m_bitmapBits
  );
  cv::cvtColor( m_cv_img, cv_header_to_qt_image, cv::BGR2RGB );
}

LRESULT COpenCVTestView::OnPaint(
  UINT   /*uMsg*/,
  WPARAM /*wParam*/,
  LPARAM /*lParam*/,
  BOOL&  /*bHandled*/
) {
  CPaintDC dc(m_hWnd);

  WTL::CRect rect;
  GetClientRect(&rect);

  if( m_cap.isOpened() ) {
    BITMAPINFO bmi = {0};
    bmi.bmiHeader.biSize = sizeof(bmi.bmiHeader);
    bmi.bmiHeader.biCompression = BI_RGB;
    bmi.bmiHeader.biWidth = m_cv_img.cols;
    // note that bitmaps default to bottom-up, use negative height to
    // represent top-down
    bmi.bmiHeader.biHeight = m_cv_img.rows * -1;

    bmi.bmiHeader.biPlanes = 1;
    bmi.bmiHeader.biBitCount = 24;  // 32 if you use RGBQUADs for the bits

    dc.StretchDIBits(
      0,                    0,
      rect.Width(),         rect.Height(),
```

```
      0,                     0,
      bmi.bmiHeader.biWidth, abs(bmi.bmiHeader.biHeight),
      m_bitmapBits,
      &bmi,
      DIB_RGB_COLORS,
      SRCCOPY
    );
  } else {
    dc.FillRect(rect, COLOR_WINDOW);
  }

  return 0;
}

LRESULT COpenCVTestView::OnTimer(
  UINT   /*uMsg*/,
  WPARAM /*wParam*/,
  LPARAM /*lParam*/,
  BOOL&  /*bHandled*/
) {
  // Nothing to do if capture object is not open
  //
  if( !m_cap.isOpened() ) return 0;

  m_cap.read( m_cv_img );
  _copyImage();

  Invalidate();

  return 0;
}

LRESULT COpenCVTestView::OnEraseBkgnd(
  UINT   /*uMsg*/,
  WPARAM /*wParam*/,
  LPARAM /*lParam*/,
  BOOL&  /*bHandled*/
) {
  // since we completely paint our window in the OnPaint handler, use
  // an empty background handler
  return 0;
}
```

This code illustrates how to use bitmap-based drawing in a C++ application under Windows. This method is simpler, but less efficient than using DirectShow to handle the video stream.

> If you are using the .NET Runtime (either through C#, VB.NET or Managed C++), then you may want to look into a package that completely wraps OpenCV, such as Emgu (*http://emgu.com*).

Drawing Things

We often want to draw some kind of picture, or to draw something on top of an image obtained from somewhere else. Toward this end, OpenCV provides a menagerie of functions that will allow us to make lines, squares, circles, and the like.

The drawing functions available will work with images of any depth, but most of them only affect the first three channels—defaulting to only the first channel in the case of single-channel images. Most of the drawing functions support a color, a thickness, what is called a "line type" (which really means whether or not to anti-alias lines), and subpixel alignment of objects.

When specifying colors, the convention is to use the `cv::Scalar` object, even though only the first three values are used most of the time. (It is sometimes convenient to be able to use the fourth value in a `cv::Scalar` to represent an alpha-channel, but the drawing functions do not currently support alpha blending.) Also, by convention, OpenCV uses BGR ordering[27] for converting multichannel images to color renderings (this is what is used by the draw functions `imshow()`, which actually paints images onto your screen for viewing). Of course, you don't have to use this convention, and it might not be ideal if you are using data from some other library with OpenCV headers on top of it. In any case, the core functions of the library are always agnostic to any "meaning" you might assign to a channel.

Line Art and Filled Polygons

Functions that draw lines of one kind or another (segments, circles, rectangles, etc.) will usually accept a `thickness` and `lineType` parameter. Both are integers, but the only accepted values for the latter are `4`, `8`, or `cv::AA`. `thickness` will be the thickness of the line measured in pixels. For circles, rectangles, and all of the other closed shapes, the `thickness` argument can also be set to `cv::FILL` (which is an alias for `-1`). In that case, the result is that the drawn figure will be filled in the same color as the edges. The `lineType` argument indicates whether the lines should be "4-connected," "8-connected," or anti-aliased. For the first two, the Bresenham algorithm is used, while for the anti-aliased lines, Gaussian filtering is used. Wide lines are always drawn with rounded ends.

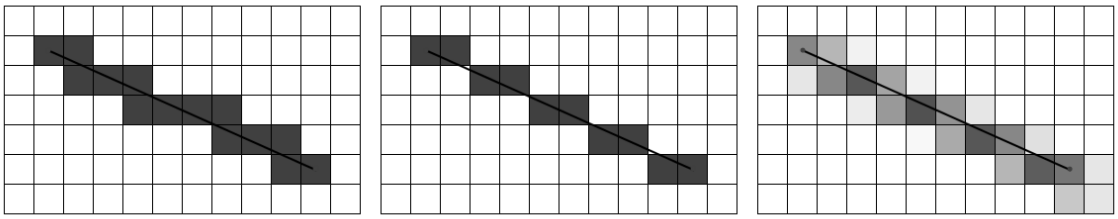


Figure 4-8: The same line as it would be rendered using the 4-connected (a), 8-connected (b), and anti-aliased (c) line types

For the drawing algorithms, endpoints (lines), center-points (circles), corners (rectangles), and so on are typically specified as integers. However, these algorithms support subpixel alignment, through the `shift` argument. Where `shift` is available, it is interpreted as the number of bits in the integer arguments to treat as fractional bits. For example, if you say you want a circle centered at $(5, 5)$, but set `shift` to `1`, then the circle will be drawn at $(2.5, 2.5)$. The effect of this will typically be quite subtle, and depend on the line type used. The effect is most noticeable for anti-aliased lines.

Table 4-9:Drawing Functions

Function	Description
`cv::circle()`	Draw a simple circle

[27] There is a slightly confusing point here, which is mostly due to legacy in origin. That is that the macro `CV_RGB(r,g,b)` produces a `cv::Scalar s` with value `s.val[] = { b, g, r, 0 }`. This is as it should be, as general OpenCV functions only know what is red, green, or blue by the order, and the ordering convention for image data is BGR as stated in the text.

`cv::clipLine()`	Determine if a line is inside a given box
`cv::ellipse()`	Draw an ellipse, which may be tilted or an elliptical arc
`cv::ellipse2Poly()`	Compute a polygon approximation to an elliptical arc
`cv::fillConvexPoly()`	Fast algorithm for drawing filled versions of simple polygons
`cv::fillPoly()`	General algorithm for drawing filled versions of arbitrary polygons
`cv::line()`	Draw a simple line
`cv::rectangle()`	Draw a simple rectangle
`cv::polyLines()`	Draw multiple polygonal curves

cv::circle()

```
void circle(
  cv::Mat&          img,                // Image to be drawn on
  cv::Point         center,             // Location of circle center
  int               radius,             // Radius of circle
  const cv::Scalar& color,              // Color, RGB form
  int               thickness = 1,      // Thickness of line
  int               lineType  = 8,      // Connectedness, 4 or 8
  int               shift     = 0       // Bits of radius to treat as fraction
);
```

The first argument to `cv::circle()` is just your image `img`. Next, are the `center`, a two-dimensional point, and the `radius`. The remaining arguments are the standard `color`, `thickness`, `lineType`, and `shift`. The shift is applied to both the radius and the center location.

cv::clipLine()

```
bool clipLine(                          // True if any part of line in 'imgRect'
  cv::Rect          imgRect,            // Rectangle to clip to
  cv::Point&        pt1,                // First end-point of line, overwritten
  cv::Point&        pt2                 // Second end-point of line, overwritten
);

bool clipLine(                          // True if any part of line in image size
  cv::Size          imgSize,            // Size of image, implies rectangle at 0,0
  cv::Point&        pt1,                // First end-point of line, overwritten
  cv::Point&        pt2                 // Second end-point of line, overwritten
);
```

This function is used to determine if a line, specified by the two points `pt1` and `pt2` lies inside of a rectangular boundary. In the first version, a `cv::Rect` is supplied and the line is compared to that rectangle. `cv::clipLine()` will return `False` only if the line is entirely outside of the specified rectangular region. The second version is the same, except that it takes a `cv::Size` argument. Calling this second version is equivalent to calling the first version with a rectangle whose (x, y) location is $(0, 0)$.

cv::ellipse()

```
bool ellipse(
  cv::Mat&              img,            // Image to be drawn on
  cv::Point             center,         // Location of ellipse center
  cv::Size              axes,           // Length of major and minor axes
  double                angle,          // Tilt angle of major axis
  double                startAngle,     // Start angle for arc drawing
  double                endAngle,       // End angle for arc drawing
```

```
  const cv::Scalar&      color,           // Color, BGR form
  int                    thickness = 1,   // Thickness of line
  int                    lineType  = 8,   // Connectedness, 4 or 8
  int                    shift     = 0    // Bits of radius to treat as fraction
);

bool ellipse(
  cv::Mat&               img,             // Image to be drawn on
  const cv::RotatedRect& rect,            // Rotated rectangle bounds ellipse
  const cv::Scalar&      color,           // Color, BGR form
  int                    thickness = 1,   // Thickness of line
  int                    lineType  = 8,   // Connectedness, 4 or 8
  int                    shift     = 0    // Bits of radius to treat as fraction
);
```

The `cv::ellipse()` function is very similar to the `cv::circle()` function, with the primary difference being the `axes` argument, which is of type `cv::Size`. In this case, the `height` and `width` arguments represent the length of the ellipse's major and minor axes. The `angle` is the angle (in degrees) of the major axis, which is measured counterclockwise from horizontal (i.e., from the *x*-axis). Similarly, the `startAngle` and `endAngle` indicate (also in degrees) the angle for the arc to start and for it to finish. Thus, for a complete ellipse, you must set these values to 0 and 360, respectively.

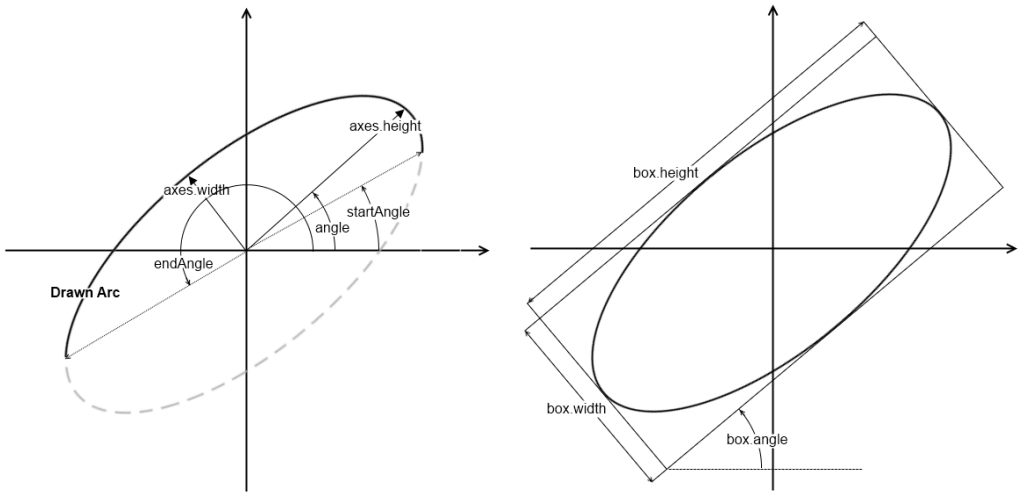


Figure 4-8: An elliptical arc specified by the major and minor axes with tilt angle (left); a similar ellipse specified using a `cv::RotatedRect` *(right).*

The alternate way to specify the drawing of an ellipse is to use a bounding box. In this case, the argument `box` of type `cv::RotatedRect` completely specifies both the size and the orientation of the ellipse. Both methods of specifying an ellipse are illustrated in Figure 4-8.

cv::ellipse2Poly()

```
void ellipse2Poly(
  cv::Point              center,          // Location of ellipse center
  cv::Size               axes,            // Length of major and minor axes
  double                 angle,           // Tilt angle of major axis
  double                 startAngle,      // Start angle for arc drawing
  double                 endAngle,        // End angle for arc drawing
  int                    delta,           // Angle between sequential vertices
  vector<cv::Point>&     pts              // Result, STL-vector of points
);
```

The `cv::ellipse2Poly()` function is used by `cv::ellipse()` internally to compute elliptical arcs, but you can call it yourself as well. Given information about an elliptical arc (`center`, `axes`, `angle`, `startAngle`, and `endAngle`—all as defined in `cv::ellipse()`) and a parameter `delta`, which specifies the angle between subsequent points you want to sample, `cv::ellipse2Poly()` computes a sequence of points that form a polygonal approximation to the elliptical arc you specified. The computed points are returned in the `vector<> pts`.

cv::fillConvexPoly()

```
void fillConvexPoly(
  cv::Mat&          img,                // Image to be drawn on
  const cv::Point*  pts,                // C-style array of points
  int               npts,               // Number of points in 'pts'
  const cv::Scalar& color,              // Color, BGR form
  int               lineType  = 8,      // Connectedness, 4 or 8
  int               shift     = 0       // Bits of radius to treat as fraction
);
```

This function draws a filled polygon. It is much faster than `cv::fillPoly()` because it uses a much simpler algorithm. This algorithm, however, will not work correctly if the polygon you pass to it has self-intersections.[28] The points in `pts` are treated as sequential, and a segment from the last point in `pts` and the first point is implied (i.e., the polygon is assumed to be closed).

cv::fillPoly()

```
void fillPoly(
  cv::Mat&          img,                // Image to be drawn on
  const cv::Point*  pts,                // C-style array of arrays of points
  int               npts,               // Number of points in 'pts[i]'
  int               ncontours,          // Number of arrays in 'pts'
  const cv::Scalar& color,              // Color, BGR form
  int               lineType = 8,       // Connectedness, 4 or 8
  int               shift    = 0,       // Bits of radius to treat as fraction
  cv::Point         offset   = Point()   // Uniform offset applied to all points
);
```

This function draws any number of filled polygons. Unlike `cv::fillConvexPoly()`, it can handle polygons with self-intersections. The argument `ncontours` specifies how many different polygon contours there will be, and the argument `npts` is a C-style array that indicates how many points there are in each contour (i.e., `npts[i]` indicates how many points there are in polygon `i`). `pts` is a C-style array of C-style arrays containing all of the points in those polygons (i.e., `pts[i][j]` contains the j^{th} point in the i^{th} polygon.) `cv::fillPoly()` also has one additional argument `offset`, which is a pixel offset that will be applied to all vertex locations when the polygons are drawn. The polygons are assumed to be closed (i.e., a segment from the last element of `pts[i][]` to the first element will be assumed).

cv::line()

```
void line(
  cv::Mat&          img,                // Image to be drawn on
  cv::Point         pt1,                // First end-point of line
  cv::Point         pt2                 // Second end-point of line
  const cv::Scalar& color,              // Color, BGR form
  int               lineType = 8,       // Connectedness, 4 or 8
  int               shift    = 0        // Bits of radius to treat as fraction
);
```

[28] The algorithm use by `cv::fillComvexPolyfillConvexPoly()` is actually somewhat more general than implied here. It will correctly draw any polygon whose contour intersects every horizontal line at most twice (though it is allowed for the top or bottom of the polygon to be flat with respect to the horizontal). Such a polygon is said to be "monotone with respect to the horizontal."

The function `cv::line()` draws a straight line from `pt1` to `pt2` in the image `img`. Lines are automatically clipped by the image boundaries.

cv::rectangle()

```
void rectangle(
  cv::Mat&          img,                // Image to be drawn on
  cv::Point         pt1,                // First corner of rectangle
  cv::Point         pt2                 // Opposite corner of rectangle
  const cv::Scalar& color,              // Color, BGR form
  int               lineType = 8,       // Connectedness, 4 or 8
  int               shift    = 0        // Bits of radius to treat as fraction
);

void rectangle(
  cv::Mat&          img,                // Image to be drawn on
  cv::Rect          r,                  // Rectangle to draw
  const cv::Scalar& color,              // Color, BGR form
  int               lineType = 8,       // Connectedness, 4 or 8
  int               shift    = 0        // Bits of radius to treat as fraction
);
```

The function `cv::rectangle()` draws a rectangle with corners `pt1` to `pt2` in the image `img`. An alternate form of `cv::rectangle()` allows the rectangle's location and size to be specified by a single `cv::Rect` argument `r`.

cv::polyLines()

```
void polyLines(
  cv::Mat&          img,                // Image to be drawn on
  const cv::Point*  pts,                // C-style array of arrays of points
  int               npts,               // Number of points in 'pts[i]'
  int               ncontours,          // Number of arrays in 'pts'
  bool              isClosed,           // If true, connect last and first points
  const cv::Scalar& color,              // Color, BGR form
  int               lineType = 8,       // Connectedness, 4 or 8
  int               shift    = 0        // Bits of radius to treat as fraction
);
```

This function draws any number of unfilled polygons. It can handle general polygons including polygons with self-intersections. The argument `ncontours` specifies how many different polygon contours there will be, and the argument `npts` is a C-style array that indicates how many points there are in each contour (i.e., `npts[i]` indicates how many points there are in polygon `i`). `pts` is a C-style array of C-style arrays containing all of the points in those polygons (i.e., `pts[i][j]` contains the j^{th} point in the i^{th} polygon.) Polygons are not assumed to be closed. If the argument `isClosed` is `true`, then a segment from the last element of `pts[i][]` to the first element will be assumed. Otherwise the contour is taken to be an open contour containing only `npts[i]-1` segments between the `npts[i]` points listed.

cv::LineIterator()

The `cv::LineIterator` object is an iterator that is used to get each pixel of a raster line in sequence. It is another one of those "objects that do stuff." The constructor for the line iterator takes the two endpoints for the line as well as a line type specifier and an additional Boolean that indicates which direction the line should be traversed.

```
LineIterator::LineIterator(
  cv::Mat&          img,                // Image to be drawn on
  cv::Point         pt1,                // First end-point of line
  cv::Point         pt2                 // Second end-point of line
  int               lineType = 8,       // Connectedness, 4 or 8
  bool              leftToRight = false // If true, always start steps on the left
);
```

Once initialized, the number of pixels in the line is stored in the member integer `cv::LineIterator::count`. The overloaded dereferencing operator `cv::LineIterator::operator*()` returns a pointer of type `uchar*`, which points to the "current" pixel. The current pixel starts at one end of the line, and is incremented by means of the overloaded increment operator `cv::LineIterator::operator++()`. The actual traversal is done according to the Bresenham algorithm mentioned earlier.

The purpose of the `cv::LineIterator` is to make it possible for you to take some specific action on each pixel along the line. This is particularly handy when creating special effects such as switching the color of a pixel from black to white and white to black (i.e., an XOR operation on a binary image).

When accessing an individual "pixel," remember that this pixel may have one or many channels and it might be any kind of image depth. The return value from the dereferencing operator is always `uchar*`, so you are responsible for casting that pointer to the correct type. For example, if your image were a three-channel image of 32-bit floating-point numbers, and your iterator were called `iter`, then you would want to cast the return (pointer) value of the dereferencing operator like this: `(Vec3f*)*iter`.

> The style of the overloaded dereferencing operator `cv::LineIterator::operator*()` is slightly different than what you are probably used to from libraries like STL. The difference is that the return value from the iterator is itself a pointer, so the iterator itself behaves not like a pointer, but like a pointer to a pointer.

Fonts and Text

One additional form of drawing is to draw text. Of course, text creates its own set of complexities, but—as always with this sort of thing—OpenCV is more concerned with providing a simple "down and dirty" solution that will work for simple cases than a robust, complex solution (which would be redundant anyway given the capabilities of other libraries).

Table 4-10: Text drawing functions

Function	Description
`cv::putText()`	Draw the specified text in an image
`cv::getTextSize()`	Determine the width and height of a text string

cv::putText()

```
void cv::putText(
  cv::Mat&      img,                    // Image to be drawn on
  const string& text,                   // String to write (often from cv::format)
  cv::Point     origin,                 // Upper-left corner of text box
  int           fontFace,               // Font to use (e.g., cv::FONT_HERSHEY_PLAIN)
  double        fontScale,              // Scale of font (not "point", but multiple)
  cv::Scalar    color,                  // Color, RGB form
  int           thickness = 1,          // Thickness of line
  int           lineType  = 8,          // Connectedness, 4 or 8
  bool          bottomLeftOrigin = false  // If true, measure 'origin' from lower left
);
```

OpenCV has one main routine, called `cv::putText()` that just throws some text onto an image. The text indicated by `text` is printed with its upper-left corner of the text box at `origin` and in the color indicated by `color`, unless the `bottomLeftOrigin` flag is `true`, in which case the lower-left corner of the text box is located at `origin`. The font used is selected by the `fontFace` argument, which can be any of those listed in **Error! Reference source not found.**.

Table 4-11: Available fonts (all are variations of Hershey)

Identifier	Description
`cv::FONT_HERSHEY_SIMPLEX`	Normal size sans-serif
`cv::FONT_HERSHEY_PLAIN`	Small size sans-serif
`cv::FONT_HERSHEY_DUPLEX`	Normal size sans-serif, more complex than `cv::FONT_HERSHEY_SIMPLEX`
`cv::FONT_HERSHEY_COMPLEX`	Normal size serif, more complex than `cv::FONT_HERSHEY_DUPLEX`
`cv::FONT_HERSHEY_TRIPLEX`	Normal size serif, more complex than `cv::FONT_HERSHEY_COMPLEX`
`cv::FONT_HERSHEY_COMPLEX_SMALL`	Smaller version of `cv::FONT_HERSHEY_COMPLEX`
`cv::FONT_HERSHEY_SCRIPT_SIMPLEX`	Handwriting style
`cv::FONT_HERSHEY_SCRIPT_COMPLEX`	More complex variant of `cv::FONT_HERSHEY_SCRIPT_SIMPLEX`

Any of the font names listed in **Error! Reference source not found.** can also be combined (with an OR operator) with `cv::FONT_HERSHEY_ITALIC` to render the indicated font in italics. Each font has a "natural" size. When `fontScale` is not 1.0, then the font size is scaled by this number before drawing.

Figure 4-9: The eight fonts of Table 4-11, with the origin of each line separated from the vertical by 30 pixels

cv::getTextSize()

```
cv::Size cv::getTextSize(
  const string& text,
  cv::Point     origin,
  int           fontFace,
  double        fontScale,
  int           thickness,
  int*          baseLine
);
```

The function `cv::getTextSize()` answers the question of how big some text would be if you were to draw it (with some set of parameters), without actually drawing it on an image. The only novel argument to

`cv::getTextSize()` is `baseLine`, which is actually an output parameter. `baseLine` is the y-coordinate of the text baseline relative to the bottom-most point in the text.[29]

Data Persistence

OpenCV provides a mechanism for serializing and de-serializing its various data types to and from disk in either YAML or XML format, which can load and store any number of OpenCV data objects (including basic types like `int`, `float`, etc.) in a single file. There functions are separate from the special functions we saw earlier in the chapter that handle the more specialized situation of loading and saving image files and video data. In this section, we will focus on general object persistence: reading and writing matrices, OpenCV structures, configuration, and log files.

The basic mechanism for reading and writing files is the `cv::FileStorage` object. This object essentially represents a file on disk, but does so in a manner that makes accessing the data represented in the file easy and natural.

Writing to a `cv::FileStorage`

```
FileStorage::FileStorage();
FileStorage::FileStorage( string fileName, int flag );
```

The `cv::FileStorage` object is a representation of an XML or YML data file. You can create it and pass a file name to the constructor, or you can just create an unopened storage object with the default constructor and open the file later with `cv::FileStorage::open()` where the `flag` argument should be either `cv::FileStorage::WRITE` or `cv::FileStorage::APPEND`.

```
FileStorage::open( string fileName, int flag );
```

Once you have opened the file you want to write to, you can write using the operator `cv::FileStorage::operator<<()` in the same manner you might write to `stdout` with an STL stream. Internally, however, there is quite a bit more going on when you write in this manner.

Data inside of the `cv::FileStorage` is stored in one of two forms, either as a "mapping" (i.e., key-value pairs) or a "sequence" (which is a series of unnamed entries). At the top level, the data you write to the file storage is all a mapping, and inside of that mapping you can place other mappings or sequences, and mappings or sequences inside of those as deep as you like.

```
myFileStorage << "someInteger" << 27;                    // save an integer
myFileStorage << "anArray" << cv::Mat::eye(3,3,CV::F32);  // save an array
```

To create a sequence entry, you first provide the string name for the entry, and then the entry itself. The entry can be a number (integer, float, etc.), a string, or any OpenCV data type.

If you would like to create a new mapping or sequence you can do so with the special characters "{" (for a mapping) or "[" (for a sequence). Once you have started the mapping or sequence, you can add new elements and then finally close the mapping or sequence with "}" or "]" (respectively).

```
myFileStorage << "theCat" << "{";
myFileStorage << "fur" << "gray" << "eyes" << "green" << "weightLbs" << 16;
myFileStorage << "}";
```

Once you have created a mapping, you enter each element with a name and the data following, just as you did for the top-level mapping. If you create a sequence, you simply enter the new data one item after another until you close the sequence.

```
myFileStorage << "theTeam" << "[";
```

[29] The "baseline" is the line on which the bottoms of characters such as a and b are aligned. Characters such as y and g hang below the baseline.

```
myFileStorage << ”eddie” << ”tom” << ”scott”;
myFileStorage << ”]”;
```

Once you are completely done writing, you close the file with the `cv::FileStorage::release()` member function.

Here is an explicit code sample from the OpenCV documentation:

```
#include "opencv2/opencv.hpp"
#include <time.h>

int main(int, char** argv)
{
    cv::FileStorage fs("test.yml", cv::FileStorage::WRITE);

    fs << "frameCount" << 5;
    time_t rawtime; time(&rawtime);
    fs << "calibrationDate" << asctime(localtime(&rawtime));
    cv::Mat cameraMatrix = (
      cv::Mat_<double>(3,3)
      << 1000, 0, 320, 0, 1000, 240, 0, 0, 1
    );
    cv::Mat distCoeffs = (
      cv::Mat_<double>(5,1)
      << 0.1, 0.01, -0.001, 0, 0
    );
    fs << "cameraMatrix" << cameraMatrix << "distCoeffs" << distCoeffs;
    fs << "features" << "[";
    for( int i = 0; i < 3; i++ )
    {
        int x = rand() % 640;
        int y = rand() % 480;
        uchar lbp = rand() % 256;

        fs << "{:" << "x" << x << "y" << y << "lbp" << "[:";
        for( int j = 0; j < 8; j++ )
            fs << ((lbp >> j) & 1);
        fs << "]" << "}";
    }
    fs << "]";
    fs.release();
    return 0;
}
```

The result of running this program would be a YML file with the following contents:

```
%YAML:1.0
frameCount: 5
calibrationDate: "Fri Jun 17 14:09:29 2011\n"
cameraMatrix: !!opencv-matrix
   rows: 3
   cols: 3
   dt: d
   data: [ 1000., 0., 320., 0., 1000., 240., 0., 0., 1. ]
distCoeffs: !!opencv-matrix
   rows: 5
   cols: 1
   dt: d
   data: [ 1.0000000000000001e-01, 1.0000000000000000e-02,
       -1.0000000000000000e-03, 0., 0. ]
features:
   - { x:167, y:49, lbp:[ 1, 0, 0, 1, 1, 0, 1, 1 ] }
   - { x:298, y:130, lbp:[ 0, 0, 0, 1, 0, 0, 1, 1 ] }
   - { x:344, y:158, lbp:[ 1, 1, 0, 0, 0, 0, 1, 0 ] }
```

In the example code, you will notice that sometimes all of the data in a mapping or sequence is stored on a single line and other times it is stored with one element per line. This is not an automatic formatting behavior. Instead, it is created by a variant of the mapping and sequence creation strings: "{:" and ":}" mappings, and "[:" and ":]" for sequences. This feature is only meaningful for YML output; if the output file is XML, this nuance is ignored and the mapping or sequence is stored as it would have been without the variant.

Reading from a `cv::FileStorage`

```
FileStorage::FileStorage( string fileName, int flag );
```

The `cv::FileStorage` can be opened for reading the same way it is opened for writing, except that the `flag` argument should be set to `cv::FileStorage::READ`. As with writing, you can also create an unopened file storage object with the default constructor and open it later with `cv::FileStorage::open()`.

```
FileStorage::open( string fileName, int flag );
```

Once the file has been opened, the data can be read with either the overloaded array operator `cv::FileStorage::operator[]()` or with the iterator `cv::FileNodeIterator`. Once you are completely done reading, you then close the file with the `cv::FileStorage::release()` member function.

To read from a mapping, the `cv::FileStorage::operator[]()` is passed the string key associated with the desired object. To read from a sequence, the same operator can be called with an integer argument instead. The return value of this operator is not the desired object, however; it is an object of type `cv::FileNode`, which represents the value that goes with the given key in an abstract form.

`cv::FileNode`

Once you have a `cv::FileNode` object, you can do one of several things with it. If it represents an object (or a number or a string) you can just load it into a variable of the appropriate type with the overloaded extraction operator `cv::FileNode::operator>>()`.

```
cv::Mat anArray;
myFileStorage["calibrationMatrix"] >> anArray;
```

The `cv::FileNode` object also supports direct casting to any of the basic data types.

```
int aNumber;
myFileStorage["someInteger"] >> aNumber;
```

is equivalent to:

```
int aNumber;
aNumber = (int)myFileStorage["someInteger"];
```

As mentioned earlier, there is also an iterator for moving through file nodes that can be used as well. Given a `cv::FileNode` object, the member functions `cv::FileNode::begin()` and `cv::FileNode::end()` have their usual interpretations as providing the first and "after last" iterator for either a mapping or a sequence. The iterator, on dereferencing with the usual overloaded dereferencing operator `cv::FileNodeIterator::operator*()`, will return another `cv::FileNode` object. Such iterators support the usual incrementing and decrementing operators. If the iterator was iterating through a mapping, then the returned `cv::FileNode` object will have a name that can be retrieved with `cv::FileNode::name()`.

Table 4-12: Member functions of `cv::FileNode`

Example	Description
`cv::FileNode fn()`	File node object default constructor

`cv::FileNode fn1( fn0 )`	File node object copy constructor, creates a node fn1 from a node fn0
`cv::FileNode fn1( fs, node )`	File node constructor that creates a C++ style `cv::FileNode` object from a C-style `CvFileStorage*` pointer `fs` and a C-style `CvFileNode*` pointer node
`fn[ (string)key ]` `fn[ (char*)key ]`	STL string or C-string accessor for named child (of mapping node), converts key to the appropriate child node
`fn[ (int)id ]`	Accessor for numbered child (of sequence node), converts ID to the appropriate child node
`fn.type()`	Returns node type enum
`fn.empty()`	Determine if node is empty
`fn.isNone()`	Determine if node has value "None"
`fn.isSeq()`	Determine if node is a sequence
`fn.isMap()`	Determine if node is a mapping

`fn.isInt()`
`fn.isReal()`
`fn.isString()`
	Determine if node is an integer, a floating-point number, or a string (respectively)

`fn.name()`	Return nodes name if node is a child of a mapping
`size_t sz=fn.size()`	Return size of node (in bytes)

`(int)fn`
`(float)fn`
`(double)fn`
`(string)fn`
	Extract the value from a node containing an integer, 32-bit float, 64-bit float, or string (respectively)

Of the methods in Table 4-12, one requires special clarification: `cv::FileNode::type()`. The returned value is an enumerated type defined in the class `cv::FileNode`. The possible values are given in Table 4-13.

Table 4-13: Possible return values for `cv::FileNode::type()`

Example	**Description**
`cv::FileNode::NONE = 0`	Node is of type "None"
`cv::FileNode::INT = 1`	Node contains an integer

`cv::FileNode::REAL = 2` `cv::FileNode::FLOAT = 2`	Node contains a floating-point number[30]
`cv::FileNode::STR = 3` `cv::FileNode::STRING = 3`	Node contains a string
`cv::FileNode::REF = 4`	Node contains a reference (i.e., a compound object)
`cv::FileNode::SEQ = 5`	Node is itself a sequence of other nodes
`cv::FileNode::MAP = 6`	Node is itself a mapping of other nodes
`cv::FileNode::FLOW = 8`	Node is a Compact representation of a sequence or mapping
`cv::FileNode::USER = 16`	Registered object (e.g., a Matrix)
`cv::FileNode::EMPTY = 32`	Node has no value assigned to it
`cv::FileNode::NAMED = 64`	Node is a child of a mapping (i.e., it has a name)

It is worth noting that the last four `enum` values are powers of two starting at 8. This is because a node may have any or all of these properties in addition to one of the first eight listed types.

The following code example (also from the OpenCV documentation) shows how we could read the file we wrote previously:

```
cv::FileStorage fs2("test.yml", cv::FileStorage::READ);

// first method: use (type) operator on FileNode.
int frameCount = (int)fs2["frameCount"];

std::string date;
// second method: use cv::FileNode::operator >>
fs2["calibrationDate"] >> date;

cv::Mat cameraMatrix2, distCoeffs2;
fs2["cameraMatrix"] >> cameraMatrix2;
fs2["distCoeffs"] >> distCoeffs2;

cout << "frameCount: " << frameCount << endl
     << "calibration date: " << date << endl
     << "camera matrix: " << cameraMatrix2 << endl
     << "distortion coeffs: " << distCoeffs2 << endl;

cv::FileNode features = fs2["features"];
cv::FileNodeIterator it = features.begin(), it_end = features.end();
int idx = 0;
std::vector<uchar> lbpval;

// iterate through a sequence using FileNodeIterator
for( ; it != it_end; ++it, idx++ )
```

[30] Note that the floating-point types are not distinguished. This is a somewhat subtle point. Recall that XML and YML are ASCII text file formats. As a result, all floating-point numbers are of no specific precision until cast to an internal machine variable type. So, at the time of parsing, all floating-point numbers are represented only as an abstract floating-point type.

```
{
    cout << "feature #" << idx << ": ";
    cout << "x=" << (int)(*it)["x"] << ", y=" << (int)(*it)["y"] << ", lbp: (";
    // you can also easily read numerical arrays using FileNode >> std::vector operator.
    (*it)["lbp"] >> lbpval;
    for( int i = 0; i < (int)lbpval.size(); i++ )
        cout << " " << (int)lbpval[i];
    cout << ")" << endl;
}
fs.release();
```

Summary

We have seen that OpenCV provides a number of ways to bring computer vision programs to the screen. The native HighGUI tools are convenient and easy to use, but not so great for functionality or final polish.

For a little more capability, the Qt-based HighGUI tools add buttons and some nice gadgets for manipulating your image on the screen—which is very helpful for debugging, parameter tuning, and for studying the subtle effects of changes in your program. Because those methods lack extensibility and are likely unsuitable for the production of professional applications, we went on to look at a few examples of how you might combine OpenCV with existing fully featured GUI toolkits.

We then touched on some other important functions you will need for manipulating arrays in basic ways and went on to drawing functions. Finally, we reviewed how OpenCV can store and retrieve objects from files.

Exercises

1. This chapter completes our introduction to basic I/O programming and data structures in OpenCV. The following exercises build on this knowledge and create useful utilities for later use.
   a) Create a program that (1) reads frames from a video, (2) turns the result to grayscale, and (3) performs Canny edge detection on the image. Display all three stages of processing in three different windows, with each window appropriately named for its function.
   b) Display all three stages of processing in one image.

   Hint: Create another image of the same height but three times the width as the video frame. Copy the images into this, either by using pointers or (more cleverly) by creating three new cv::Mat objects that reference into the beginning of, to one-third, and to two-thirds of the way into, the image data and then copying directly into these (i.e. with the `copyTo()` function).

   c) Write appropriate text labels describing the processing in each of the three slots.
2. Create a program that reads in and displays an image. When the user's mouse clicks on the image, read in the corresponding pixel (blue, green, red) values and write those values as text to the screen at the mouse location.
   a) For the program of Exercise 1b, display the mouse coordinates of the individual image when clicking anywhere within the three-image display.
3. Create a program that reads in and displays an image.
   a) Allow the user to select a rectangular region in the image by drawing a rectangle with the mouse button held down, and highlight the region when the mouse button is released. Be careful to save an image copy in memory so that your drawing into the image does not destroy the original values there. The next mouse click should start the process all over again from the original image.
   b) In a separate window, use the drawing functions to draw a graph in blue, green, and red for how many pixels of each value were found in the selected box. This is the *color histogram* of that color region. The *x*-axis should be eight bins that represent pixel values falling within the ranges 0–31,

32–63,…, 223–255. The *y*-axis should be counts of the number of pixels that were found in that bin range. Do this for each color channel, BGR.

4. Make an application that reads and displays a video and is controlled by sliders. One slider will control the position within the video from start to end in 10 increments; another binary slider should control pause/unpause. Label both sliders appropriately.
   a) Do this using the built in HighGUI native toolkit functions.
   b) Do this using the Qt Backend.
5. Create your own simple paint program.
   a) Write a program that creates an image, sets it to 0, and then displays it. Allow the user to draw lines, circles, ellipses, and polygons on the image using the left mouse button. Create an eraser function when the right mouse button is held down.
   b) Allow "logical drawing" by allowing the user to set a slider setting to AND, OR, and XOR. That is, if the setting is AND then the drawing will appear only when it crosses pixels greater than 0 (and so on for the other logical functions).
6. Write a program that creates an image, sets it to 0, and then displays it. Hitting Enter should fix the label at the spot it was typed.
   a) When the user clicks on a location, he or she can type in a label there.
   b) Enable Backspace during editing and provide for an abort key.
7. Perspective transform.
   a) Write a program that reads in an image and uses the numbers 1–9 on the keypad to control a perspective transformation matrix (refer to our discussion of the `cv::warpPerspective()` in the Dense Perspective Transform section of Chapter 6). Tapping any number should increment the corresponding cell in the perspective transform matrix; tapping with the Shift key depressed should decrement the number associated with that cell (stopping at 0). Each time a number is changed, display the results in two images: the raw image and the transformed image.
   b) Add functionality to zoom in or out.
   c) Add functionality to rotate the image.
8. Face fun. Go to the *…/samples/cpp/* directory and build the *facedetect.cpp* code. Draw a skull image (or find one on the Web) and store it to disk. Modify the *facedetect* program to load in the image of the skull.
   a) When a face rectangle is detected, draw the skull in that rectangle.

   Hint: `cv::convertImage()` can convert the size of the image, or you could look up the `cv::resize()` function. One may then set the ROI to the rectangle and use `copyTo()` to copy the properly resized image there.

   a) Add a slider with 10 settings corresponding to 0.0 to 1.0. Use this slider to alpha blend the skull over the face rectangle using the `cv::addWeighted()` function.
9. Image stabilization. Go to the *.../samples/cpp/* directory and build the *lkdemo.cpp* code (this code does motion tracking, also known as *optical flow*). Create and display a video image in a much larger window image. Move the camera slightly but use the optical flow vectors to display the image in the same place within the larger window. This is a rudimentary image stabilization technique.
10. Create a structure of an integer, a `cv::Point2i` and a `cv::Rect`; call it `my_struct`.
    a) Write two functions:

```
void write_my_struct(
  cv::FileStorage* fs,
  const char*      name,
  my_struct*       ms
);
```

    and:

```
void read_my_struct(
  cv::FileStorage* fs,
  CvFileNode*      ms_node,
  my_struct*       ms
);
```

Use them to write and read `my_struct`.

b) Write and read an array of 10 `my_struct` structures.

5

Filters and Convolution

Overview

At this point, we have all of the basics at our disposal. We understand the structure of the library as well as the basic data structures it uses to represent images. We understand the HighGUI interface and can actually run a program and display our results on the screen. Now that we understand these primitive methods required to manipulate image structures, we are ready to learn some more sophisticated operations.

We will now move on to higher-level methods that treat the images as images, and not just as arrays of colored (or grayscale) values. When we say "image processing," we mean just that: using higher-level operators that are defined on image structures in order to accomplish tasks whose meaning is naturally defined in the context of graphical, visual images.

Before We Begin

There are a couple of important concepts we will need throughout this chapter, so it is worth taking a moment to review these ideas before we dig into the specific image processing functions that make up the bulk of this chapter. We will need to learn two main concepts: first, we'll need to understand filters (also called kernels) and how they are handled in OpenCV. Next, we'll take a look at how boundaries are handled, and what happens when OpenCV needs to compute something that is a function of the area around a pixel if that area spills off of the edge of the image.

Filters, Kernels, and Convolution

Most of the functions we will discuss in this chapter are special cases of a general concept called *image filtering*. A filter is any algorithm that starts with some image $I(x, y)$ and computes a new image $I'(x, y)$ by computing for each pixel location x, y in I' some function of the pixels in I that are in some area around that point. The template that defines both this area's shape, as well as how the elements of that area are combined, is called a *filter* or a *kernel*.[1] In this chapter, many of the important kernels we encounter will be

[1] These two terms can be considered essentially interchangeable for our purposes. The signal processing community typically prefers the word *filter*, while the mathematical community tends to prefer *kernel*.

linear kernels. This means that the value assigned to point x, y in I' can be expressed as a weighted sum of the points around (and usually including) x, y in I[2]. If you like equations, this can be written as:

$$I'(x,y) = \sum_{i,j \in kernel} k_{i,j} \cdot I(x+i, y+j).$$

This basically says that for some kernel of whatever size (e.g., 5-*by*-5), we should sum over the area of the kernel, and for each pair i, j (representing one point in the kernel), we should add a contribution equal to some value $k_{i,j}$ multiplied by the value of the pixel in I that is offset from x, y by i, j. The size of the array $k_{i,j}$ is called the *support* of the kernel.[3] Any filter which can be expressed in this way (i.e., with a linear kernel) is also known as *convolutions*.

It is often convenient (and more intuitive) to represent the kernel graphically as an array of the values of $k_{i,j}$ (Figure 5-1). We will typically use this representation throughout the book when it is necessary to represent a kernel.

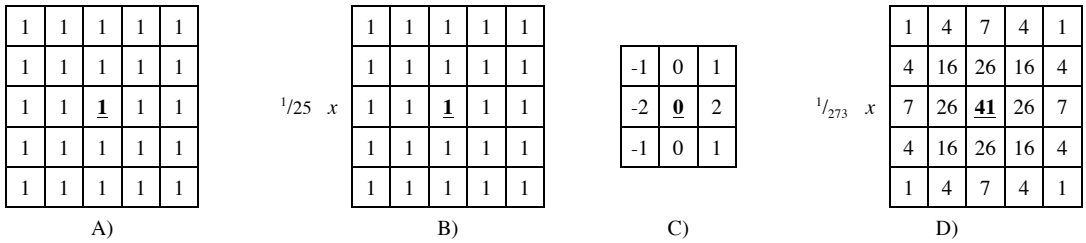


Figure 5-1: (a) A 5-by-5 box kernel; (b) a normalized 5-by-5 box kernel; (c) a 3-by-3 Sobel "x-derivative" kernel; and (d) a 5-by-5 normalized Gaussian kernel; in each case, the "anchor" is represented in bold.

Anchor Points

Each kernel shown in Figure 5-1 has one value depicted in bold. This is the *anchor point* of the kernel. This indicates how the kernel is to be aligned with the source image. For example, in Figure 5-1 (d), the number 41 appears in bold. This means that in the summation used to compute $I'(x,y)$, it is $I(x,y)$ that is multiplied by $^{41}/_{273}$ (and similarly, the terms corresponding to $I(x-1,y)$ and $I(x+1,y)$ are multiplied by $^{26}/_{273}$).

Border Extrapolation and Boundary Conditions

Something that will come up with some frequency as we look at how images are processed in OpenCV is the issue of how borders are handled. Unlike some other image handling libraries,[4] the filtering operations in OpenCV (e.g., `cv::blur()`, `cv::erode()`, `cv::dilate()`, etc.) produce output images of the same size as the input. To achieve that result, OpenCV creates "virtual" pixels outside of the image at the borders. You can see this would be necessary for operation like `cv::blur()`, which is going to take all of the pixels in a neighborhood of some point and average them to determine a new value for that point. How could a meaningful result be computed for an edge pixel that does not have the correct number of neighbors? In fact, it will turn out that in the absence of any clearly "right" way of handling this, we will often find ourselves explicitly asserting how this issue is to be resolved in any given context.

---

[2] An example of a nonlinear kernel that comes up relatively often is the *median filter*, which replaces the pixel at x, y with the median value inside of the kernel area.

[3] For technical purists, the "support" of the kernel actually consists of only the nonzero portion of the kernel array.

[4] E.g., MATLAB.

Making Borders Yourself

Most of the library functions you will use will create these virtual pixels for you. In that context, you will only need to tell the particular function how you would like those pixels created.[5] Just the same, in order to know what the options you have mean, it is best to take a look at the function that allows you to explicitly create "padded" images that use one method or another.

The function that does this is `cv::copyMakeBorder()`. Given an image you want to pad out, and a second image that is somewhat larger, you can ask `cv::copyMakeBorder()` to fill all of the pixels in the larger image in one way or another.

```
void cv::copyMakeBorder(
  cv::InputArray    src,                    // Input Image
  cv::OutputArray   dst,                    // Result image
  int               top,                    // Top side padding (pixels)
  int               bottom,                 // Bottom side padding (pixels)
  int               left,                   // Left side padding (pixels)
  int               right,                  // Right side padding (pixels)
  int               borderType,             // Pixel extrapolation method
  const cv::Scalar& value = cv::Scalar()    // Used for constant borders
);
```

The first two arguments to `cv::copyMakeBorder()` are the smaller source image and the larger destination image. The next four arguments specify how many pixels of padding are to be added to the source image on the `top`, `bottom`, `left`, and `right` edges. The next argument `borderType` actually tells `cv::copyMakeBorder()` how to determine the correct values to assign to the padded pixels (as shown in Figure 5-2).

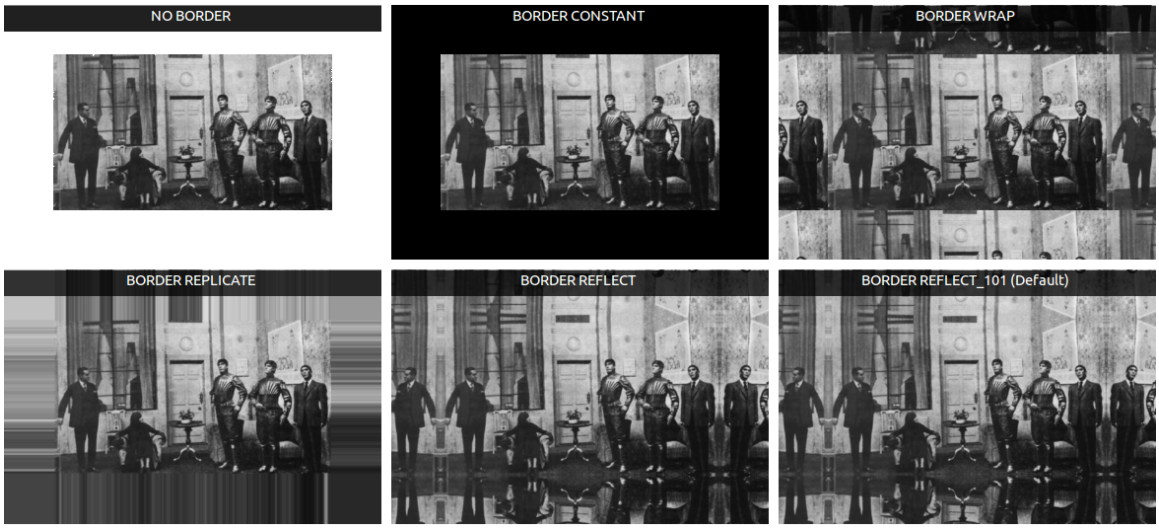


Figure 5-2: The same image is shown padded using each of the six different `borderType` options available to `cv::copyMakeBorder()` (the "NO BORDER" image in the upper-left is the original for comparison)

To understand what each option does in detail, it is useful to consider an extremely zoomed in section at the edge of each image (Figure 5-3).

[5] Actually, the pixels are usually not even really created, but rather they are just "effectively created" by the generation of the correct boundary conditions in the evaluation of the particular function in question.

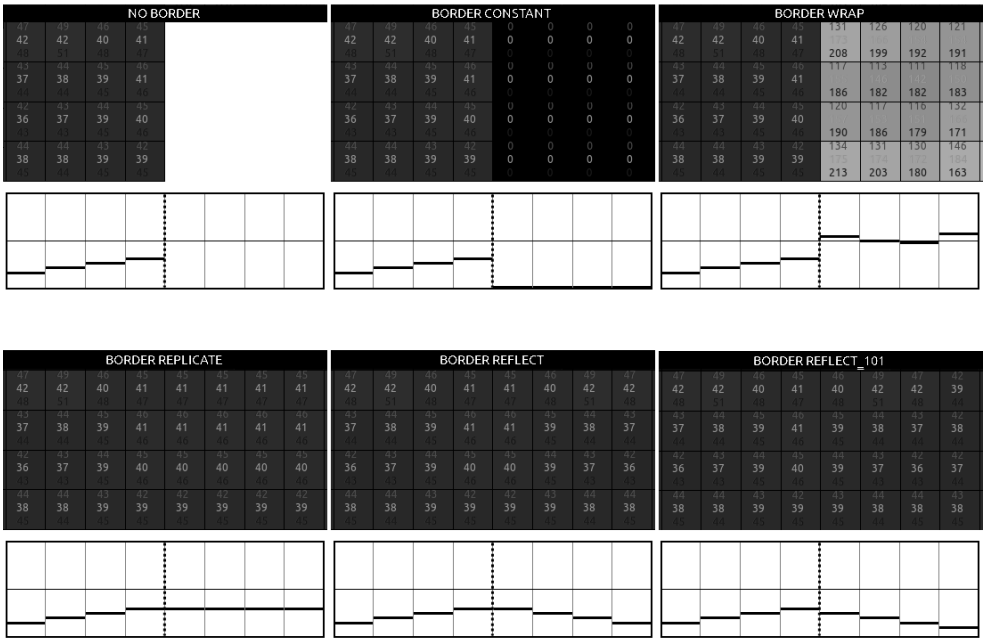


Figure 5-3:. An extreme zoom in at the left side of each image; for each case, the actual pixel values are shown, as well as a schematic representation; the vertical dotted line in the schematic represents the edge of the original image

As you can see by inspecting the figures, some of the available options are quite different. The first option, a constant border `cv::BORDER_CONSTANT` sets all of the pixels in the border region to some fixed value. This value is set by the `value` argument to `cv::copyMakeBorder()`. (In Figure 5-2 and Figure 5-3, this value happens to be `cv::Scalar(0,0,0)`) The next option is to wrap around `cv::BORDER_WRAP`, assigning each pixel that is a distance n off of the edge of the image the value of the pixel that is a distance n in from the opposite edge. The replicate option `cv::BORDER_REPLICATE` assigns every pixel off of the edge the same value as the pixel on that edge. Finally, there are two slightly different forms of reflection available: `cv::BORDER_REFLECT` and `cv::BORDER_REFLECT_101`. The first assigns each pixel that is a distance `n` off of the edge of the image the value of the pixel that is a distance n in from that same edge. In contrast, `cv::BORDER_REFLECT_101` assigns each pixel that is a distance n off of the edge of the image the value of the pixel that is a distance $n + 1$ in from that same edge (with the result that the very edge pixel is not replicated). In most cases, `cv::BORDER_REFLECT_101` is the default behavior for OpenCV methods. The value of `cv::BORDER_DEFAULT` resolves to `cv::BORDER_REFLECT_101`. Table 5-1 summarizes these options.

Table 5-1: `borderType` options available to `cv::copyMakeBorder()`, as well as many other functions that need to implicitly create boundary conditions

Border Type	**Effect**
`cv::BORDER_CONSTANT`	Extend pixels by using a supplied (constant) value
`cv::BORDER_WRAP`	Extend pixels by replicating from opposite side
`cv::BORDER_REPLICATE`	Extend pixels by copying edge pixel
`cv::BORDER_REFLECT`	Extend pixels by reflection
`cv::BORDER_REFLECT_101`	Extend pixels by reflection, edge pixel is not “doubled”
`cv::BORDER_DEFAULT`	Alias for `cv::BORDER_REFLECT_101`

Manual Extrapolation

On some occasions, you will want to compute the location of the reference pixel to which a particular off-the-edge pixel is referred. For example, given an image of width w and height h, you might want to know what pixel in that image is being used to assign a value to virtual pixel $(w + dx, h + dy)$. Though this operation is essentially extrapolation, the function that computes such a result for you is (somewhat confusingly) called `cv::borderInterpolate()`:

```
int cv::borderInterpolate(            // Returns coordinate of "donor" pixel
  int p,                              // 0-based coordinate of extrapolated pixel
  int len,                            // Length of array (on relevant axis)
  int borderType                      // Pixel extrapolation method
);
```

The function `cv::borderInterpolate()` computes the extrapolation for one dimension at a time. It takes a coordinate `p`, a length `len` (which is the actual size of the image in the associated direction), and a `borderType` value. So, for example, you could compute the value of a particular pixel in an image under a mixed set of boundary conditions, using `BORDER_REFLECT_101` in one dimension, and `BORDER_WRAP` in another:

```
float val = img.at<float>(
  cv::borderInterpolate( 100, img.rows, BORDER_REFLECT_101 ),
  cv::borderInterpolate(  -5, img.cols, BORDER_WRAP )
);
```

This function is typically used internally to OpenCV, for example inside of `cv::copyMakeBorder` or the `cv::FilterEngine` class (more on that later), but it can come in handy in your own algorithms as well. The possible values for `borderType` are exactly the same as those used by `cv::copyMakeBorder`. Throughout this chapter, we will encounter functions that take a `borderType` argument; in all of those cases, they take the same list of argument.

Threshold Operations

Frequently we have done many layers of processing steps and want either to make a final decision about the pixels in an image or to categorically reject those pixels below or above some value while keeping the others. The OpenCV function `cv::threshold()` accomplishes these tasks (see survey [Sezgin04]). The basic idea is that an array is given, along with a threshold, and then something happens to every element of the array depending on whether it is below or above the threshold. If you like, you can think of threshold as a very simple convolution operation that uses a 1-by-1 kernel and performs one of several nonlinear operations on that one pixel:[6]

```
double cv::threshold(
  cv::InputArray    src,                    // Input Image
  cv::OutputArray   dst,                    // Result image
  double            thresh,                 // Threshold value
  double            maxValue,               // Max value for upward operations
  int               thresholdType           // Threshold type to use (see Example 5-2)
);
```

As shown in Table 5-2, each threshold type corresponds to a particular comparison operation between the i^{th} source pixel (src_i) and the threshold `thresh`. Depending on the relationship between the source pixel and the threshold, the destination pixel dst_i may be set to 0, the src_i, or the given maximum value `maxValue`.

[6] The utility of this point of view will become clearer as we proceed through this chapter, and look at other more complex convolutions. Many useful operations in computer vision can be expressed as a sequence of common convolutions, and more often than not, the last one of those convolutions is a threshold operation.

Table 5-2: `thresholdType` options for `cv::threshold()`

Threshold type	Operation
`cv::THRESH_BINARY`	$DST_I = (SRC_I > THRESH)\ ?\ MAXVALUE : 0$
`cv::THRESH_BINARY_INV`	$DST_I = (SRC_I > THRESH)\ ?\ 0 : MAXVALUE$
`cv::THRESH_TRUNC`	$DST_I = (SRC_I > THRESH)\ ?\ THRESH : SRC_I$
`cv::THRESH_TOZERO`	$DST_I = (SRC_I > THRESH)\ ?\ SRC_I : 0$
`cv::THRESH_TOZERO_INV`	$DST_I = (SRC_I > THRESH)\ ?\ 0 : SRC_I$

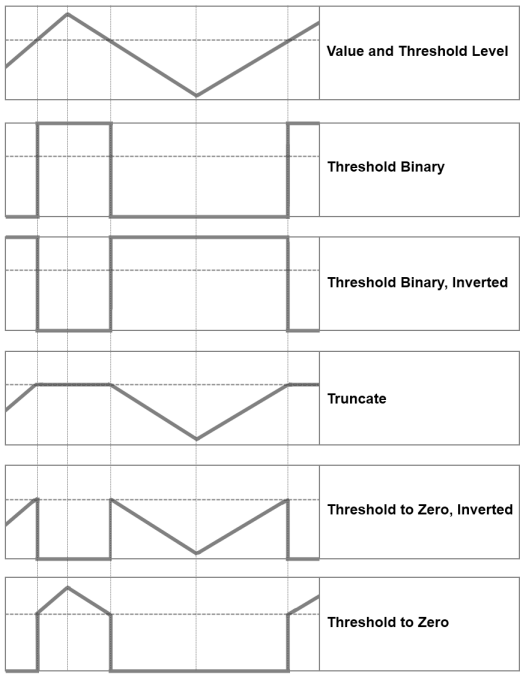


Figure 5-4 should help to clarify the exact implications of each of the available values for `thresholdType`, the thresholding operation.

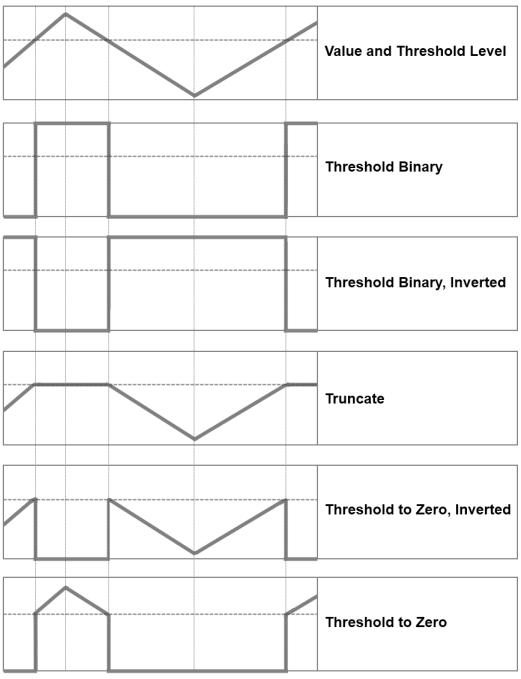


Figure 5-4: Results of varying the threshold type in `cv::threshold()`; the horizontal line through each chart represents a particular threshold level applied to the top chart and its effect for each of the five types of threshold operations below

Let's look at a simple example. In Example 5-1, we sum all three channels of an image and then clip the result at `100`.

Example 5-1: Example code making use of `cv::threshold()`

```
#include <opencv2/opencv.hpp>
#include <iostream>
using namespace std;

void sum_rgb( const cv::Mat& src, cv::Mat& dst ) {

  // Split image onto the color planes.
  vector< cv::Mat> planes;
  cv::split(src, planes);

  cv::Mat b = planes[0], g = planes[1], r = planes[2], s;

  // Add equally weighted rgb values.
  cv::addWeighted( r, 1./3., g, 1./3., 0.0, s );
  cv::addWeighted( s, 1., b, 1./3., 0.0, s );

  // Truncate values above 100.
  cv::threshold( s, dst, 100, 100, cv::THRESH_TRUNC );
}

void help()
{
    cout << "Call: ./ch5_ex5_2 faceScene.jpg" << endl;
    cout << "Shows use of alpha blending (addWeighted) and threshold" << endl;
}

int main(int argc, char** argv)
```

```
{
    help();
    if(argc < 2) { cout << "specify input image" << endl; return -1; }

    // Load the image from the given file name.
    cv::Mat src = cv::imread( argv[1] ), dst;
    if( src.empty() ) { cout << "can not load " << argv[1] << endl; return -1; }
    sum_rgb( src, dst);

    // Create a named window with the name of the file and
    // show the image in the window
    cv::imshow( argv[1], dst );

    // Idle until the user hits any key.
    cv::waitKey(0);

    return 0;
}
```

Some important ideas are shown here. One thing is that we don't want to add directly into an 8-bit array (with the idea of normalizing next) because the higher bits will overflow. Instead, we use equally weighted addition of the three color channels (`cv::addWeighted()`); then the sum is truncated to saturate at the value of 100 for the return. Had we used a floating-point temporary image for `s` in Example 5-1, we could have substituted the code shown in Example 5-2 instead. Note that `cv::accumulate()` can accumulate 8-bit integer image types into a floating-point image.

Example 5-2: Alternative method to combine and threshold image planes

```
void sum_rgb( const cv::Mat& src, cv::Mat& dst ) {

  // Split image onto the color planes.
  vector<cv::Mat> planes;
  cv::split( src, planes );

  cv::Mat b = planes[0], g = planes[1], r = planes[2];

  // Accumulate separate planes, combine and threshold
  cv::Mat s = cv::Mat::zeros( b.size(), cv::F32 );
  cv::accumulate( b, s );
  cv::accumulate( g, s );
  cv::accumulate( r, s );

  // Truncate values above 100 and rescale into dst
  cv::threshold( s, s, 100, 100, cv::THRESH_TRUNC );
  s.convertTo(dst, b.type());
}
```

Otsu's Algorithm

It is also possible to have `cv::threshold()` attempt to determine the optimal value of the threshold for you. This is done by passing the special value `cv::THRESH_OTSU` as the value of `thresh`.

Briefly, Otsu's algorithm is to consider all possible thresholds, and to compute the variance σ_i^2 for each of the two classes of pixels (i.e., the class below the threshold and the class above it). Otsu's algorithm minimizes:

$$\sigma_w^2 \equiv w_1(t) \cdot \sigma_1^2 + w_2(t) \cdot \sigma_2^2,$$

where $w_1(t)$ and $w_2(t)$ are the relative weights for the two classes given by the relative (normalized) number of pixels in each class (that is, their probability) and σ_1^2 and σ_2^2 are the variances in each class. Some thought (think of an image with 2 colors) will convince you that minimizing the variance over both

classes is the same as maximizing the variance between the two classes. Because an exhaustive search of the space of possible thresholds is required, this is not a particularly fast process.

Adaptive Threshold

There is a modified threshold technique in which the threshold level is itself variable (across the image). In OpenCV, this method is implemented in the `cv::adaptiveThreshold()` [Jain86] function:

```
void cv::adaptiveThreshold(
  cv::InputArray    src,                    // Input Image
  cv::OutputArray   dst,                    // Result image
  double            maxValue,               // Max value for upward operations
  int               adaptiveMethod,         // Method to weight pixels in block by
  int               thresholdType           // Threshold type to use (see Example 5-2)
  int               blockSize,              // Block size
  double            C                       // Constant to offset sum over block by
);
```

`cv::adaptiveThreshold()` allows for two different adaptive threshold types depending on the settings of `adaptiveMethod`. In both cases, the *adaptive threshold T(x, y)* is set on a pixel-by-pixel basis by computing a weighted average of the *b*-by-*b* region around each pixel location minus a constant, where b is given by `blockSize` and the constant is given by `C`. If the method is set to `cv::ADAPTIVE_THRESH_MEAN_C`, then all pixels in the area are weighted equally. If it is set to `cv::ADAPTIVE_THRESH_GAUSSIAN_C`, then the pixels in the region around (x, y) are weighted according to a Gaussian function of their distance from that center point.

Finally, the parameter `thresholdType` is the same as for `cv::threshold()` shown in Table 5-2.

The adaptive threshold technique is useful when there are strong illumination or reflectance gradients that you need to threshold relative to the general intensity gradient. This function handles only single-channel 8-bit or floating-point images, and it requires that the source and destination images be distinct.

Example 5-3 shows source code for comparing `cv::adaptiveThreshold()` and `cv::threshold()`.

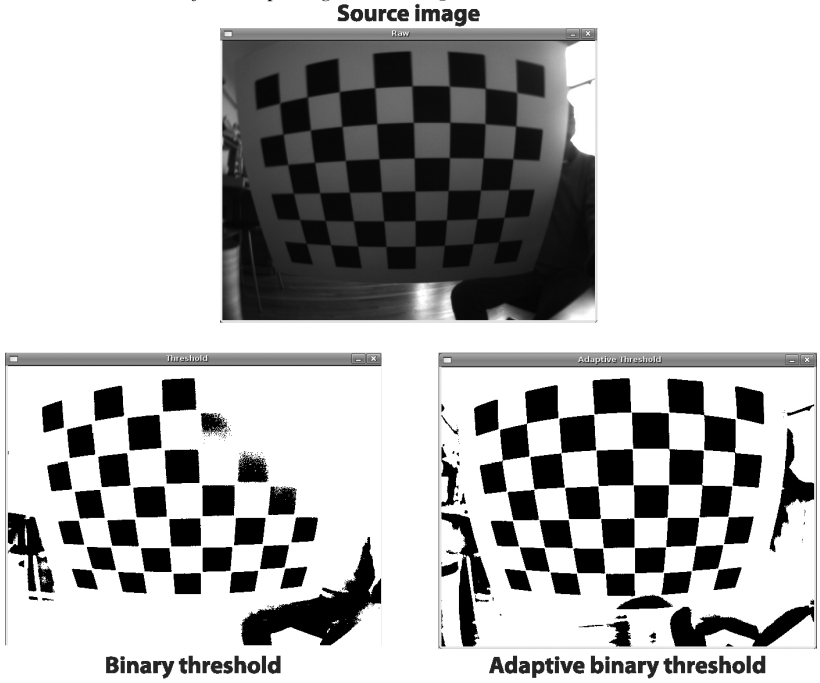


Figure 5-5 illustrates the result of processing an image that has a strong lighting gradient across it with both functions. The lower-left portion of the figure shows the result of using a single global threshold as in `cv::threshold()`; the lower-right portion shows the result of adaptive local threshold using `cv::adaptiveThreshold()`. We see that we get the whole checkerboard via adaptive threshold, a result that is impossible to achieve when using a single threshold. Note the calling-convention comments at the top of the code in Example 5-3; the parameters used for Figure 5-5 were:

```
./adaptThresh 15 1 1 71 15 ../Data/cal3-L.bmp
```

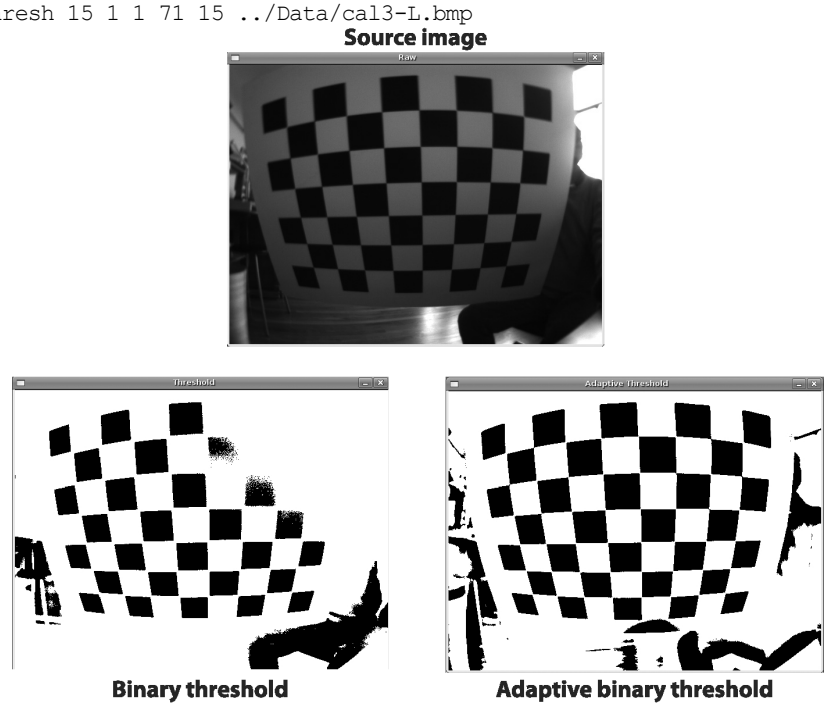


Figure 5-5: Binary threshold versus adaptive binary threshold: the input image (top) was turned into a Boolean image using a global threshold (lower-left) and an adaptive threshold (lower-right); raw image courtesy of Kurt Konolige

Example 5-3: Threshold versus adaptive threshold

```
#include <iostream>

using namespace std;

int main( int argc, char** argv )
{
  if(argc != 7) { cout <<
   "Usage: ch5_ex5_3 fixed_threshold invert(0=off|1=on) "
   "adaptive_type(0=mean|1=gaussian) block_size offset image\n"
   "Example: ch5_ex5_3 100 1 0 15 10 fruits.jpg\n"; return -1; }

  // Command line
  double fixed_threshold = (double)atof(argv[1]);
  int threshold_type  = atoi(argv[2]) ? cv::THRESH_BINARY : cv::THRESH_BINARY_INV;
  int adaptive_method = atoi(argv[3]) ? cv::ADAPTIVE_THRESH_MEAN_C
                                      : cv::ADAPTIVE_THRESH_GAUSSIAN_C;
  int block_size = atoi(argv[4]);
  double offset = (double)atof(argv[5]);
  cv::Mat Igray = cv::imread(argv[6], cv::LOAD_IMAGE_GRAYSCALE);

  // Read in gray image
```

```
  if( Igray.empty() ){ cout << "Can not load " << argv[6] << endl; return -1; }

  // Declare the output images
  cv::Mat It, Iat;

  // Thresholds
  cv::threshold(
    Igray,
    It,
    fixed_threshold,
    255,
    threshold_type);
  cv::adaptiveThreshold(
    Igray,
    Iat,
    255,
    adaptive_method,
    threshold_type,
    block_size,
    offset
  );

  // Show the results
  cv::imshow("Raw",Igray);
  cv::imshow("Threshold",It);
  cv::imshow("Adaptive Threshold",Iat);
  cv::waitKey(0);
  return 0;
}
```

Smoothing

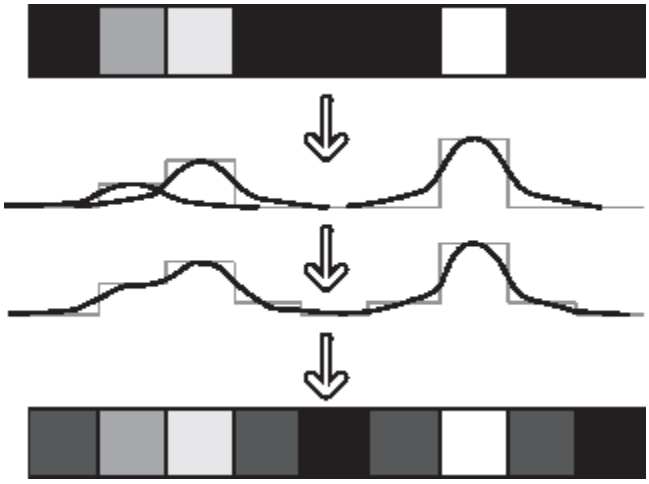


Figure 5-6: Gaussian blur on 1D pixel array

Smoothing, also called *blurring*, is a simple and frequently used image processing operation. There are many reasons for smoothing, but it is often done to reduce noise or camera artifacts. Smoothing is also important when we wish to reduce the resolution of an image in a principled way (we will discuss this in more detail in the "Image Pyramids" section of this chapter).

OpenCV offers five different smoothing operations, each with its own associated library function, which each accomplish slightly different kinds of smoothing.

The `src` and `dst` arguments in all of these functions are the usual source and destination arrays. After that, each smoothing operation has parameters that are specific to the associated operation. Of these, the only common parameter is the last, `borderType`. This argument tells the smoothing operation how to handle pixels at the edge of the image.

Simple Blur and the Box Filter

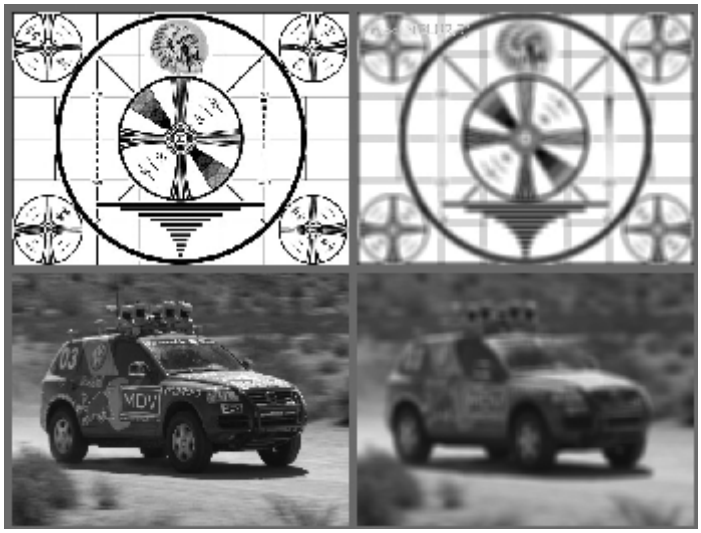


Figure 5-7: Image smoothing by block averaging: on the left are the input images; on the right, the output images

```
void cv::blur(
  cv::InputArray  src,                          // Input Image
  cv::OutputArray dst,                          // Result image
  cv::Size        ksize,                        // Kernel size
  cv::Point       anchor     = cv::Point(-1,-1),  // Location of anchor point
  int             borderType = cv::BORDER_DEFAULT // Border extrapolation to use
);
```

The *simple blur* operation is provided by `cv::blur()`. Each pixel in the output is the simple mean of all of the pixels in a window, usually called a *kernel*, around the corresponding pixel in the input. The size of this window is specified by the argument `ksize`. The argument `anchor` can be used to specify how the kernel is aligned with the pixel being computed. By default, the value of `anchor` is `cv::Point(-1,-1)`, which indicates that the kernel should be centered relative to the filter. In the case of multichannel images, each channel will be computed separately.

$$\frac{1}{25} \times \begin{bmatrix} 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & \underline{\mathbf{1}} & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \end{bmatrix}$$

Figure 5-8: A 5-by-5 blur filter, also called a normalized box filter.

The simple blur is a specialized version of the *box filter*. A box filter is any filter that has a rectangular profile and for which the values $k_{i,j}$ are all equal. In most cases, $k_{i,j} = 1$ for all i, j, or $k_{i,j} = {}^{1}/_{A}$, where A is the area of the filter. The latter case is called a *normalized box filter*.

```
void cv::boxFilter(
  cv::InputArray  src,                 // Input Image
  cv::OutputArray dst,                 // Result image
  int             ddepth,              // Pixel depth of output image (e.g., cv::U8)
  cv::Size        ksize,               // Kernel size
  cv::Point       anchor     = cv::Point(-1,-1),  // Location of anchor point
  bool            normalize  = true,              // If true, divide by box area
  int             borderType = cv::BORDER_DEFAULT // Border extrapolation to use
);
```

The OpenCV function `cv::boxFilter()` is the somewhat more general form of which `cv::blur()` is essentially a special case. The main difference between `cv::boxFilter()` and `cv::blur()` is that the former can be run in an un-normalized mode (`normalize = false`), and that the depth of the output image `dst` can be controlled. (In the case of `cv::blur()`, the depth of `dst` will always equal the depth of `src`.) If the value of `ddepth` is set to `-1`, then the destination image will have the same depth as the source; otherwise, you can use any of the usual aliases (e.g., `cv::F32`).

Median Filter

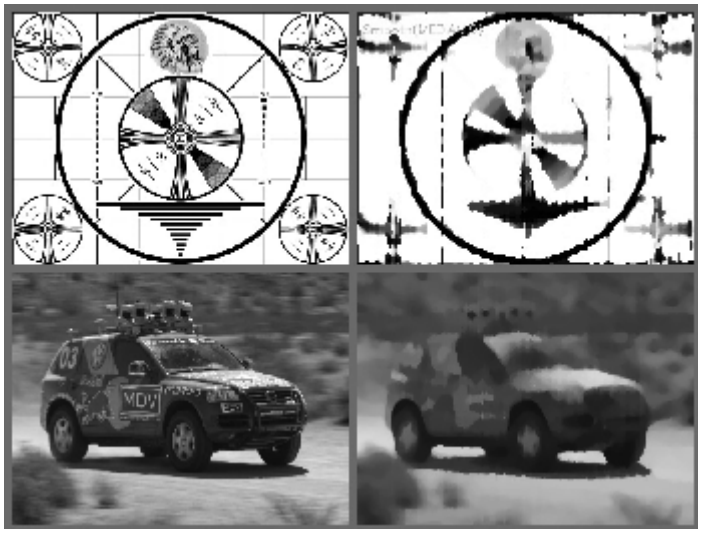


Figure 5-9: Image blurring by taking the median of surrounding pixels

The median filter *[Bardyn84] replaces each pixel by the median or "middle-valued" pixel (as opposed to the mean pixel) in a rectangular neighborhood around the center pixel.*[7] *Results of median filtering are shown in*

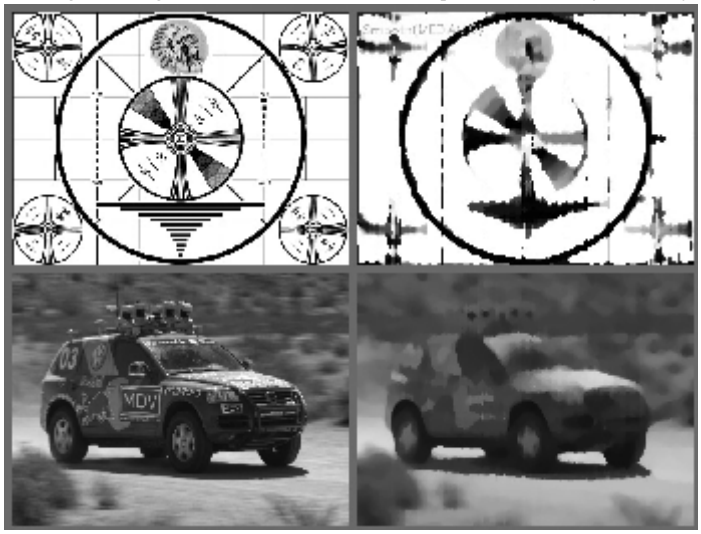


Figure 5-9. Simple blurring by averaging can be sensitive to noisy images, especially images with large isolated outlier values (sometimes called "shot noise"). Large differences in even a small number of points can cause a noticeable movement in the average value. Median filtering is able to ignore the outliers by selecting the middle points, though at a cost in speed.

```
void cv::medianBlur(
  cv::InputArray  src,                // Input Image
  cv::OutputArray dst,                // Result image
  cv::Size        ksize               // Kernel size
);
```

The arguments to `cv::medianBlur` are essentially the same as with previous filters, the source array `src`, the destination array `dst`, and the kernel size `ksize`. For `cv::medianBlur()`, the anchor point is always assumed to be at the center of the kernel.

[7] Note that the median filter is an example of a nonlinear kernel, which cannot be represented in the pictorial style shown in Figure 5-1.

Gaussian Filter

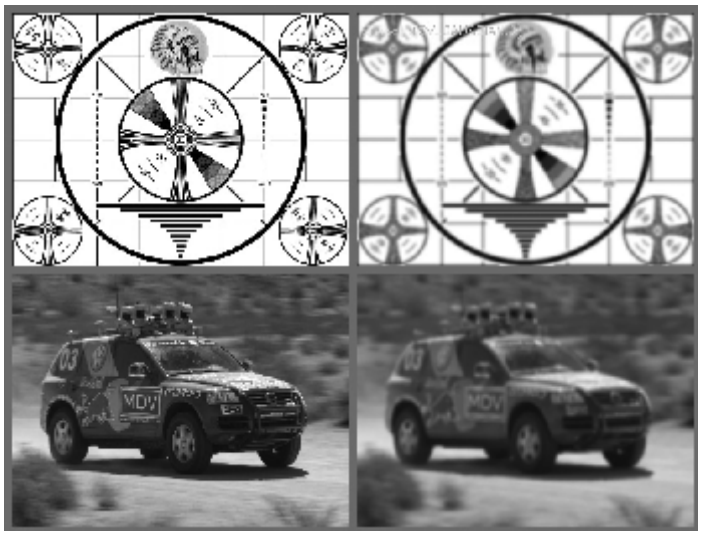


Figure 5-10: Gaussian filtering (blurring)

The next smoothing filter, the *Gaussian filter*, is probably the most useful. Gaussian filtering is done by convolving each point in the input array with a Gaussian kernel and then summing to produce the output array:

```
void cv::GaussianBlur(
  cv::InputArray  src,                 // Input Image
  cv::OutputArray dst,                 // Result image
  cv::Size        ksize,               // Kernel size
  double          sigmaX,              // Gaussian half-width in x-direction
  double          sigmaY      = 0.0,   // Gaussian half-width in y-direction
  int             borderType = cv::BORDER_DEFAULT // Border extrapolation to use
);
```

For the Gaussian blur (Figure 5-11), the parameter `ksize` gives the width and height of the filter window. The next parameter indicates the sigma value (half width at half max) of the Gaussian kernel in the x-dimension. The fourth parameter similarly indicates the sigma value in the y-dimension. If you specify only the x value, and set the y value to zero (its default value), then the y and x values will be taken to be equal. If you set them both to zero, then the Gaussian's parameters will be automatically determined from the window size using the following formulae:

$$\sigma_x = \left(\frac{n_x - 1}{2}\right) \cdot 0.30 + 0.80, \qquad n_x = ksize.width - 1,$$

$$\sigma_y = \left(\frac{n_y - 1}{2}\right) \cdot 0.30 + 0.80, \qquad n_y = ksize.height - 1.$$

Finally, `cv::GaussianBlur()` takes the usual `borderType` argument.

$$\frac{1}{141} \times \begin{bmatrix} 1 & 4 & 7 & 4 & 1 \\ 7 & 26 & \underline{\mathbf{41}} & 26 & 7 \\ 1 & 4 & 7 & 4 & 1 \end{bmatrix}$$

Figure 5-11: An example Gaussian kernel. Here `ksize=(5,3)`, `sigmaX=1`, and `sigmaY=0.5`

Gaussian smoothing is faster than one might expect. Because convolution by Gaussians is linearly combinable, we may separate the convolution along the x and the y axis converting the operation from

being n^2 in an n-by-n kernel size to being 2n for each pixel. The OpenCV implementation of Gaussian smoothing also provides even higher performance for several common kernels. 3-by-3, 5-by-5, and 7-by-7 with the "standard" sigma (i.e., `sigmaX = 0.0`) give better performance than other kernels. Gaussian blur supports single- or three-channel images in either 8-bit or 32-bit floating-point formats, and it can be done in place. Results of Gaussian blurring are shown in Figure 5-10.

Bilateral Filter

Figure 5-12: Results of bilateral smoothing

```
void cv::bilateralFilter(
  cv::InputArray  src,                // Input Image
  cv::OutputArray dst,                // Result image
  int             d,                  // Pixel neighborhood size (max distance)
  double          sigmaColor,         // Width parameter for color weighting function
  double          sigmaSpace,         // Width parameter for spatial weighting function
  int             borderType = cv::BORDER_DEFAULT // Border extrapolation to use
);
```

The fifth and final form of smoothing supported by OpenCV is called *bilateral filtering* [Tomasi98], an example of which is shown in Figure 5-12. Bilateral filtering is one operation from a somewhat larger class of image analysis operators known as *edge-preserving smoothing*. Bilateral filtering is most easily understood when contrasted to Gaussian smoothing. A typical motivation for Gaussian smoothing is that pixels in a real image should vary slowly over space and thus be correlated to their neighbors, whereas random noise can be expected to vary greatly from one pixel to the next (i.e., noise is not spatially correlated). It is in this sense that Gaussian smoothing reduces noise while preserving signal. Unfortunately, this method breaks down near edges, where you do expect pixels to be uncorrelated with their neighbors across the edge. As a result, Gaussian smoothing blurs away edges. At the cost of what is unfortunately substantially more processing time, bilateral filtering provides a means of smoothing an image without smoothing away the edges.

Like Gaussian smoothing, bilateral filtering constructs a weighted average of each pixel and its neighboring components. The weighting has two components, the first of which is the same weighting used by Gaussian smoothing. The second component is also a Gaussian weighting but is based not on the spatial distance from the center pixel but rather on the difference in intensity[8] from the center pixel.[9] When intensities are

[8] In the case of multichannel (i.e., color) images, the difference in intensity is replaced with a weighted sum over colors. This weighting is chosen to enforce a Euclidean distance in the CIE Lab color space.

close, the intensity weighting is nearly 1.0 and we get Gaussian smoothing, but when there are large differences in intensity, the weight is nearly zero and almost no averaging is done, keeping high-contrast edges sharp. The effect of this filter is typically to turn an image into what appears to be a watercolor painting of the same scene.[10] This can be useful as an aid to segmenting the image.

Bilateral filtering takes three parameters (other than the source and destination). The first is the diameter `d` of the pixel neighborhood that is considered when filtering. The second is the width of the Gaussian kernel used in the color domain called `sigmaColor`, which is analogous to the sigma parameters in the Gaussian filter. The third is the width of the Gaussian kernel in the spatial domain called `sigmaSpace`. The larger this second parameter, the broader the range of intensities (or colors) that will be included in the smoothing (and thus the more extreme a discontinuity must be in order to be preserved).

The filter size has a strong effect (as you might expect) on the speed of the algorithm. Typical values are less than or equal to five for video processing, but might be as high as nine for non-real-time applications. As an alternative to specifying `d` explicitly, it can be set to `-1`, in which case, it will be automatically computed from `sigmaSpace`.

> In practice, small values (e.g., `10`) give a very light, but noticeable effect, while large values (e.g., `150`) have a very strong effect and tend to render the image into a somewhat "cartoonish" appearance.

Derivatives and Gradients

One of the most basic and important convolutions is the computation of derivatives (or approximations to them). There are many ways to do this, but only a few are well suited to a given situation.

The Sobel Derivative

In general, the most common operator used to represent differentiation is the *Sobel derivative* [Sobel68] operator (see Figure 5-13 and Figure 5-14). Sobel operators exist for any order of derivative as well as for mixed partial derivatives (e.g., $\partial^2/\partial x \partial y$).

[9] Technically, the use of Gaussian distribution functions is not a necessary feature of bilateral filtering. The implementation in OpenCV uses Gaussian weighting even though the method allows many possible weighting functions.

[10] This effect is particularly pronounced after multiple iterations of bilateral filtering.

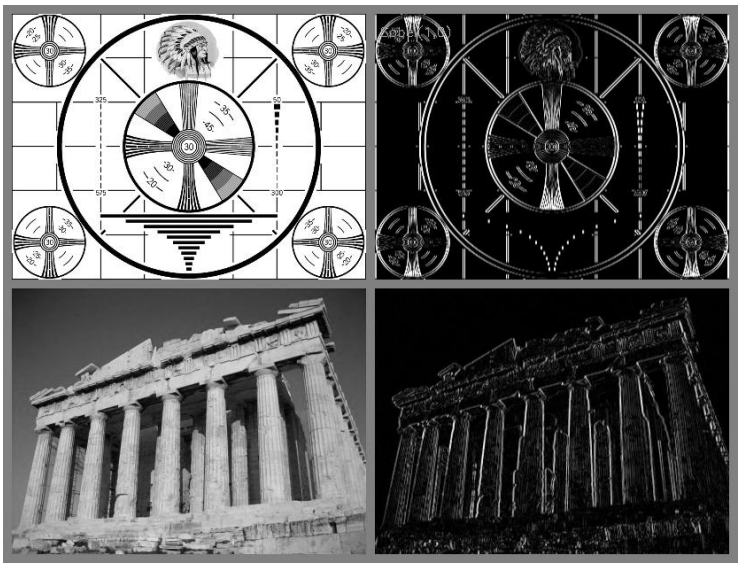


Figure 5-13: The effect of the Sobel operator when used to approximate a first derivative in the x-dimension.

```
void cv::Sobel(
  cv::InputArray  src,                  // Input Image
  cv::OutputArray dst,                  // Result image
  int             ddepth,               // Pixel depth of output image (e.g., cv::U8)
  int             xorder,               // order of corresponding derivative in x
  int             yorder,               // order of corresponding derivative in y
  cv::Size        ksize = 3,            // Kernel size
  double          scale = 1,            // Scale applied before assignment to dst
  double          delta = 0,            // Offset applied before assignment to dst
  int             borderType = cv::BORDER_DEFAULT // Border extrapolation to use
);
```

Here, `src` and `dst` are your image input and output. The argument `ddepth` allows you to select the depth (type) of the generated output (e.g., `cv::F32`). As a good example of how to use `ddepth`, if `src` is an 8-bit image, then the `dst` should have a depth of at least `cv::S16` to avoid overflow. `xorder` and `yorder` are the orders of the derivative. Typically, you'll use `0`, `1`, or at most `2`; a `0` value indicates no derivative in that direction.[11] The `ksize` parameter should be odd and is the width (and the height) of the filter used. Currently, aperture sizes up to `31` are supported.[12] The `scale` factor and `delta` are applied to the derivative before storing in `dst`. This can be useful when you want to actually visualize a derivative in an 8-bit image you can show on the screen:

$$dst_i = scale \cdot \left\{ \sum_{i,j \epsilon sobel_{kernel}} k_{i,j} * I(x+i, y+j) \right\} + delta.$$

The `borderType` argument functions exactly as described for other convolution operations.

Sobel computations are fast for the same reason Gaussian blurring is fast: the kernel can be separated and so, rather than a 2D x,y convolution over the area of the kernel, we reduce to a 1D x convolution combined with a 1D y convolution (see the exercises below).

[11] Either `xorder` or `yorder` must be nonzero.

[12] In practice, it really only makes sense to set the kernel size to `3` or greater. If you set `ksize` to `1`, then the kernel size will automatically be adjusted up to `3`.

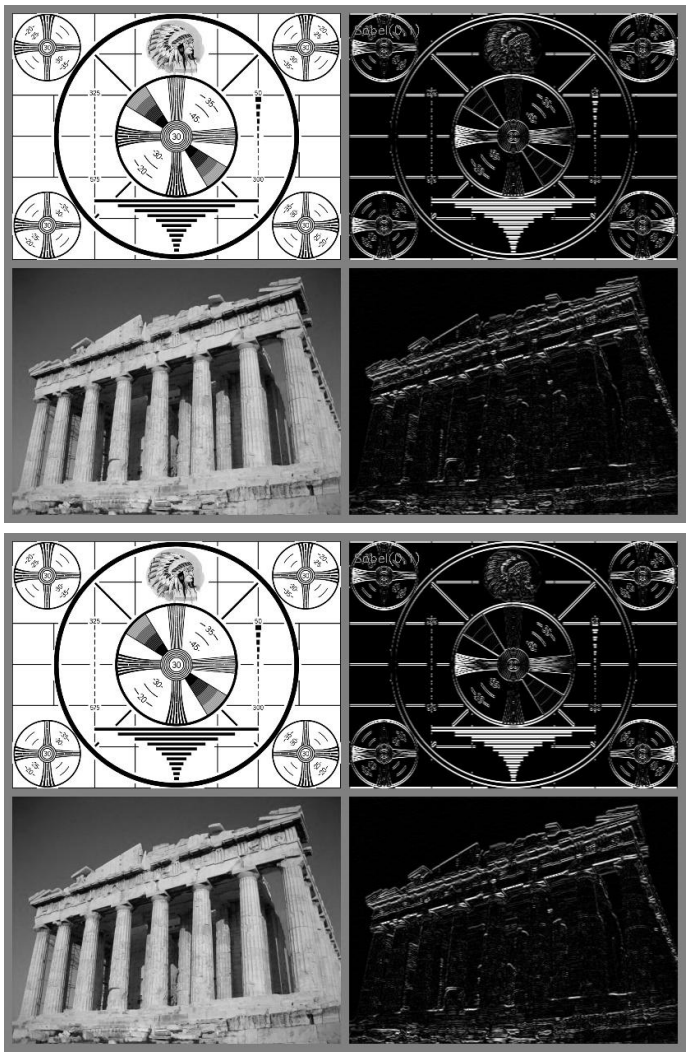


Figure 5-14: The effect of the Sobel operator when used to approximate a first derivative in the y-dimension.

Sobel derivatives have the nice property that they can be defined for kernels of any size, and those kernels can be constructed quickly and iteratively. Up to the limit of not spanning key image structures, larger kernels give a better approximation to the derivative because they span more area and are thus less sensitive to noise.

To understand this more exactly, we must realize that a Sobel derivative is not really a derivative as it is defined on a discrete space. What the Sobel operator actually represents is a fit to a polynomial. That is, the Sobel derivative of second order in the *x*-direction is not really a second derivative; it is a local fit to a parabolic function. This explains why one might want to use a larger kernel: that larger kernel is computing the fit over a larger number of pixels.

Scharr Filter

In fact, there are many ways to approximate a derivative in the case of a discrete grid. The downside of the approximation used for the Sobel operator is that it is less accurate for small kernels. For large kernels, where more points are used in the approximation, this problem is less significant. This inaccuracy does not show up directly for the x and y filters used in `cv::Sobel()`, because they are exactly aligned with the *x*- and *y*-axes. The difficulty arises when you want to make image measurements that are approximations of

directional derivatives (i.e., direction of the image gradient by using the arctangent of the y/x filter responses)[13].

To put this in context, a concrete example of where you may want image measurements of this kind would be in the process of collecting shape information from an object by assembling a histogram of gradient angles around the object. Such a histogram is the basis on which many common shape classifiers are trained and operated. In this case, inaccurate measures of gradient angle will decrease the recognition performance of the classifier because the data to be learned will vary depending on the rotation of the object.

For a 3-by-3 Sobel filter, the inaccuracies are more apparent the further the gradient angle is from horizontal or vertical. OpenCV addresses this inaccuracy for small (but fast) 3-by-3 Sobel derivative filters by a somewhat obscure use of the special `ksize` value `cv::SCHARR` in the `cv::Sobel()` function. The Scharr filter is just as fast but more accurate than the Sobel filter, so it should always be used if you want to make image measurements using a 3-by-3 filter. The filter coefficients for the Scharr filter are shown in Figure 5-15 [Scharr00].

-3	0	+3
-10	0	+10
-3	0	+3

A)

-3	-10	-3
0	0	0
+3	+10	+3

B)

Figure 5-15: The 3-by-3 Scharr filter using flag `cv::SCHARR`

The Laplacian

The OpenCV *Laplacian* function (first used in vision by Marr [Marr82]) implements a discrete approximation to the Laplacian operator[14]:

$$Laplace(f) = \frac{\partial^2 f}{\partial x^2} + \frac{\partial^2 f}{\partial y^2}$$

Because the Laplacian operator can be defined in terms of second derivatives, you might well suppose that the discrete implementation works something like the second-order Sobel derivative. Indeed it does, and in fact, the OpenCV implementation of the Laplacian operator uses the Sobel operators directly in its computation:

```
void cv::Laplacian(
  cv::InputArray  src,                  // Input Image
  cv::OutputArray dst,                  // Result image
  int             ddepth,               // Pixel depth of output image (e.g., cv::U8)
  cv::Size        ksize = 3,            // Kernel size
  double          scale = 1,            // Scale applied before assignment to dst
  double          delta = 0,            // Offset applied before assignment to dst
  int             borderType = cv::BORDER_DEFAULT // Border extrapolation to use
);
```

The `cv::Laplacian()` function takes the same arguments as the `cv::Sobel()` function, with the exception that the orders of the derivatives are not needed. This aperture `ksize` is precisely the same as the aperture appearing in the Sobel derivatives and, in effect, gives the size of the region over which the pixels are sampled in the computation of the second derivatives. In the actual implementation, for `ksize` anything other than `1`, the Laplacian is computed directly from the sum of the corresponding Sobel

[13] As you might recall, there are functions `cv::cartToPolar()` and `cv::polarToCart()` that implement exactly this transformation. If you find yourself wanting to call `cv::cartToPolar()` on a pair of x- and y-derivative images, you should probably be using `CV_SCHARR` to compute those images.

[14] Note that the *Laplacian operator* is distinct from the *Laplacian pyramid*, which we will discuss in Chapter 6.

operators. In the special case of `ksize=1`, the Laplacian is computed by convolution with the following single kernel:

0	1	0
1	-4	1
0	1	0

Figure 5-16. The single kernel used by `cv::Laplacian()` when `ksize=1`

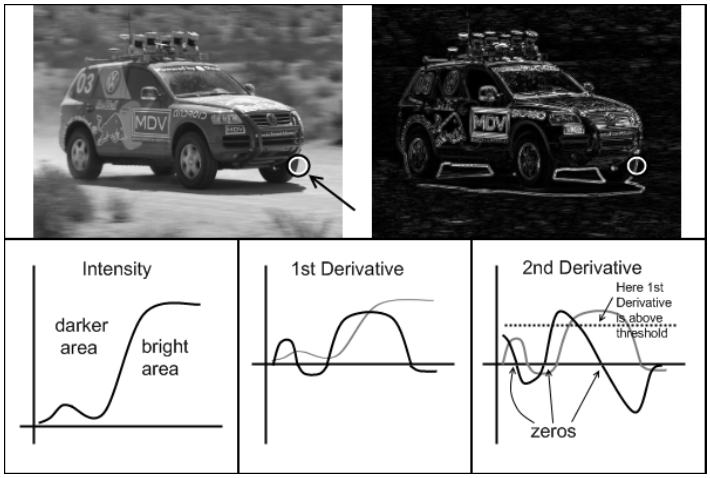


Figure 5-17: Laplace transform (upper right) of the racecar image: zooming in on the tire (circled in white) and considering only the x-dimension, we show a (qualitative) representation of the brightness as well as the first and second derivatives (lower three cells); the 0s in the second derivative correspond to edges, and the 0 corresponding to a large first derivative is a strong edge

The Laplace operator can be used in a variety of contexts. A common application is to detect "blobs." Recall that the form of the Laplacian operator is a sum of second derivatives along the *x*-axis and *y*-axis. This means that a single point or any small blob (smaller than the aperture) that is surrounded by higher values will tend to maximize this function. Conversely, a point or small blob that is surrounded by lower values will tend to maximize the negative of this function.

With this in mind, the Laplace operator can also be used as a kind of edge detector. To see how this is done, consider the first derivative of a function, which will (of course) be large wherever the function is changing rapidly. Equally important, it will grow rapidly as we approach an edge-like discontinuity and shrink rapidly as we move past the discontinuity. Hence, the derivative will be at a local maximum somewhere within this range. Therefore, we can look to the 0s of the second derivative for locations of such local maxima. Got that? Edges in the original image will be 0s of the Laplacian. Unfortunately, both substantial and less meaningful edges will be 0s of the Laplacian, but this is not a problem because we can simply filter out those pixels that also have larger values of the first (Sobel) derivative. Figure 5-17 shows an example of using a Laplacian on an image together with details of the first and second derivatives and their zero crossings.

Image Morphology

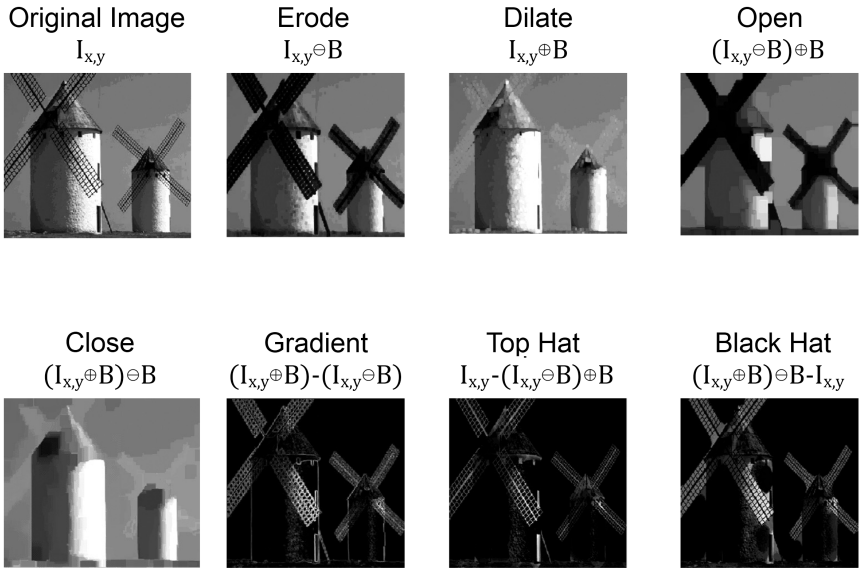


Figure 5-18: Summary results for all morphology operators

OpenCV provides a fast, convenient interface for doing *morphological transformations* [Serra83] on an image. Image morphology is its own topic, and over the years, especially in the early days of computer vision, a great number of morphological operations were developed. Most were developed for one specific purpose or another, and some of those found broader utility over the years. Essentially, all morphology operations are based on just two primitive operations. We will start with those, and then move on to the more complex operations, each of which is typically defined in terms of its simpler predecessors.

Dilation and Erosion

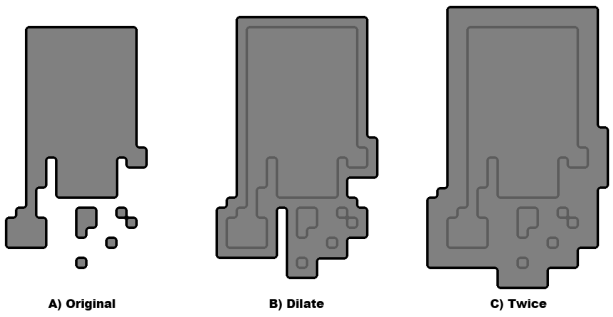


Figure 5-19: Morphological dilation: take the maximum under a square kernel

The basic morphological transformations are called *dilation* and *erosion*, and they arise in a wide variety of contexts such as removing noise, isolating individual elements, and joining disparate elements in an image. More sophisticated morphology operations, based on these two basic operations, can also be used to find intensity peaks (or holes) in an image and to define a particular form of image gradient.

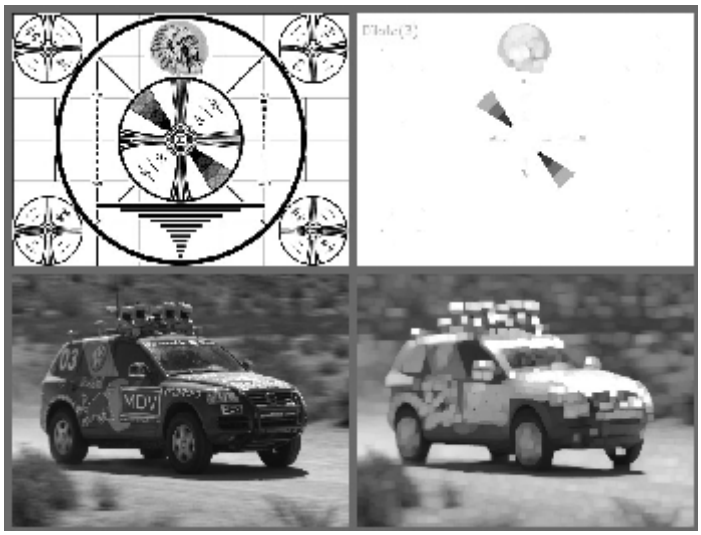


Figure 5-20: Results of the dilation, or "max," operator: bright regions are expanded and often joined

Dilation is a convolution of some image with a kernel in which any given pixel is replaced with the local maximum *of all of the pixel values covered by the kernel. As we mentioned earlier, this is an example of a nonlinear operation, so the kernel cannot be expressed in the form shown in Figure 5-1. Most often, the kernel used for Dilation is a "solid" square kernel, or sometimes a disk, with the anchor point at the center. The effect of dilation is to cause filled*[15] *regions within an image to grow as diagrammed in*

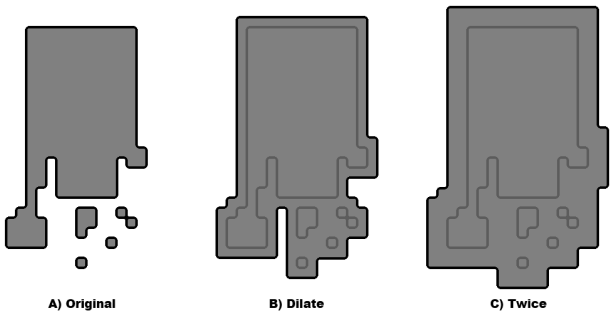


Figure 5-19.

[15] Here the term "filled" actually means those pixels whose value is nonzero. You could read this as "bright," since the local maximum actually takes the pixel with the highest intensity value under the template (kernel). It is worth mentioning that the diagrams that appear in this chapter to illustrate morphological operators are in this sense inverted relative to what would happen on your screen (because books write with dark ink on light paper instead of light pixels on a dark screen).

Erosion is the converse operation. The action of the erosion operator is equivalent to computing a local minimum *over the area of the kernel.*[16] *Erosion is diagrammed in*

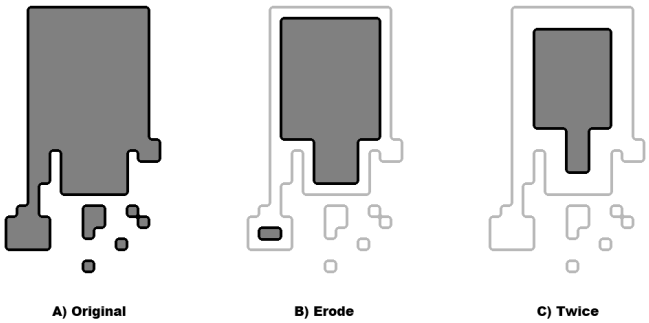


Figure 5-21.

Image morphology is often done on Boolean[17] images that result from a threshold operation. However, because dilation is just a max operator and erosion is just a min operator, morphology may be used on intensity images as well.

In general, whereas dilation expands a bright region A, erosion (

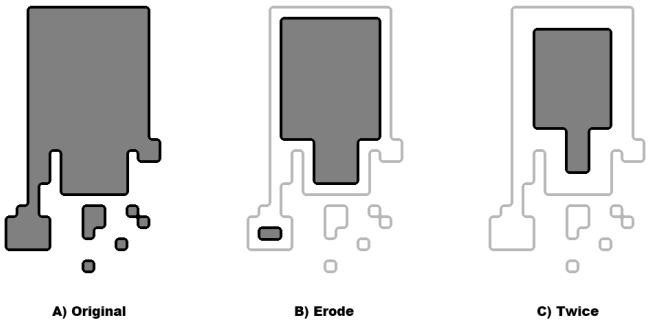


Figure 5-21) reduces such a bright region. Moreover, dilation will tend to fill concavities and erosion will tend to remove protrusions. Of course, the exact result will depend on the kernel, but these statements are generally true so long as the kernel is both convex and filled.

In OpenCV, we effect these transformations using the `cv::erode()` and `cv::dilate()` functions:

```
void cv::erode(
  cv::InputArray    src,              // Input Image
  cv::OutputArray   dst,              // Result image
  cv::InputArray    element,          // Structuring element, can be cv::Mat()
  cv::Point         anchor     = cv::Point(-1,-1),  // Location of anchor point
  int               iterations = 1,                 // Number of times to apply
  int               borderType = cv::BORDER_DEFAULT // Border extrapolation to use
  const cv::Scalar& borderValue = cv::morphologyDefaultBorderValue()
```

[16] To be precise, the pixel in the destination image is set to the value equal to the minimal value of the pixels under the kernel in the source image.

[17] It should be noted that OpenCV does not actually have a Boolean image data type. The minimum size representation is 8-bit characters. Those functions which *interpret* an image as Boolean do so by classifying all pixels as either zero ("False" or 0) or nonzero ("True" or 1).

```
);
```

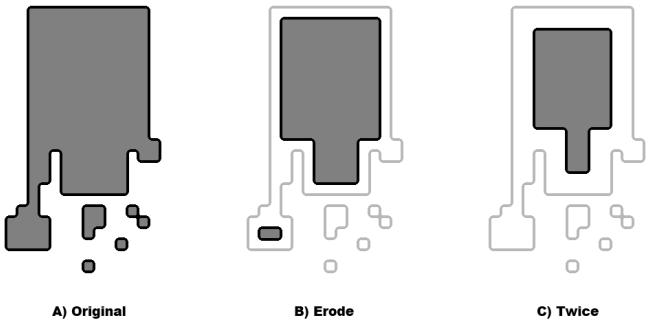


Figure 5-21: Morphological erosion: take the minimum under a square kernel

```
void cv::dilate(
  cv::InputArray    src,               // Input Image
  cv::OutputArray   dst,               // Result image
  cv::InputArray    element,           // Structuring element, can be cv::Mat()
  cv::Point         anchor     = cv::Point(-1,-1),  // Location of anchor point
  int               iterations = 1,                 // Number of times to apply
  int               borderType = cv::BORDER_CONSTANT // Border extrapolation to use
  const cv::Scalar& borderValue = cv::morphologyDefaultBorderValue()
);
```

Both `cv::erode()` and `cv::dilate()` take a source and destination image, and both support "in place" calls (in which the source and destination are the same image). The third argument is the kernel, to which you may pass an uninitialized array `cv::Mat()`, which will cause it to default to using a 3-by-3 kernel with the anchor at its center (we will discuss how to create your own kernels later). The fourth argument is the number of iterations. If not set to the default value of 1, the operation will be applied multiple times during the single call to the function. The `borderType` argument is the usual border type, and the `borderValue` is the value that will be used for off-the-edge pixels when the `borderType` is set to `cv::BORDER_CONSTANT`.

The results of an erode operation on a sample image are shown in

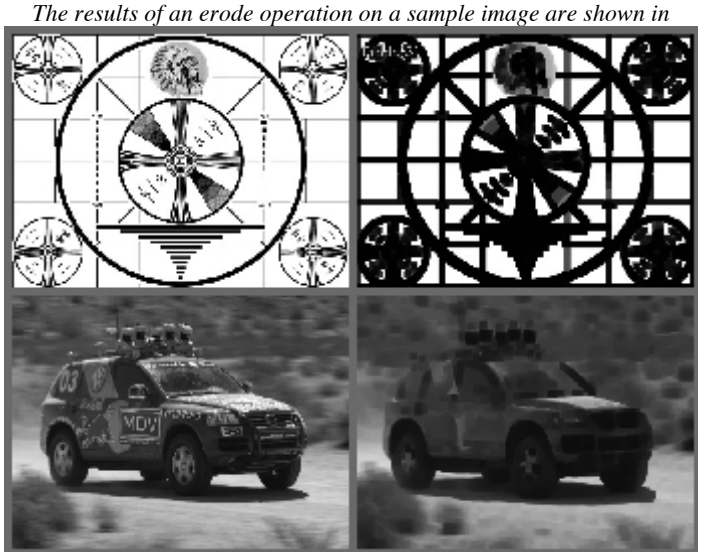


Figure 5-22 and those of a dilation operation on the same image are shown in

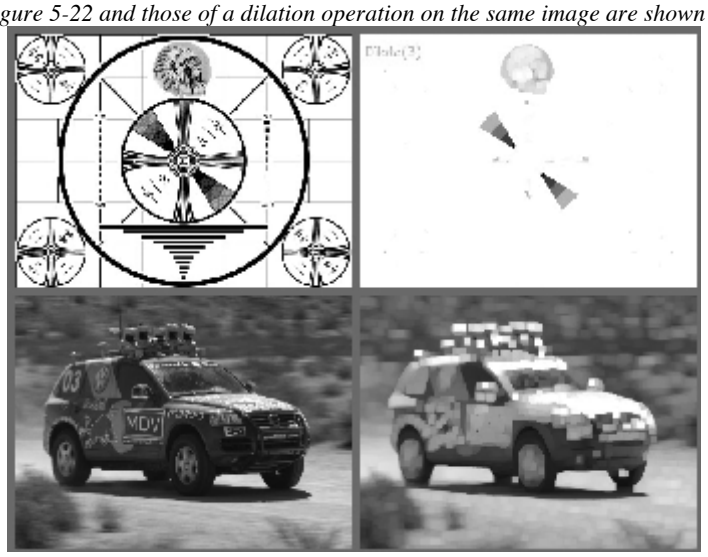


Figure 5-20. The erode operation is often used to eliminate "speckle" noise in an image. The idea here is that the speckles are eroded to nothing while larger regions that contain visually significant content are not affected. The dilate operation is often used when attempting to find *connected components* (i.e., large discrete regions of similar pixel color or intensity). The utility of dilation arises because in many cases a large region might otherwise be broken apart into multiple components as a result of noise, shadows, or some other similar effect. A small dilation will cause such components to "melt" together into one.

To recap: when OpenCV processes the `cv::erode()` function, what happens beneath the hood is that the value of some point p is set to the minimum value of all of the points covered by the kernel when aligned at p; for the dilation operator, the equation is the same except that max is considered rather than min:

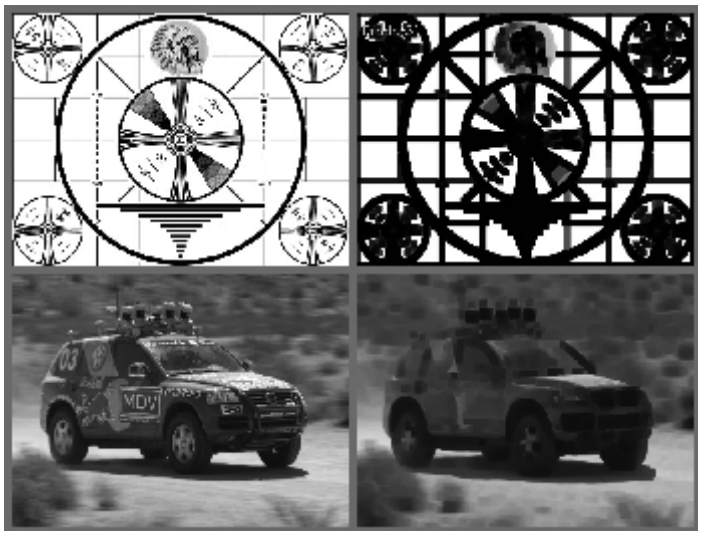


Figure 5-22: Results of the erosion, or "min," operator: bright regions are isolated and shrunk

$$erode(x,y) = \min_{(i,j)\in kernel} src(x+i, y+j)$$

$$dilate(x,y) = \max_{(i,j)\in kernel} src(x+i, y+j)$$

You might be wondering why we need a complicated formula when the earlier heuristic description was perfectly sufficient. Some users actually prefer such formulas but, more importantly, the formulas capture

some generality that isn't apparent in the qualitative description. Observe that if the image is not Boolean, then the min and max operators play a less trivial role. Take another look at

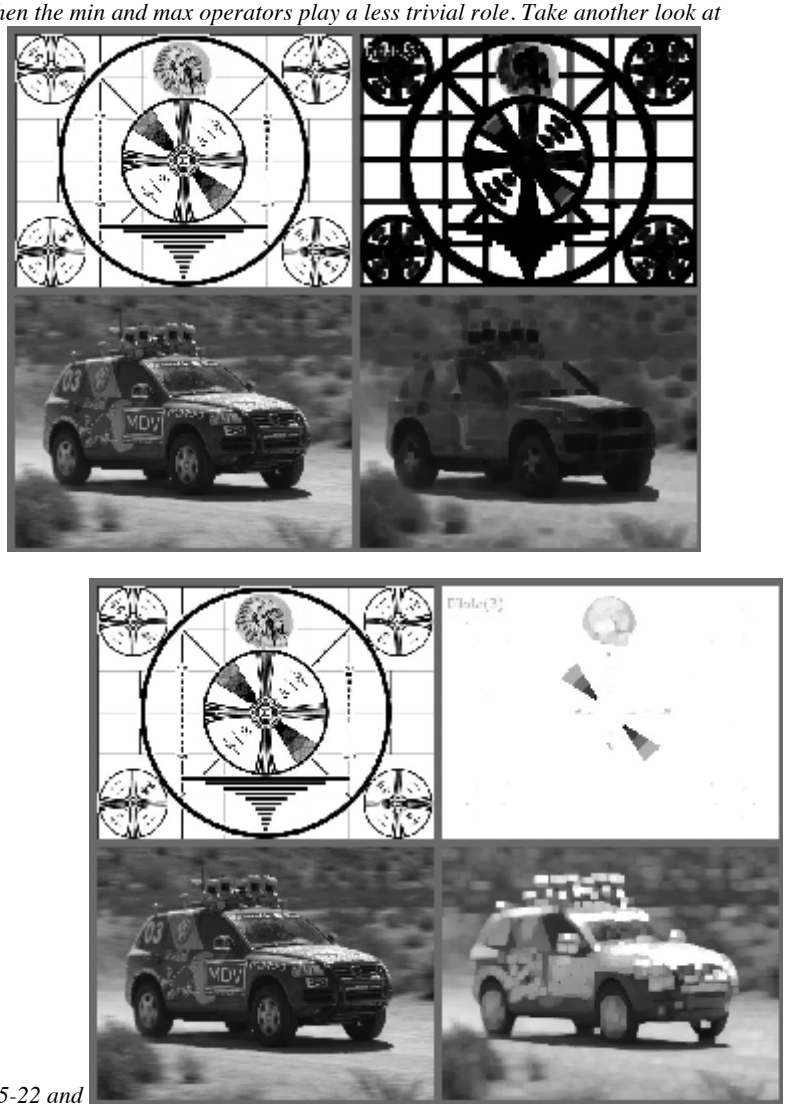


Figure 5-22 and Figure 5-20, which show the erosion and dilation operators applied to two real images.

The General Morphology Function

When working with Boolean images and image masks, the basic erode and dilate operations are usually sufficient. When working with grayscale or color images, however, a number of additional operations are often helpful. Several of the more useful operations can be handled by the multipurpose `cv::morphologyEx()` function.

```
void cv::morphologyEx(
  cv::InputArray    src,              // Input Image
  cv::OutputArray   dst,              // Result image
  int               op,               // Morphology operator to use e.g. cv::MOP_OPEN
  cv::InputArray    element,          // Structuring element, can be cv::Mat()
  cv::Point         anchor     = cv::Point(-1,-1),  // Location of anchor point
  int               iterations = 1,                 // Number of times to apply
  int               borderType = cv::BORDER_DEFAULT // Border extrapolation to use
  const cv::Scalar& borderValue = cv::morphologyDefaultBorderValue()
```

```
);
```

In addition to the arguments that we saw with the `cv::dilate()` and `cv::erode()` functions, `cv::morphologyEx()` has one new—and very important—parameter. This new argument, called `op`, is the specific operation to be done; the possible values of this argument are listed on Table 5-3.

Table 5-3: `cv::morphologyEx()` operation options

Value of `operation`	Morphological operator	Requires temp image?
`cv::MOP_OPEN`	Opening	No
`cv::MOP_CLOSE`	Closing	No
`cv::MOP_GRADIENT`	Morphological gradient	Always
`cv::MOP_TOPHAT`	Top Hat	For in-place only (`src = dst`)
`cv::MOP_BLACKHAT`	Black Hat	For in-place only (`src = dst`)

Opening and closing

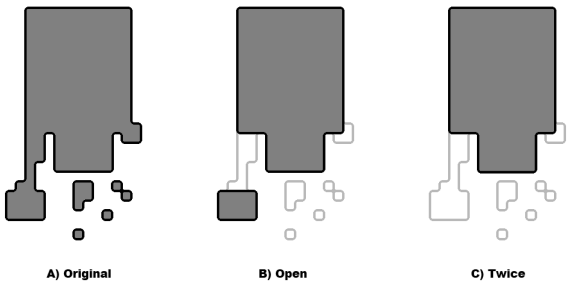


Figure 5-23: Morphological opening applied to a simple Boolean image.

The first two operations, *opening* and *closing*, are actually simple combinations of the erosion and dilation operators. In the case of opening, we erode first and then dilate (Figure 5-23). Opening is often used to count regions in a Boolean image. For example, if we have thresholded an image of cells on a microscope slide, we might use opening to separate out cells that are near each other before counting the regions.

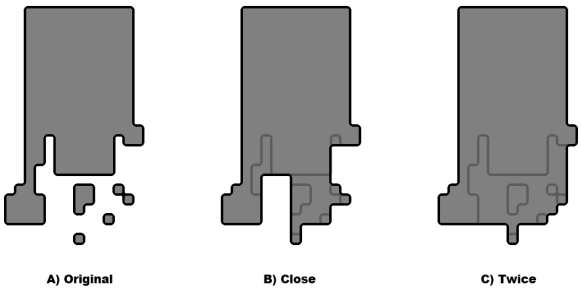


Figure 5-24: Morphological closing applied to a simple Boolean image.

In the case of closing, we dilate first and then erode (Figure 5-24). Closing is used in most of the more sophisticated connected-component algorithms to reduce unwanted or noise-driven segments. For connected components, usually an erosion or closing operation is performed first to eliminate elements that

arise purely from noise and then an opening operation is used to connect nearby large regions. (Notice that, although the end result of using open or close is similar to using erode or dilate, these new operations tend to preserve the area of connected regions more accurately.)

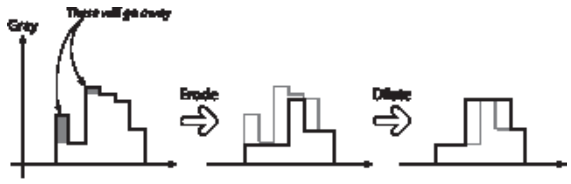


Figure 5-25: Morphological opening operation applied to a (one-dimensional) non-Boolean image: the upward outliers are eliminated as a result

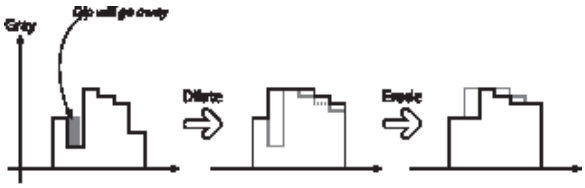


Figure 5-26: Morphological closing operation applied to a (one-dimensional) non-Boolean image: the downward outliers are eliminated as a result

When used on non-Boolean images, the most prominent effect of closing is to eliminate lone outliers that are are lower in value than their neighbors, whereas the effect of opening is to eliminate lone outliers that are higher than their neighbors. Results of using the opening operator are shown in

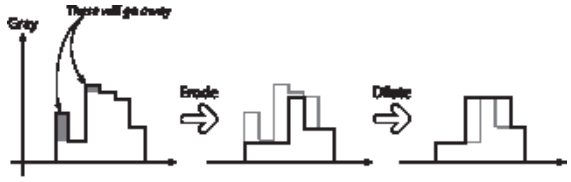


Figure 5-25, and of the closing operator in

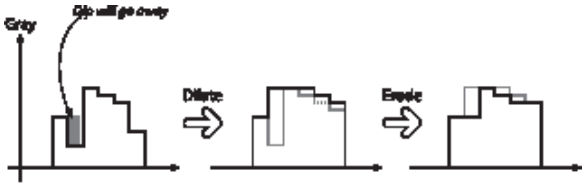


Figure 5-26.

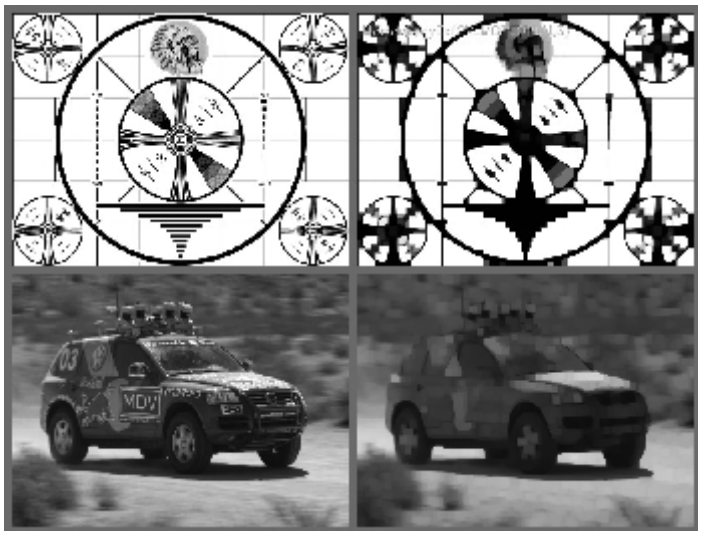


Figure 5-27: Results of morphological opening on an image: small bright regions are removed, and the remaining bright regions are isolated but retain their size

One last note on the opening and closing operators concerns how the `iterations` *argument is interpreted. You might expect that asking for two iterations of closing would yield something like dilate-erode-dilate-erode. It turns out that this would not be particularly useful. What you usually want (and what you get) is dilate-dilate-erode-erode. In this way, not only the single outliers but also neighboring pairs of outliers will disappear. Figure 5-23 (c) and*

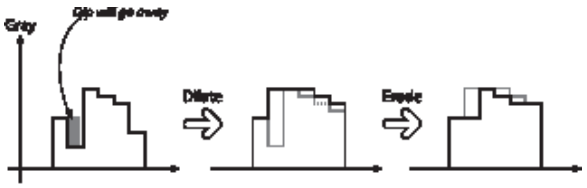


Figure 5-26 (c) illustrate the effect of calling open and close (respectively) with an iteration count of two.

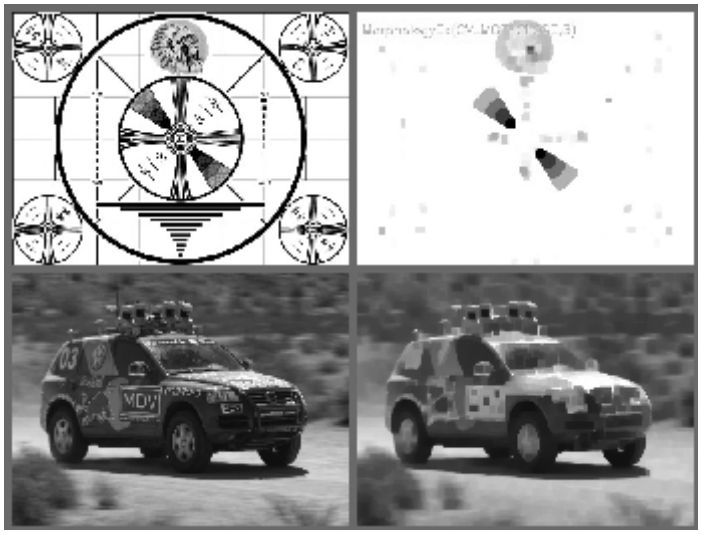


Figure 5-28: Results of morphological closing on an image: bright regions are joined but retain their basic size

Morphological gradient

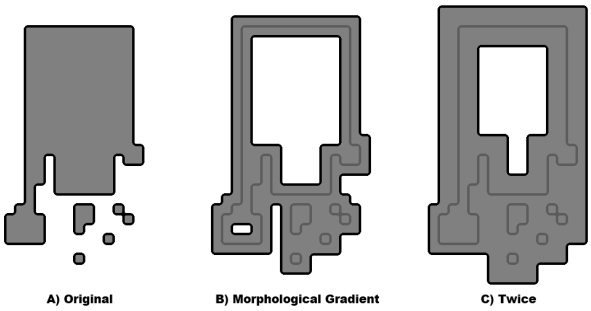


Figure 5-29: Morphological gradient applied to a simple Boolean image.

Our next available operator is the *morphological gradient*. For this one, it is probably easier to start with a formula and then figure out what it means:

$$gradient(src) = dilate(src) - erode(src).$$

As we can see in Figure 5-29, the effect of subtracting the eroded (slightly reduced) image from the dilated (slightly enlarged) image is to leave behind a representation of the edges of objects in the original image.

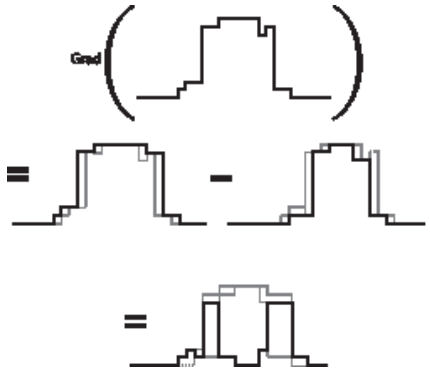
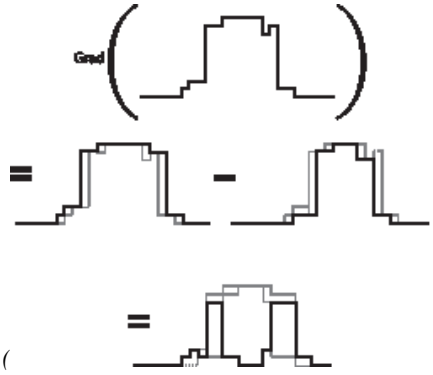


Figure 5-30: Morphological gradient applied to (one-dimensional) non-Boolean image: as expected, the operator has its highest values where the grayscale image is changing most rapidly.



With a grayscale image (

Figure 5-30), we see that the value of the operator is telling us something about how fast the image brightness is changing; this is why the name "morphological gradient" is justified. Morphological gradient is often used when we want to isolate the perimeters of bright regions so we can treat them as whole objects (or as whole parts of objects). The complete perimeter of a region tends to be found because a contracted

version is subtracted from an expanded version of the region, leaving a complete perimeter edge. This differs from calculating a gradient, which is much less likely to work around the full perimeter of an object.

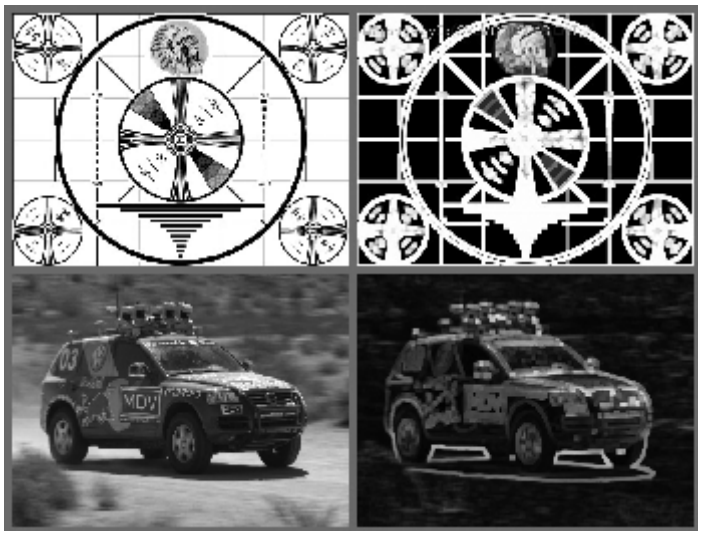


Figure 5-31: Results of the morphological gradient operator: bright perimeter edges are identified

Top Hat and Black Hat

Figure 5-32: The Top Hat (left) and Black Hat (right) morphology operators

The last two operators are called *Top Hat* and *Black Hat [Meyer78]*. These operators are used to isolate patches that are, respectively, brighter or dimmer than their immediate neighbors. You would use these when trying to isolate parts of an object that exhibit brightness changes relative only to the object to which they are attached. This often occurs with microscope images of organisms or cells, for example. Both operations are defined in terms of the more primitive operators, as follows:

$$TopHat(src) = src - open(src),$$

$$BlackHat(src) = close(src) - src.$$

Figure 5-33: Results of morphological Top Hat operation: bright local peaks are isolated

As you can see, the Top Hat operator subtracts the opened form of A from A. Recall that the effect of the open operation was to exaggerate small cracks or local drops. Thus, subtracting $open(A)$ from A should reveal areas that are lighter than the surrounding region of A, relative to the size of the kernel (see Figure 5-33); conversely, the Black Hat operator reveals areas that are darker than the surrounding region of A (Figure 5-34). Summary results for all the morphological operators discussed in this chapter are assembled in Figure 5-18.[18]

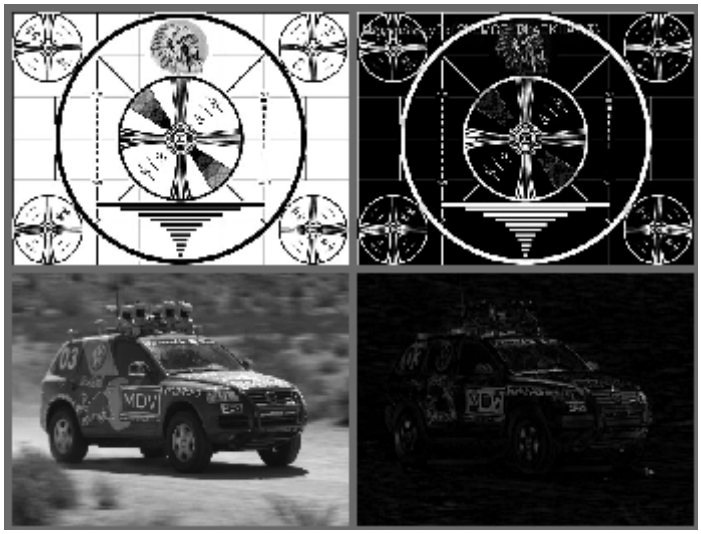


Figure 5-34: Results of morphological Black Hat operation: dark holes are isolated

Making Your Own Kernel

In the morphological operations we have looked at so far, the kernels considered were always square and 3-*by*-3. If you need something a little more general than that, OpenCV allows you to create your own kernel.

[18] Both of these operations (Top Hat and Black Hat) make more sense in grayscale morphology, where the structuring element is a matrix of real numbers (not just a Boolean mask) and the matrix is added to the current pixel neighborhood before taking a minimum or maximum. Unfortunately, this is not yet implemented in OpenCV.

In the case of morphology, the kernel is often called a *structuring element*, so the routine that allows you to create your own morphology kernels is called `cv::getStructuringElement()`.

In fact, you can just create any array you like and use it as a structuring element in functions like `cv::dilate()`, `cv::erode()`, or `cv::morphologyEx()`, but this is often more work than is necessary. Often what you need is a nonsquare kernel of an otherwise common shape. This is what `cv::getStructuringElement()` is for:

```
cv::Mat cv::getStructuringElement(
  int       shape,                    // Element shape, e.g. cv::MORPH_RECT
  cv::Size  ksize,                    // Size of structuring element (should be odd)
  cv::Point anchor = cv::Point(-1,-1)  // Location of anchor point
);
```

The first argument `shape` controls which basic shape will be used to create the element (Table 5-4), while `ksize` and `anchor` specify the size of the element and the location of the anchor point, respectively. As usual, if the `anchor` argument is left with its default value of `cv::Point(-1,-1)`, then `cv::getStructuringElement()` will take this to mean that the anchor should automatically be placed at the center of the element.

Table 5-4 `cv::getStructuringElement()` element shapes

Value of `shape`	Element	Description
`cv::MORPH_RECT`	Rectangular	$E_{i,j} = 1, \forall i, j$
`cv::MORPH_ELLIPSE`	Elliptic	Ellipse with axes `ksize.width` and `ksize.height`.
`cv::MORPH_CROSS`	Cross-shaped	$E_{i,j} = 1$, iff $i == anchor.y$ or $j == anchor.x$

Of the options for shape shown in Table 5-4, the last is there only for legacy compatibility. In the old C API (v1.x), there was a separate structure used for the purpose of expressing convolution kernels. There is no need to use this functionality now, as you can simply pass any `cv::Mat` to the morphological operators as a structuring element if you need something more complicated than the basic shape-based elements created by `cv::getStructuringElement()`.

Convolution with an Arbitrary Linear Filter

In the functions we have seen so far, the basic mechanics of the convolution were happening deep down below the level of the OpenCV API. We took some time to understand the basics of convolution, then we went on to look at a long list of functions that implemented different kinds of useful convolutions. In essentially every case, there was a kernel that was implied by the function we chose, and we just passed that function a little extra information that parameterized that particular filter type. For linear filters, however, it is possible to just provide the entire kernel and let OpenCV handle the convolution for us.

From an abstract point of view, this is very straightforward: we just need a function that takes an array argument to describe the kernel and we are done. At a practical level, there is an important subtlety that strongly affects performance. That subtlety is that some kernels are *separable*, and others are not.

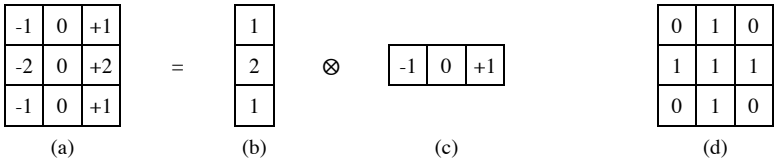


Figure 5-35: The Sobel kernel (a) is separable; it can be expressed as two one-dimensional convolutions (b and c); (d) is an example of a non-separable kernel

A separable kernel is one that can be thought of as two one-dimensional kernels, the application of which is done by first convolving with the x-kernel and then with the y-kernel. The benefit of this decomposition is that the computational cost of a kernel convolution is approximately the image area multiplied by the kernel area[19]. This means that convolving your image of area A by an n-by-n kernel takes time proportional to An^2, while convolving your image once by an n-by-1 kernel and then by a 1-by-n kernel takes time proportional to $An + An = 2An$. For even n as small as 3 there is a benefit, and the benefit grows with n.

Applying a general filter with `cv::filter2D()`

Given that the number of operations required for an image convolution, at least at first glance[20], seems to be the number of pixels in the image multiplied by the number of pixels in the kernel, this can be a lot of computation and so is not something you want to do with some `for` loop and a lot of pointer de-referencing. In situations like this, it is better to let OpenCV do the work for you and take advantage of the internal optimizations. The OpenCV way to do this is with `cv::filter2D()`:

```
cv::filter2D(
  cv::InputArray  src,                 // Input Image
  cv::OutputArray dst,                 // Result image
  int             ddepth,              // Pixel depth of output image (e.g., cv::U8)
  cv::InputArray  kernel,              // Your own kernel
  cv::Point       anchor     = cv::Point(-1,-1),  // Location of anchor point
  double          delta = 0,                      // Offset before assignment to dst
  int             borderType = cv::BORDER_DEFAULT // Border extrapolation to use
);
```

Here we create an array of the appropriate size, fill it with the coefficients of our linear filter, and then pass it together with the source and destination images into `cv::filter2D()`. As usual, we can specify the depth of the resulting image with `ddepth`, the anchor point for the filter with `anchor`, and the border extrapolation method with `borderType`. The kernel can be of even size if its anchor point is defined; otherwise, it should be of odd size. If you want an overall offset applied to the result after application of the linear filter, you can use the argument `delta`.

Applying a general separable filter with cv::sepFilter2D

In the case where your kernel is separable, you will get the best performance from OpenCV by expressing it in its separated form and passing those one-dimensional kernels to OpenCV (e.g., passing the kernels shown in Figure 5-35b and Figure 5-35c instead of the one shown in Figure 5-35a.) The OpenCV function `cv::sepFilter2D()` is like `cv::filter2D()`, except that it expects these two one-dimensional kernels instead of one two-dimensional kernel.

```
cv::sepFilter2D(
  cv::InputArray  src,                 // Input Image
  cv::OutputArray dst,                 // Result image
  int             ddepth,              // Pixel depth of output image (e.g., cv::U8)
  cv::InputArray  rowKernel,           // 1-by-N row Kernel
  cv::InputArray  columnKernel,        // M-by-1 column kernel
  cv::Point       anchor     = cv::Point(-1,-1),  // Location of anchor point
```

[19] This statement is only exactly true for convolution in the spatial domain, which is how OpenCV handles only small kernels.

[20] We say "at first glance" because it is also possible to perform convolutions in the frequency domain. In this case, for an n-by-n image and an m-by-m kernel with $n \gg m$, the computational time will be proportional to $n^2 \log(n)$ and not to the n^2m^2 that is expected for computations in the spatial domain. Because the frequency domain computation is independent of the size of the kernel, it is more efficient for large kernels. OpenCV automatically decides whether to do the convolution in the frequency domain based on the size of the kernel.

```
  double          delta      = 0,                 // Offset before assignment to dst
  int             borderType = cv::BORDER_DEFAULT // Border extrapolation to use
);
```

All the arguments of `cv::sepFilter2D()` are the same as those of `cv::filter2D()`, with the exception of the replacement of the `kernel` argument with the `rowKernel` and `columnKernel` arguments. The latter two are expected to be n_1-by-1 and 1-by-n_2 arrays (with n_1 not necessarily equal to n_2).

The Filter Architecture

Almost all of the functions described in this chapter are based on a unified *Filter Architecture*. The Filter Architecture is a class system that allows you (or the OpenCV developers) to create a new filter and automatically reuse all of the things that every filter has in common (the convolution steps, handling of the borders, etc.)

The Big Picture

The process of doing convolutions is complex because these are always very computationally intensive operations. As a result, there is a great benefit to doing things very efficiently. On the other hand, writing very efficient code is often very difficult to do in a modular way. As a result, OpenCV has an entire architecture for handling just this problem. This architecture begins at the top with high-level operations like `cv::blur()`. This function takes your image, smoothes it, and gives it back. Sounds simple, but this is actually the result of many different layers operating together. When you want to do your own thing, you will have to figure out exactly which level you want to tinker with. Tinker at too high a level, and you won't have the control you need. Tinker at too low a level, and you will find yourself reinventing every sort of wheel before you are done. To get the most out of OpenCV's filter architecture, it is important to keep the big picture in mind.

The big picture is best understood by working through an example. The "top" layer of the filter architecture is a collection of high-level service routines, like `cv::blur()` mentioned above. Inside of `cv::blur()`, however, your image is given to an object called a *filter engine*. The filter engine is a class that handles all of the general aspects of applying a filter kernel to your image, like iterating over the image, operating on each segment of the image, and handling boundary conditions. As the filter engine moves through the image, it will pass chunks of the image down to the next level that contains the actual filters. These filters, called *base filters* in OpenCV, do the actual work of computing individual destination image pixels from small windows of the source image and the kernel. At the very bottom of this hierarchy are the kernel builders. These are routines that actually compute individual kernels from some specific set of parameters (i.e., the variance of the Gaussian used in the `cv::blur()` code).

Filter Engines

You can use the Filter Engine layer to implement your own basic filters. The filters you implement need not be linear filters as was the case with `cv::filter2D()`. For performance reasons, you may also want to create composite filters, which—though they may accomplish something similar or identical to some sequence of filters provided for you by OpenCV—will run much faster. This is because combining the operations at a local level will mean that your data will stay in the processor cache, get handled, and get sent back to main memory, in just one transaction (as opposed to one for each filter you used in the composition).

The Basics of How Filter Engines Work

The basic objects that are used to represent filters in two dimensions are all derived from the `cv::BaseFilter` class. This class essentially defines the kernel operation. For linear kernels, this can

be thought of as the kernel weights. For nonlinear kernels, in principle, this could be just about any function that maps the weights in the kernel support area to a single number that is the result of the filter.

The object responsible for applying a filter is a `cv::FilterEngine`. The `cv::FilterEngine` class is one of those "classes that do things." When you want to use a `cv::BaseFilter`-derived kernel object, you have to create a `cv::FilterEngine` object to do the work, and pass it a pointer to your `cv::BaseFilter` object when you create it.

The `cv::FilterEngine` Class

The filter engine class is both a container for the base filters, as well as a device for actually applying the filter to an image. When constructed, the filter engine constructor will require either a two-dimensional filter (non-separable case) or two one-dimensional filters (separable case). In addition, you will have to tell it what kinds of images it will operate on, what kind it will return, and what data type to use for its intermediate buffer (more on that in a moment). The way the borders are handled is also a property of the filter engine, as it is responsible for the action of the more atomic filter object it contains on the images provided to the engine. Here is the constructor for the `cv::FilterEngine` class:

```
cv::FilterEngine(
  const cv::Ptr< cv::BaseFilter>&       _filter2D,
  const cv::Ptr< cv::BaseRowFilter>&    _rowFilter,
  const cv::Ptr< cv::BaseColumnFilter>& _columnFilter,
  int                                   srcType,
  int                                   dstType,
  int                                   bufType,
  int                                   _rowBorderType    = cv::BORDER_REPLICATE,
  int                                   _columnBorderType = -1,
  const Scalar&                         _borderValue      = cv::Scalar()
);
```

The first three arguments are used to pass the base filter to the filter engine. We will cover the details of how to make those shortly. What is important at the moment is that you will only use either the first option, `_filter2D`, or the next two, `_rowFilter` and `_columnFilter`. If you are using `_filter2D`, then you should pass `cv::Mat()` (empty matrix constructor) to `_rowFilter` and `_columnFilter`; conversely if you wish to use `_rowFilter` and `_columnFilter`, you should pass `cv::Mat()` to `_filter2D`.

The next three arguments, `srcType`, `dstType`, and `bufType` should all be depth types (e.g., `cv::F32` etc.) All data passed to the filter engine must be of type `srcType`. All final returned data will be of type `dstType`. The buffer type `bufType` must be the same as the source type for non-separable filters. In the case of separable filters, the row filter will be processed first, and its output will be written to a buffer of type `bufType`. The column filter will then be applied on that buffer and the results written to the output area, which will be of the destination type. Thus the row filter must be one that takes arrays of type `srcType` and returns `bufType`, while the column filter must take arrays of type `bufType` and return arrays of type `dstType`.

Row and column borders can be handled differently by the filter engine if you like, by passing different values to `_rowBorderType` and `_columnBorderType`. If you wish, you may pass `-1` (the default value) to `_columnBorderType` to make it the same as the row border type. The border value `_borderValue` is only used for `cv::BORDERTYPE_CONSTANT`, in which case extrapolated pixels will have value `_borderValue`.

You may create a filter engine with the constructor described above, or you may create a filter engine using the default constructor `cv::FilterEngine()`, and then initialize the engine with its `init()` method:

```
cv::FilterEngine::init(
  const cv::Ptr< cv::BaseFilter>&       _filter2D,
  const cv::Ptr< cv::BaseRowFilter>&    _rowFilter,
  const cv::Ptr< cv::BaseColumnFilter>& _columnFilter,
  int                                   srcType,
```

```
  int                                   dstType,
  int                                   bufType,
  int                                   _rowBorderType    = cv::BORDER_REPLICATE,
  int                                   _columnBorderType = -1,
  const Scalar&                         _borderValue      = cv::Scalar()
);
```

Once you have the filter engine, there are several methods that allow you to use the engine to actually do a convolution. They are the methods called `cv::FilterEngine::start()`, `cv::FilterEngine::proceed()`, and `cv::FilterEngine::apply()`. The `start()` methods initializes convolution for processing pixels in a set region of the source image, though it does not actually process the pixels. The `proceed()` method does the processing of pixels in that section of the image after `start()` or a previous call to `proceed()`. The `apply()` method is essentially a higher-level call that processes the entire image at once.

```
virtual int cv::FilterEngine::start(   // Return starting y-position in source image
  cv::Size        wholeSize,           // Size of entire source image
  Rect            roi,                 // Region of interest inside of source image
  int             maxBufRows = -1      // Leave this at -1, or see text below
);
virtual int cv::FilterEngine::start(   // Return starting y-position in source image
  const cv::Mat&  src,                 // Source image
  const cv::Rect& srcRoi     = cv::Rect(0,0,-1,-1), // Region of interest in source
  bool            isolated   = false,  // if true, use edge extrapolation
  int             maxBufRows = -1      // Leave this at -1, or see text below
);
```

Both forms of `cv::FilterEngine::start()` support a *region of interest*, typically abbreviated ROI. This is the portion of the incoming source image that will be acted on by the filter engine. The first form of start() requires to be told both the size of the image that is going to be processed, called `wholeSize`, and the size and location of the ROI, which are together in the `cv::Rect` called `roi`. The argument `maxBufRows` should be left at its default value of `-1`.[21]

The second form is the one you will probably use most often; it takes the source image (and computes the equivalent of `wholeSize` from that), the region of interest, and a new argument called `isolated`. The parameter `isolated` deals with the way off-the-edge pixels are treated for an ROI that is smaller than the entire image. If `isolated` is `true`, those pixels will be extrapolated the same as would be off-the-edge pixels of the image as a whole. If `isolated` is `false`, however, off-the-edge pixels will not be extrapolated. Instead they will be taken from the image outside of the ROI. `cv::FilterEngine::start()` returns an integer indicating the y-position in the image at which processing should begin (as computed from the ROI).

Once you have called `cv::FilterEngine::start()`, you can begin processing your data with `cv::FilterEngine::proceed()`.

---

[21] This parameter is the size of internal ring buffer used for filtering and affects only the processing speed. As `maxBufRows` increases from the minimal value (i.e., `kernel_aperture_height`), the speed will increase until, after a certain value, it will decrease because of the less efficient cache use. The measured difference in speed should be quite small, but you can play with it if filter performance is critical to your application.

In the case of non-separable filters with large apertures however, keep in mind that the function `cv::filter2D()` checks the aperture size and, if it's large enough (~11-by-11 or larger), it uses the fast DFT-based algorithm. This method filters each pixel in $\sim O(\log n)$ time (i.e. time almost does not depend on the aperture size). By contrast, `FilterEngine` always uses the direct algorithm that takes $O(n^2)$ operations for each pixel (in the case of an n-by-n filter). In other words, if you want the highest performance in linear non-separable filtering operations, where aperture size is sufficiently large, it's better to use `cv::filter2D()`, rather than to use `FilterEngine` and try to tune `maxBufRows`.

Thank Vadim Pisarevsky for this important insight.

```
virtual int cv::FilterEngine::proceed( // Returns the number of produced rows
  const uchar* src,                    // Points into source image
  int          srcStep,                // Source image step size
  int          srcCount,               // Number of row steps to process
  uchar*       dst,                    // Points into destination image
  int          dstStep                 // Destination image step size
);
```

When called, `cv::FilterEngine::proceed()` requires to be passed some detailed bookkeeping information. The first argument `src` is the location in memory of the current row being operated on. This means, for example, that the first time you call `cv::FilterEngine::proceed()`, you will need to compute `src` from the y-value returned to you by `cv::FilterEngine::start()`. You will also have to pass the step size from the source image as `srcStep`. The argument `srcCount` is the number or row steps you would like to process in this call to `cv::FilterEngine::proceed()`. Finally, you must give the data pointer and step size for the area to which the results of the filter will be written. These two arguments are `dst` and `dstStep`, respectively. The way in which both `cv::FilterEngine::start()` and `cv::FilterEngine::proceed()` are used is best understood by example. Below is an example from the OpenCV source code, which is a (slightly simplified) implementation of the Laplace operator:

Example 5-4: An example implementation of a Laplace operator using the `cv::FilterEngine()` class

```
void laplace_f(
  const cv::Mat& src,                  // Input image
  cv::Mat&       dst                   // Result image
) {
  CV_Assert( src.type() == cv::F32 );
  dst.create( src.size(), src.type() );

  // get the derivative and smooth kernels for d2I/dx2
  // for d2I/dy2 use the same kernels, just swapped
  cv::Mat kd, ks;
  cv::getSobelKernels( kd, ks, 2, 0, ksize, false, ktype );

  // process 10 source rows at once
  int DELTA = std::min(10, src.rows);
  cv::Ptr< cv::FilterEngine> Fxx = cv::createSeparableLinearFilter(
    src.type(),
    dst.type(),
    kd, ks,
    Point(-1,-1),
    0,
    borderType, borderType,
    Scalar()
  );
  cv::Ptr< cv::FilterEngine> Fyy = cv::createSeparableLinearFilter(
    src.type(),
    dst.type(),
    ks, kd,
    Point(-1,-1),
    0,
    borderType, borderType,
    Scalar()
  );

  int y = Fxx->start(src), dsty = 0, dy = 0;
  Fyy->start(src);
  const uchar* sptr = src.data + y*src.step;

  // allocate the buffers for the spatial image derivatives;
  // the buffers need to have more than DELTA rows, because at the
  // last iteration the output may take max(kd.rows-1,ks.rows-1)
```

```
  // rows more than the input.
  cv::Mat Ixx( DELTA + kd.rows - 1, src.cols, dst.type() );
  cv::Mat Iyy( DELTA + kd.rows - 1, src.cols, dst.type() );

  // inside the loop always pass DELTA rows to the filter
  // (note that the "proceed" method takes care of possibe overflow, since
  // it was given the actual image height in the "start" method)
  // on output you can get:
  //  * < DELTA rows (initial buffer accumulation stage)
  //  * = DELTA rows (settled state in the middle)
  //  * > DELTA rows (when the input image is over, generate
  //                  "virtual" rows using the border mode and filter them)
  // this variable number of output rows is dy
  // dsty is the current output row
  // sptr is the pointer to the first input row in the portion to process
  //
  for( ; dsty < dst.rows; sptr += DELTA*src.step, dsty += dy )
  {
    Fxx->proceed(
      sptr,
      (int) src.step,
      DELTA,
      Ixx.data,
      (int) Ixx.step
    );
    dy = Fyy->proceed(
      sptr,
      (int) src.step,
      DELTA,
      Iyy.data,
      (int) Iyy.step
    );
    if( dy > 0 )
    {
      Mat dstripe = dst.rowRange( dsty, dsty + dy );
      add( Ixx.rowRange(0, dy), Iyy.rowRange(0, dy), dstripe );
    }
  }
}
```

We will cover shortly how functions like `cv::createSeparableLinearFilter()` work, but what is important here is that these functions just return instances of classes derived from the `cv::FilterEngine` base class. Also, notice how the `start()` and `proceed()` methods are used. As you can see, the implementation `laplace_f()` loops through the incoming image in chunks of size `DELTA` and processes each chunk before moving on.

Now that you know how to use `start()` and `proceed()`, you might wonder why it is worth the effort. In fact, the next method we will look at called `cv::FilterEngine::apply()` just does the whole thing in one go.[22] The reason is performance. If you have several things you want to do, like applying the two kernels (or more!) in sequence in our example, it is much better to do everything you want to do to a bit of data and then write it out to the destination image. In this way, you can keep all of the working data in processor cache and your filter will run much faster.

If this kind of performance detail is not critical to your application, you can simply tell a filter engine to just go through an entire image and output the result. This is done with the `cv::FilterEngine::apply()` method.

[22] One additional consequence of this is that when `apply()` calls `start()`, it always passes `maxBufRows` the default value of `-1`.

```
virtual void apply(
  const cv::Mat&  src,                 // Input image
  cv::Mat&        dst                  // Result image
  const cv::Rect& srcRoi   = cv::Rect(0,0,-1,-1), // Region of interest to he processed
  cv::Point       dstOffs  = cv::Point(0,0),      // Offset to write to in destination image
  bool            isolated   = false,             // if true, use edge extrapolation
);
```

Internally, `cv::FilterEngine::apply()` just wraps up a call to `cv::FilterEngine::start()` and then one big call to `cv::FilterEngine::proceed()`, which then goes through the entire image. The only new detail is the argument `dstOffs`, which is the *destination offset*. The destination offset is a point that indicates where in the destination image to write the result of the processed ROI. If left at its default value of `cv::Point(0,0)`, then the result data will be written into the upper-left corner of the destination image. Otherwise, it will be offset appropriately. This is particularly useful when you wish to process some ROI in the source image and write the results to the corresponding location in the destination image.

Filter Engine Builders

OpenCV contains built in functions that will create filter engines for you. It is worth taking a moment to appreciate what this really means. Creating a filter engine means both creating the filters that the engine will apply (i.e., a box filter of a Gaussian filter, etc.), as well as actually implementing the `start()`, `proceed()`, and `apply()` methods of that filter for you. Each of the library routines we will discuss in this section creates an object of a different type, but all of these objects are derived from `cv::FilterEngine`, and so inherit the interface we discussed in the previous section, as well as provide the implementation of that interface.

For each of these `cv::FilterEngine` factories, there is also a corresponding higher-level routine we have already encountered, which (unbeknownst to you) creates the appropriate engine, applies it to your image, and cleans it up, returning you only the result of that computation. For example, `cv::boxFilter()` is really just a higher-level wrapper around `cv::createBoxFilter()` and the subsequent application of the created filter to your image.[23]

You will notice that all of these routines return `cv::Ptr<cv::FilterEngine>`. It is probably worth taking a moment to appreciate what this means. Notice first that this is a good example of where OpenCV uses its smart pointer construct `cv::Ptr<>`. This way you will not have to deallocate the filter engine you create. You can pass the smart pointer around, do what you need to with it, and trust that at the end of whatever you are doing, when there are no more references to the engine around, the engine itself will be deallocated. It is also worth noting that smart pointer, just like the traditional pointer, can be a base class pointer, and can be used to operate on derived class objects. Thus, just because all of these filter builders return a (smart) *pointer* of type `cv::FilterEngine`, it is not correct to conclude that they return *objects* of type `cv::FilterEngine`. Exactly on the contrary, these functions all return their own specific filter engine types that are all derived from `cv::FilterEngine`.

`cv::createBoxFilter()`

You can directly create a box filter using `cv::createBoxFilter()`:

```
cv::Ptr<cv::FilterEngine> cv::createBoxFilter(
  int       srcType,                   // Source image type (e.g., cv::U8)
  int       dstType,                   // Destination image type (e.g., cv::U8)
  cv::Size  ksize,                     // Kernel size
  cv::Point anchor     = cv::Point(-1,-1),  // Location of anchor point
  bool      normalize  = true,              // If true, divide by box area
```

[23] Actually, `cv::boxFilter()` does not literally call `cv::createBoxFilter()` explicitly, but it essentially implements the latter's functionality internally before calling the further subroutines shared by both routines.

```
  int       borderType = cv::BORDER_DEFAULT // Border extrapolation to use
);
```

As you might have expected, the arguments to `cv::createBoxFilter()` are essentially the same as those for `cv::boxFilter()`, with the exception that instead of providing the source and destination images, only the types of those images are required to define the filter. The `ksize` and `anchor` arguments define the size of the kernel and the (optional) anchor point as with `cv::boxFilter`, and the filter can be normalized or not by setting the normalize argument to `true` or `false`. The `borderType` argument has the usual effect.

`cv::createDerivFilter()`

You can create a derivative filter using `cv::createBoxFilter()`:

```
cv::Ptr<cv::FilterEngine> cv::createDerivFilter(
  int srcType,                          // Source image type (e.g., cv::U8)
  int dstType,                          // Destination image type (e.g., cv::U8)
  int dx,                               // order of corresponding derivative in x
  int dy,                               // order of corresponding derivative in y
  int ksize,                            // Kernel size
  int borderType = cv::BORDER_DEFAULT  // Border extrapolation to use
);
```

The arguments to `cv::createDerivFilter()` are essentially the same as those for `cv::Sobel()`, with the exception that instead of providing the source and destination images, only the types of those images are required to define the filter. The `dx` and `dy` arguments define the order of the derivative to be approximated, and `ksize` gives the size of the kernel to be used[24]. The `borderType` argument has the usual effect.

`cv::createGaussianFilter()`

You can directly create a Gaussian filter using `cv::createGaussianFilter()`:

```
cv::Ptr<cv::FilterEngine> cv::createGaussianFilter(
  int      type,                        // Source and destination type (e.g., cv::U8)
  cv::Size ksize,                       // Kernel size
  double   sigmaX,                      // Gaussian half-width in x-direction
  double   sigmaY      = 0.0,           // Gaussian half-width in y-direction
  int      borderType = cv::BORDER_DEFAULT // Border extrapolation to use
);
```

As you might expect by now, the arguments to `cv::createGaussianFilter()` are essentially the same as those for `cv::GaussianBlur()`. In this case, however, there is only one `type` argument for the source and destination; this filter requires both to be of the same type[25]. `sigmaX` and `sigmaY` indicate the variance of the Gaussian in the x- and y-direction respectively, with `sigmaY` being optional (if set to zero `sigmaY` will be equal to `sigmaX`). The `borderType` argument has the usual effect.

`cv::createLinearFilter()`

You can directly create a general linear filter using `cv::createLinearFilter()`:

```
cv::Ptr<cv::FilterEngine> cv::createLinearFilter(
  int               srcType,          // Source image type (e.g., cv::U8)
  int               dstType,          // Destination image type (e.g., cv::U8)
  cv::InputArray    kernel,           // Your own kernel
```

[24] Note that the `ksize` argument for `cv::createDerivFilter()` is an integer, while the `ksize` argument for `cv::createBoxFilter()` was of type `cv::Size`. This is natural because the kernel of a derivative filter will always be square.

[25] If you need to work around this, you can call `cv::getGaussianKernel()` directly, and then `cv::createSeparableLinearFilter()`. (We will get to both of those functions shortly.)

```
  cv::Point         anchor     = cv::Point(-1,-1),  // Location of anchor point
  double            delta = 0,                      // Offset before assignment to dst
  int               rowBorderType    = cv::BORDER_DEFAULT, // Row extrapolation
  int               columnBorderType = -1,                 // Column extrapolation
  const cv::Scalar& borderValue      = cv::Scalar()        // Value for constant borders
);
```

If you have your own linear kernel, you can have OpenCV make it into a filter engine for you with `cv::createLinearFilter()`; this is the filter engine analog of `cv::filter2D()`. This function should only be used when you have a non-separable kernel, as you will get much better performance from `cv::createSeparableLinearFilter()` if your kernel is separable. The arguments are the source type, `srcType`, and destination type, `dstType`, for the input and output images, the `kernel` you want to use, an optional `anchor` point (the default is to use the center of your kernel), an optional offset `delta`, and the border extrapolation parameters. Unlike most of the other similar functions, `cv::createLinearFilter()` allows you to specify the border extrapolation method for the vertical, `rowBorderType`, and the horizontal, `columnBorderType`, separately. If you omit the latter (or set it to `-1`) it will default to being the same as the former. The final `borderValue` argument is used to define the off-the-edge pixel values when either `rowBorderType` or `columnBorderType` is set to `cv::BORDER_CONSTANT`.

`cv::createMorphologyFilter()`

You can create a filter which implements the morphological operations using `cv::createMorphologyFilter()`:

```
cv::Ptr<cv::FilterEngine> cv::createMorphologyFilter(
  int               op,
  int               type,
  cv::InputArray    element,
  cv::Point         anchor           = cv::Point(-1,-1),   // Location of anchor point
  int               rowBorderType    = cv::BORDER_DEFAULT, // Row extrapolation
  int               columnBorderType = -1,                 // Column extrapolation
  const cv::Scalar& borderValue      = cv::Scalar()        // Value for constant borders
)
```

By now you know the pattern. The arguments to `cv::createMorphologyFilter()` are essentially the same as those for `cv::morphologyEx()`. The `op` argument must be one of the operations from Table 5-3, and the type is the type of the source and destination images, which must be the same. The element is your structuring element and the optional anchor lets you locate the anchor point somewhere other than the (default) center of your structuring element. `rowBorderType` and `columBorderType` let you specify the vertical and horizontal border extrapolation methods, and `borderValue` is the constant used in the case that one of the extrapolation methods is `cv::BORDER_CONSTANT`.

`cv::createSeparableLinearFilter()`

In the case of a separable linear kernel, you can directly create a filter using `cv::createSeparableLinearFilter()`:

```
cv::Ptr<cv::FilterEngine> cv::createSeparableLinearFilter(
  int srcType,                        // Source image type (e.g., cv::U8)
  int dstType,                        // Destination image type (e.g., cv::U8)
  cv::InputArray    rowKernel,        // 1-by-N row Kernel
  cv::InputArray    columnKernel,     // M-by-1 column kernel
  cv::Point         anchor          = cv::Point(-1,-1),   // Location of anchor point
  double            delta           = 0,                  // Offset before dst
  int               rowBorderType   = cv::BORDER_DEFAULT, // Row extrapolation
  int               columnBorderType = -1,                // Column extrapolation
  const cv::Scalar& borderValue     = cv::Scalar()        // Value for constant borders
);
```

The `cv::createSeparableLinearFilter()` function is the filter engine analog of the `cv::sepFilter2D()` function. It requires the source type, `srcType`, and the destination type,

dstType. The row and column kernels are each specified separately with rowKernel and columnKernel. anchor, delta, and the border extrapolation arguments all have their usual meanings.

Base Filters

The filter builders we looked at in the last section created the low-level filter object you needed for you, and then wrapped it up in a cv::FilterEngine object that you could then use. Of course, if you were always going to rely on these functions, it would not have been necessary to learn so much about how the filter engine itself actually works or gets constructed.

The real reason to use this is when you are going to create your own filter entirely, and then use that to build a filter engine that does your own special thing.

In this section, we will look at how you create a low-level filter object of your own. Because separability plays such an important role in the performance of filters, and because you would not be making your own base-level filters if you were not concerned about performance, the base filter classes are divided into the non-separable cv::BaseFilter class, and the two classes used for separable filters, cv::BaseRowFilter and cv::BaseColumnFilter.

The cv::BaseFilter Class

Down at the bottom of the image filtering class hierarchy lives cv::BaseFilter. It implements the actual operator that combines some kernel with the appropriate window of an input image to compute the corresponding output image pixel. This mapping need not be linear at all, and can be any general function of the input image pixels and the kernel pixels. The only restriction is that this function must be computable on a row-by-row basis[26].

The definition of the cv::BaseFilter class looks like the following:

```
class cv::BaseFilter {

public:

  virtual ~BaseFilter();

  // To be overriden by the user.
  //
  // runs a filtering operation on the set of rows,
  // "dstcount + ksize.height - 1" rows on input,
  // "dstcount" rows on output,
  // each input row has "(width + ksize.width-1)*cn" elements
  // each output row has "width*cn" elements.
  // the filtered rows are written into "dst" buffer.
  //
  virtual void operator() (
    const uchar** src,
    uchar* dst,
    int dststep,
    int dstcount,
    int width,
    int cn
  ) = 0;

  // resets the filter state (may be needed for IIR filters)
  //
```

[26] This means that the function F, which maps the pixels $\{x_{i,j}\}$ of the image and the values $\{k_{i,j}\}$ of the kernel may contain terms which are functions of multiple values of j—the column index—but may not contain terms which are functions of multiple values of i—the row index.

```
  virtual void reset();

  Size ksize;
  Point anchor;
};
```

As you can see, the method `operator()` is declared pure virtual, and of course this means that you cannot (directly) instantiate objects of type `cv::BaseFilter`. This should not surprise you, however; the intent is for you to be able to derive your own base filter objects off of `cv::BaseFilter`, and to implement the `operator()` and `reset()` functions yourself as appropriate for your own needs.

The `cv::BaseFilter::operator()` is called by the filter engine (into which you have installed your base filter), so the filter engine is going to supply all of the arguments. The `src` and `dst` pointers will point to the source and destination data respectively, but they will point into those images at the proper place. The `src` pointer actually points to an array of pointers, each of which points to one of the rows that you will need to process the kernel (given by the `ksize` member variable). Similarly, the `dst` pointer points to the row in which the destination pixels are located. At the time when the call to `cv::BaseFilter::operator()` is made, these pointers will be "aligned," in the sense that the `dst` points to the first pixel you will be computing, and the pointers in `src` point to the first pixels you will need for the kernel you have. All of this was set up for you by the filter engine. At this level, you do not need to worry about boundary conditions either, as the `src` pointers actually point to a place where the necessary boundary values are already computed. `dststep` is the step size you will need in order to increment the `dst` pointer to compute other destination values, while `dstcount` is the total number of destination values you will need to compute. `width` and `cn` are the width of the image rows to be processed and the number of channels respectively.

The `cv::BaseFilter::reset()` method is only used in cases where the filter maintains an internal state. The method is automatically called before any new image is processed[27].

Once you have created your base filter, you can construct a filter engine to run it with the `cv::FilterEngine()` constructor and you are ready for business.

The `cv::BaseRowFilter` and `cv::BaseColumnFilter` Classes

As we saw when we looked at `cv::FilterEngine`, it can be given either a single non-separable base filter, or a pair of base filters corresponding to the row and column portions of a separable filter. The `cv::BaseRowFilter` and `cv::BaseColumnFilter` classes are how we implement those components of separable filters. Here are the class definitions for these two (virtual base) classes.

```
class cv::BaseRowFilter {

public:
  virtual ~BaseRowFilter();

  // To be overriden by the user.
  //
  // runs filtering operation on the single input row
  // of "width" element, each element is has "cn" channels.
  // the filtered row is written into "dst" buffer.
  //
  virtual void operator()(
    const uchar* src,
    uchar* dst,
    int width,
    int cn
  ) = 0;
```

[27] As an example of where the reset method might be used, the current implementation of the box filter uses a sliding sum buffer, which needs to be cleared before each new call.

```
  int ksize, anchor;
};

class cv::BaseColumnFilter {

public:
  virtual ~BaseColumnFilter();

  // To be overriden by the user.
  //
  // runs a filtering operation on the set of rows,
  // "dstcount + ksize - 1" rows on input,
  // "dstcount" rows on output,
  // each input and output row has "width" elements
  // the filtered rows are written into "dst" buffer.
  //
  virtual void operator()(
    const uchar** src,
    uchar* dst,
    int dststep,
    int dstcount,
    int width
  ) = 0;

  // resets the filter state (may be needed for IIR filters)
  //
  virtual void reset();

  int ksize, anchor;
};
```

The `cv::BaseRowFilter` class is the simplest because row data is sequential in memory, and so there is a little less bookkeeping. The associated `cv::BaseRowFilter::operator()` inherits this relative simplicity. The arguments are just the pointers to the source and destination rows, the `width` of the rows, and the number of channels, `cn`. As with `cv::BaseFilter::operator()`, the source and destination pointers are of type `uchar*`, and so it will be your responsibility to cast them to the correct type.

The `cv::BaseColumnFilter` class needs to handle the non-sequential nature of the required data. As a result, there are the additional arguments `dststep` and `dstcount`, which the row filter did not require[28]. These have the same meaning as they did for the `cv::BaseFilter()` class.

Base Filter Builders

There will be times when you want to build your own filter engine, but will want to use existing filters to build it. In these cases, it is not necessary to go and re-implement the low-level filters you want. Instead, you can ask OpenCV to just give you the base filter object it normally uses for a particular task.

`getLinearFilter()`, `getLinearRowFilter()`, and `getLinearColumnFilter()`

```
cv::Ptr<cv::BaseFilter> cv::getLinearFilter( // Return filter object
  int            srcType,              // Source image type (e.g., cv::U8)
  int            dstType,              // Destination image type (e.g., cv::U8)
  cv::InputArray kernel,               // Your own kernel
  cv::Point      anchor = cv::Point(-1,-1),  // Location of anchor point
  double         delta  = 0,                 // Offset before assignment to dst
  int            bits   = 0                  // For fixed precision integer kernels
```

[28] You will also notice that the number of channels is not present. This is because it is, in effect, now absorbed into `dststep`.

```
);

cv::Ptr<cv::BaseColumnFilter> cv::getLinearColumnFilter( // Return filter object
  int            bufType,              // Buffer image type (e.g., cv::U8)
  int            dstType,              // Destination image type (e.g., cv::U8)
  cv::InputArray columnKernel,         // M-by-1 column kernel
  int            anchor,               // 1-d Location of anchor point
  int            symmetryType,         // See Table 5-5
  double         delta  = 0,           // Offset before assignment to dst
  int            bits   = 0            // For fixed precision integer kernels
);

cv::Ptr<cv::BaseRowFilter> cv::getLinearRowFilter(      // Return filter object
  int            srcType,              // Source image type (e.g., cv::U8)
  int            bufType,              // Buffer image type (e.g., cv::U8)
  cv::InputArray rowKernel,            // 1-by-N row Kernel
  int            anchor,               // 1-d Location of anchor point
  int            symmetryType,         // See Table 5-5
;
```

The function `cv::getLinearFilter()` returns a (smart) pointer to a `cv::BaseFilter`-derived object that implements a general two-dimensional non-separable linear filter based on the `kernel` (and `anchor` point) you provide. You can provide an optional output offset `delta`, which will be applied after your kernel and before the result is written to the destination array. The last argument, `bits`, is used when your kernel is an integer matrix, but you want to use it to represent fixed (but nonzero) precision numbers.

The `cv::getLinearRowFilter()` and `cv::getLinearColumnFilter()` functions return (smart) pointers to `cv::BaseRowFilter` and `cv::BaseColumnFilter` objects, respectively. The one additional argument that is interesting in these two is the `symmetryType` argument. The symmetry type may be any of the options shown in Table 5-5, including combinations of the types where meaningful (e.g., a kernel might be both smooth and symmetrical, in which case `symmetryType` would be `cv::KERNEL_SMOOTH | cv::KERNEL_SYMMETRICAL`).

Table 5-5: Filter kernel symmetry types

Kernel Symmetry	Meaning
`cv::KERNEL_GENERAL`	Generic type, means no special symmetries.
`cv::KERNEL_SYMMETRICAL`	Kernel is symmetrical and anchor is at the center.
`cv::KERNEL_ASYMMETRICAL`	Kernel is asymmetrical and anchor is at the center.
`cv::KERNEL_SMOOTH`	Kernel is greater than or equal to zero everywhere and normalized.
`cv::KERNEL_INTEGER`	All kernel elements are integers.

You should make sure that what you indicate to be the symmetry properties of your kernel are in fact correct. The benefit of this information is that it can lead to a much faster operation of the filter. Alternatively, you can ask OpenCV to automatically inspect your kernel for you, by calling the function `cv::getKernelType( cv::InputArray kernel, cv::Point anchor )`, which will determine the kernel type for you. This is not really the best way to do things if you can compute the answer yourself at coding time—because of the runtime required to inspect the kernel—but can be handy if the kernel itself is being dynamically generated at runtime.

getMorphologyFilter(), getMorphologyRowFilter(), and getMorphologyColumnFilter()

These functions are the analogs of the linear filter generators above, only for morphological filters. As with the linear filters, there is a function for creating a non-separable two-dimensional filter, and two methods for creating the two one-dimensional filter types needed to define a separable kernel:

```
cv::Ptr<cv::BaseFilter> cv::getMorphologyFilter(
  int            op,                  // Morphology operator to use e.g. cv::MOP_OPEN
  int            type,                // Source and destination type (e.g., cv::U8)
  cv::InputArray element,             // Structuring element
  cv::Point      anchor = cv::Point(-1,-1),  // Location of anchor point
);

cv::Ptr<cv::BaseRowFilter> cv::getMorphologyRowFilter(
  int            op,                  // Morphology operator to use e.g. cv::MOP_OPEN
  int            type,                // Source and destination type (e.g., cv::U8)
  int            esize,               // Element size (width)
  int            anchor = -1          // 1-d Location of anchor point
);

cv::Ptr<cv::BaseColumnFilter> cv::getMorphologyColumnFilter(
  int            op,                  // Morphology operator to use e.g. cv::MOP_OPEN
  int            type,                // Source and destination type (e.g., cv::U8)
  int            esize,               // Element size (height)
  int            anchor = -1          // 1-d Location of anchor point
);
```

The first function, `cv::getMorphologyFilter()`, which is used for non-separable kernels expects a morphological operator type `op` (from Table 5-3), the input and output image type given by `type`, a structuring `element` and an optional `anchor` point.

The row and column filter constructors require the same arguments, except that the element size, `esize`, is just an integer that indicates the extent of the row (or column) dimension of the kernel. Similarly, the anchor point, `anchor`, is an integer that, as usual, can be set to `-1` to indicate that the center point of the kernel is to be used.

Kernel Builders

At this point, you have found your way to the absolute bottom of the food chain: the OpenCV filter stack. High-level functions call filter engines. Filter engines call base filter generators. Base filter generators go to the kernel builders to get the actual arrays that represent individual kernels. The two kernel builders that you may want to access yourself are `cv::getDerivKernel()`, which constructs the Sobel and Scharr kernels, and `cv::getGaussianKernel()`, which constructs Gaussian kernels.

cv::getDerivKernel()

The actual kernel array for a derivative filter is generated by `cv::getDerivKernel()`.

```
void cv::getDerivKernels(
  cv::OutputArray kx,
  cv::OutputArray ky,
  int             dx,                 // order of corresponding derivative in x
  int             dy,                 // order of corresponding derivative in y
  int             ksize,              // Kernel size
  bool            normalize = true,   // If true, divide by box area
  int             ktype     = cv::F32   // Type for filter coefficients
);
```

The result of `cv::getDerivKernel()` is placed in the `kx` and `ky` array arguments. You might recall that the derivative type kernels (Sobel and Scharr) are separable kernels. For this reason, you will get back two arrays, one that is 1-by-`ksize` (row coefficients, `kx`) and another that is `ksize`-by-1 (column

coefficients, ky). These are computed from the x- and y-derivative orders dx and dy. The derivative kernels are always square, so the size argument ksize is an integer. ksize can be any of 1, 3, 5, 7, or cv::SCHARR. The normalize argument tells cv::getDerivKernels() if it should normalize the kernel elements "correctly." For situations where you are operating on floating point images, there is no reason not to set normalize to true, but when you are doing operations on integer arrays, it is often more sensible to not normalize the arrays until some later point in your processing, so that you will not throw away precision that you will later need[29]. The final argument ktype indicates the type of the filter coefficients (or equivalently the type of the arrays kx and ky). The value of ktype can be either cv::F32 or cv::F64.

cv::getGaussianKernel()

The actual kernel array for a Gaussian filter is generated by cv::getGaussianKernel().

```
cv::Mat cv::getGaussianKernel(
  int             ksize,              // Kernel size
  double          sigmaX,             // Gaussian half-width
  int             ktype     = cv::F32   // Type for filter coefficients
);
```

As with the derivative kernel, the Gaussian kernel is separable. For this reason cv::getGaussianKernel() computes only a ksize-by-1 array of coefficients. The value of ksize can be any odd positive number. The argument sigma sets the standard deviation of the approximated Gaussian distribution. The coefficients are computed from sigma according to the following function:

$$k_i = \alpha \cdot e^{-\frac{(i-(ksize-1)2)^2}{(2\sigma)^2}}$$

That is, the coefficient alpha is computed such that the filter overall is normalized. sigma may be set to -1, in which case the value of sigma will be automatically computed from the size ksize.[30]

The Filter Architecture up Close (Advanced)

An architecture as nuanced as the filter architecture in OpenCV is something like an automobile. You can choose to not understand it, and hope it works right (in which case, you could pretty much have skipped this entire section of the chapter), you can understand it at a user level, and be content that this will help you not do terribly unwise things that bite you later (like knowing that the oil does, in fact, have to be changed every few thousand miles), or you can really get down and understand the thing, with the mindset of maintaining it, modifying it, and bending it to your will. This above section of the chapter was mainly for that kind of intermediate users, who want to understand and use wisely, but maybe not seriously get down into it. This little subsection, however, is for the benefit of those of you who really want to go do some work yourself in the filter architecture.

To this end, let's consider a simple high-level call to cv::boxFilter(). This call will manage to use just about every nasty detail of the architecture before it is done. Example 5-5 shows what really happens when you call cv::boxFilter() and supply it with a source array, a destination array, and the kernel parameters.

Example 5-5: A detailed walk-through of the operation of a high-level filter function: cv::boxFilter()

```
cv::boxFilter calls cv::createBoxFilter, and passes the kernel parameters.
```

[29] If you are, in fact, going to do this, you will find yourself needing that normalization coefficient at some point. The normalization coefficient you will need is: $2^{ksize*2-dx-dy-2}$.

[30] In this case, $\sigma = 0.3 \cdot \left(\frac{ksize-1}{2} - 1\right) + 0.8$.

- cv::createBoxFilter calls cv::getRowSumFilter() and gives it type information and kernel parameters
  - cv::getRowSumFilter uses the type information to call the template RowSum<>() constructor
    - RowSum<> constructor returns a type-specific object derived from BaseRowFilter. class cv::RowSum<> implements operator() for row summation.
  - cv::getRowSumFilter returns the cv::RowSum<> object
- cv::createBoxFilter calls cv::getColumnSumFilter() and gives it type information and kernel parameters
  - cv::getColumnSumFilter uses the type information to call the template ColumnSum<>() constructor
    - ColumnSum<>() constructor returns a type-specific object derived from BaseColumnFilter. class cv::ColumnSum<> implements operator() for column summation.
  - cv::getColumnSumFilter returns the cv::ColumnSum<> object
- cv::createBoxFilter creates a new cv::FilterEngine object, and passes it the cv::RowSum<> and cv::ColumnSum<> filter objects.
- cv::createBoxFilter returns a pointer to the new cv::FilterEngine object

cv::boxFilter() calls the apply method on the new cv::FilterEngine object it was given. The source and destination images given to cv::boxFilter are passed to the apply() method. (note that apply() would accept an ROI, but in this context one is not generated).

- The (generic) cv::FilterEngine::apply() method calls the (generic) cv::FilterEngine::start() method. It supplies the source image and any source image ROI.
  - If an ROI was passed in, that is recorded in a cv::FilterEngine member variable.
  - cv::FilterEngine::start() does some book-keeping that has to do with whether or not the source image is actually a subwindow of a larger image, we'll ignore this for now.
  - cv::FilterEngine::start() calls another (different) cv::FilterEngine::start() method. It passes this start() method the size of the "true" source image (i.e., if source is a view on some parent, this is the parent), and the location of the source ROI relative to *that* parent image.
    - This cv::FilterEngine::start() allocates the internal 'rows' buffer of the FilterEngine. This is a vector of N char* pointers, where N is set either by the maxBufRows argument to apply (which nobody ever uses) or the kernel height.
    - cv::FilterEngine::start() handles the borders
    - cv::FilterEngine::start() computes the first source row that will be used in the computation of all results. Similarly, cv::FilterEngine::start() computes the last source row that will be used in the computation of all results. These are stored in in a cv::FilterEngine member variable
    - cv::FilterEngine::start() calls the reset() member of the columnFilter member (which is the cv::ColumnSum object constructed above).

cv::FilterEngine::start() (somewhat unnecessarily) returns the computed start source row.

The first cv::FilterEngine::start() method returns the computed first row, but adjusted into the "coordinates" of the parent view (if the source image is actually a view on a larger array.)

The cv::FilterEngine::apply() method computes the destination *pointer* for the first destination pixel. This is just the data area for the source image offset by the "true" start row multiplied by the source step.

The cv::FilterEngine::apply() method computes the total number of rows that will be analyzed (using the difference of the two row value member variables computed earlier by cv::FilterEngine::start()).

The cv::FilterEngine::apply() method computes the address of the first pixel in the destination image to be computed.

cv::FilterEngine::apply() method calls the (generic) cv::FilterEngine::proceed() method, passing it the locations of the first source pixel, the first destination pixel, the source step, the destination step, the total number of rows to be "used" to compute all destination pixels.

For each destination pixel, cv::FilterEngine proceed() calls the member operator() of the rowFilter member of the cv::FilterEngine (our cv::RowSum<> object), passing it the source pointer, the destination pointer, and the width of the given to apply (which in this case is null) or the width of the image.

cv::RowSum<>::operator() then computes a sum across 'width' source pixels accumulating them into the indicated destination pixel. All subsequent destination pixels in the row are then computed from that.

For each destination pixel, proceed() calls the member operator() of the cv::columnFilter member of the cv::FilterEngine (our cv::ColumnSum<> object), passing it an array of source pointers corresponding to the relevant rows of the source image, a pointer to the destination pixel, the destination step, the number of columns that will be summed, and the width of the image (or of any ROI that was given to the filter when it was constructed).

The cv::ColumnSum<>::operator() then computes the sum across count columns and accumulates the result in the destination pixel. Compute the sum for width pixels.

cv::FilterEngine::proceed() returns

cv::FilterEngine::apply() returns

cv::boxFilter() returns

In words, this is what happens. At the top level, the user calls `cv::boxFilter()`, and gives it a source and destination image and the size of the kernel. `cv::boxFilter()`, however, is really just a pretty thin wrapper; its purpose is to create a `cv::FilterEngine` that will handle the box filtering process. In order to do this, however, it will need the actual base filter objects. Because the box filter is a separable filter, two base filter objects are required, one for the horizontal (rows) and one for the vertical (columns).

To create a base filter object, `cv::boxFilter()` will use the `cv::getRowSumFilter()` and `cv::getColumnSumFilter()` methods[31]. What these do is they instantiate and return template base

[31] It is worth mentioning here that these latter two functions are not officially part of the OpenCV API; they are *below the API* functions. This means that you should probably not be calling them yourself, as they are not guaranteed to

filters called `cv::RowSum<>` and `cv::ColumnSum<>`. These objects are templates because each filter needs to know what types are being used for the source and destination images in order to implement their functionality in the most efficient way possible. Recall that it is the `operator()` members of these objects that do the low-level work (which we will get to momentarily).

The `cv::getRowSumFilter()` and `cv::getColumnSumFilter()` methods return `cv::Ptr<>` smart pointers to the base filters, which are then passed to the `cv::FilterEngine` constructor. The constructed filter engine is then returned to `cv::boxFilter()` in the form of another `cv::Ptr<>` smart pointer. Once this pointer to the box filter engine is returned, `cv::boxFilter()` then just calls the `apply()` method of the filter engine. Of course, a lot gets done inside of this method.

The `cv::FilterEngine::apply()` method takes as argument the source and destination images. It would also accept a region of interest (ROI), but none is generated in this context (so the default "whole image" is assumed). This `apply()` method is a generic member of `cv::FilterEngine`, in the sense that it is not overloaded in any way. The filter engine object is not a derived object, but the generic base class element. This is the way the architecture is designed; you should not have to overload `cv::FilterEngine`. (This is in contrast to the base filters, which are interfaces only and always need to be overloaded.)

The `cv::FilterEngine::apply()` then calls `cv::FilterEngine::start()`. There is some opportunity for confusion here, because there are two `start()` methods associated with `cv::FilterEnging()`. The first start method takes the source image and the source ROI (and some other stuff) as arguments. It is responsible for some minor checking and housekeeping, and then it calls the second `start()` method. The main piece of housekeeping it does is to resolve the possibility that the source image is actually a view[32] on a larger array, which has implications for the `isolated` argument, for example. Once this is done, the second `start()` method can be called, and given the *parent array* (i.e., if the current source is a view of some other array, it is that other larger array that is passed), and a corrected ROI (which is the given ROI converted to the coordinates of the larger parent array). The second `start()` routine now does some real work.

The main responsibility of this `start()` method is to set up all of the data structures that are going to be needed for the base filters to operate. This is more effort than it might sound like, because the base filters will not be made aware of things like the boundary type, or if they are in an ROI. They will just be handed data to chew on and they will spit out results. In order to achieve this, there will have to be areas for "off the edge" pixels to be placed alongside the real image pixel values so that the base filters will not have to be aware of the difference. This data goes in a filter engine member variable called `rows` which is an array of `char*` pointers, one for each row of scratch space that will be needed (these actually form a circular buffer so that data only needs to be copied in once, and can then be used until it is no longer needed). `start()` is also responsible for computing the (integer ordinal for the) first and last rows that will be used. Again, these are the first and last row of the image in our case, because there is no ROI, but generally, if `start()` were not being called from inside of `cv::boxFilter()`, we might have gotten one from `cv::FilterEngine::apply()` or directly by whomever else might have called `cv::FilterEngine::start()`. A final responsibility that `cv::FilterEngine::start()` has is to call the `reset()` method on the base column filter. (We will understand why shortly when we get into the base column filter's internal details.) The starting row (in parent image coordinates) is then returned to the first `start()` function, which in turn returns it to `cv::FilterEngine::apply()` in coordinates local to the source image.

---

remain in existence or have stable interfaces in future versions of the library. There will be several things in this current exposition which are of this kind.

[32] Recall that a 'view' refers to the case in which you have a `cv::Mat A`, and then generate a second `cv::Mat B` with a function like `cv::Mat::getSubRect()`. In this case `B` does not have its own data, it has a pointer to the data area in `A` and offsets and strides which make accesses to `B` appear as if `B` were a separate array.

Next, `cv::FilterEngine::apply()` calls `cv::FilterEngine::proceed()`, and passes it the source image data, the source image step, the destination image data, the destination step, and the number of rows from the source image that will be required to compute all destination pixels. It is important not to miss here that, in fact, there is a lot more information provided to `proceed()`, because when `start()` was running, it was computing various things that it stored in member variables of the filter engine (like those scratch buffers for the data rows). The source data and destination data areas given to `proceed()` are the location of the first data needed for actual computation of a required destination pixel. This is a little subtle, because all of the implications of ROIs are handled in `cv::FilterEngine::apply()` before calling `proceed()`. These pointers are not just pointers to the data in the images, they are pointers to the beginning or the relevant data.

`cv::FilterEngine::proceed()` is essentially a big `for` loop over all of the desired destination pixels. It actually handles them on a row by row basis. It counts through the destination rows, handles actual copy of data into the buffers (the ones that `start()` allocated for us), as well as updates the circular buffer, and calls the base filters once the data areas are ready. First it calls the `operator()` member of the base row filter for any given destination pixel, then it calls the `operator()` member of the base column filter for those pixels. (At least, that is what it does in this example. If the filter engine were a non-separable filter, then obviously it would just make a single call to the `operator()` member of the installed two-dimensional filter.)

Inside of the base row filter, which in our example is `cv::RowSum<>`, each pixel in the destination row is computed as a sum over the neighboring pixels. The first such destination pixel is a straight sum. All subsequent pixels are computed from the previous pixel, by subtracting the one contributing pixel that is now out of the window and adding the one pixel that has just entered the window. Inside of the base column filter `cv::ColumnSum<>`, the summation over vertical columns (of the results from the row filter summation) is performed. One important difference between the row and column sum filters is that the column filters need to maintain some internal state—this is to enable a similar trick to what the row filter uses for summing, except that the row filter is given an entire row at a time, while the column filter needs to be called many times in order to complete an entire column.

This call to `cv::FilterEngine::proceed()` finishes up all of its work and returns. `cv::FilterEngine::apply()` returns, and then `cv::boxFilter()` returns. At this point, the results of your computation are sitting in your destination array and you are all done.

```
For each destination pixel, cv::FilterEngine proceed() calls the member operator()
of the rowFilter member of the cv::FilterEngine (our cv::RowSum<> object), passing
it the source pointer, the destination pointer, and the width of the given to
apply (which in this case is null) or the width of the image.

    cv::RowSum<>::operator() then computes a sum across 'width' source pixels
    accumulating them into the indicated destination pixel. All subsequent
    destination pixels in the row are then computed from that.

For each destination pixel, proceed() calls the member operator() of the
cv::columnFilter member of the cv::FilterEngine (our cv::ColumnSum<> object),
passing it an array of source pointers corresponding to the relevant rows of the
source image, a pointer to the destination pixel, the destination step, the number
of columns that will be summed, and the 'width' of the image (or of any ROI that
was given to the filter when it was constructed).

    The cv::ColumnSum<>::operator() then computes the sum across 'count' columns
    and accumulates the result in the destination pixel. Compute the sum for
    'width' pixels.
```

Summary

In this chapter, we learned about general image convolution, including the importance of how boundaries are handled in convolutions. We learned about image kernels, and the difference between linear and nonlinear kernels.

We learned how OpenCV implements a number of common image filters, and learned what those filters do to different kinds of input data.

Finally, we learned how OpenCV actually implements the functions that do convolutions. We saw how the general filter architecture is designed, and learned how we could make our own components for that architecture and thus how to "micromanage" the process to tune custom filters for speed.

Exercises

1. Load an image of an interesting or sufficiently "rich" scene. Using `cv::threshold()`, set the threshold to 128. Use each setting type in Table 5-5 on the image and display the results. You should familiarize yourself with thresholding functions because they will prove quite useful.
   a) Repeat the exercise but use `cv::adaptiveThreshold()` instead. Set `param1=5`.
   b) Repeat part a using `param1=0` and then `param1=-5`.
2. Load an image with interesting textures. Smooth the image in several ways using `cv::smooth()` with `smoothtype=cv::GAUSSIAN`.
   a) Use a symmetric 3-by-3, 5-by-5, 9-by-9 and 11-by-11 smoothing window size and display the results.
   b) Are the output results nearly the same by smoothing the image twice with a 5-by-5 Gaussian filter as when you smooth once with two 11-by-11 filters? Why or why not?
3. Display the filter, creating a 100-by-100 single-channel image. Clear it and set the center pixel equal to 255.
   a) Smooth this image with a 5-by-5 Gaussian filter and display the results. What did you find?
   b) Do this again but now with a 9-by-9 Gaussian filter.
   c) What does it look like if you start over and smooth the image twice with the 5-by-5 filter? Compare this with the 9-by-9 results. Are they nearly the same? Why or why not?
4. Load an interesting image. Again, blur it with `cv::smooth()` using a Gaussian filter.
   a) Set `param1=param2=9`. Try several settings of `param3` (e.g., 1, 4, and 6). Display the results.
   b) This time, set `param1=param2=0` before setting `param3` to 1, 4, and 6. Display the results. Are they different? Why?
   c) Again use `param1=param2=0` but now set `param3=1` and `param4=9`. Smooth the picture and display the results.
   d) Repeat part c but with `param3=9` and `param4=1`. Display the results.
   e) Now smooth the image once with the settings of part c and once with the settings of part d. Display the results.
   f) Compare the results in part e with smoothings that use `param3=param4=9` and `param3=param4=0` (i.e., a 9-by-9 filter). Are the results the same? Why or why not?
5. Create a low-variance random image (use a random number call such that the numbers don't differ by much more than 3 and most numbers are near 0). Load the image into a drawing program such as PowerPoint and then draw a wheel of lines meeting at a single point. Use bilateral filtering on the resulting image and explain the results.
6. In a drawing program such as PowerPoint, draw a series of concentric circles forming a bull's-eye.

a) Make a series of lines going into the bull's-eye. Save the image.

b) Using a 3-by-3 aperture size, take and display the first-order *x*- and *y*-derivatives of your picture. Then increase the aperture size to 5-by-5, 9-by-9, and 13-by-13. Describe the results.

7. Create a new image that is just a 45 degree line, white on black. For a given series of aperture sizes, we will take the image's first-order *x*-derivative (dx) and first-order *y*-derivative (dy). We will then take measurements of this line as follows. The (dx) and (dy) images constitute the gradient of the input image. The magnitude at location (i,j) is $mag(i,j) = \sqrt{dx^2(i,j) + dy^2(i,j)}$ and the angle is $\Theta(i,j) = atan2(dy(i,j), dx(i,j))$. Scan across the image and find places where the magnitude is at or near maximum. Record the angle at these places. Average the angles and report that as the measured line angle.

   a) Do this for a 3-by-3 aperture Sobel filter.

   b) Do this for a 5-by-5 filter.

   c) Do this for a 9-by-9 filter.

   d) Do the results change? If so, why?

8. Find and load a picture of a face where the face is frontal, has eyes open, and takes up most or all of the image area. Write code to find the pupils of the eyes.

   A Laplacian "likes" a bright central point surrounded by dark. Pupils are just the opposite. Invert and convolve with a sufficiently large Laplacian.

9. Use a camera to take two pictures of the same scene while moving the camera as little as possible. Load these images into the computer as `src1` and `src1`.

   a) Take the absolute value of `src1` minus `src1` (subtract the images); call it `diff12` and display. If this were done perfectly, `diff12` would be black. Why isn't it?

   b) Create `cleandiff` by using `cv::erode()` and then `cv::dilate()` on `diff12`. Display the results.

   c) Create `dirtydiff` by using `cv::dilate()` and then `cv::erode()` on `diff12` and then display.

   d) Explain the difference between `cleandiff` and `dirtydiff`.

10. Take a picture of a scene. Then, without moving the camera, put a coffee cup in the scene and take a second picture. Load these images and convert both to 8-bit grayscale images.

    a) Take the absolute value of their difference. Display the result, which should look like a noisy mask of a coffee mug.

    b) Do a binary threshold of the resulting image using a level that preserves most of the coffee mug but removes some of the noise. Display the result. The "on" values should be set to `255`.

    c) Do a `cv::MOP_OPEN` on the image to further clean up noise.

11. Create a clean mask from noise. After completing exercise 10, continue by keeping only the largest remaining shape in the image. Set a pointer to the upper-left of the image and then traverse the image. When you find a pixel of value `255` ("on"), store the location and then flood fill (Chapter 6, `cv::floodFill()`) it using a value of `100`. Read the connected component returned from flood fill and record the area of filled region. If there is another larger region in the image, then flood fill the smaller region using a value of 0 and delete its recorded area. If the new region is larger than the previous region, then flood fill the previous region using the value 0 and delete its location. Finally, fill the remaining largest region with `255`. Display the results. We now have a single, solid mask for the coffee mug.

12. For this exercise, use the mask created in exercise 10 or create another mask of your own (perhaps by drawing a digital picture, or simply use a square). Load an outdoor scene. Now use this mask with `copyTo()`, to copy an image of a mug into the scene.

13. Load an image of a scene and convert it to grayscale.

    a) Run the morphological Top Hat operation on your image and display the results.

b) Convert the resulting image into an 8-bit mask.

c) Copy a grayscale value into the original image where the Top Hat mask (from b) is nonzero. Display the results.

14. Use `cv::filter2D()` to create a filter that detects only 60 degree lines in an image. Display the results on a sufficiently interesting image scene.

15. Refer to the Sobel derivative filter shown in Figure 5-15. Make a kernel of this figure and convolve an image with it. Then, make two kernels of the separable parts. Take the image again, and convolve with first the x and then the y kernels. Subtract these two images to convince yourself that the separable operation produces the exact same result.

16. Separable kernels. Create a 3-by-3 Gaussian kernel using rows [(1/16, 2/16, 1/16), (2/16, 4/16, 2/16), (1/16, 2/16, 1/16)] and with anchor point in the middle.

    a) Run this kernel on an image and display the results.

    b) Now create two one-dimensional kernels with anchors in the center: one going "across" (1/4, 2/4, 1/4), and one going down (1/4, 2/4, 1/4). Load the same original image and use `cv::filter2D()` to convolve the image twice, once with the first 1D kernel and once with the second 1D kernel. Describe the results.

    c) Describe the order of complexity (number of operations) for the kernel in part a) and for the kernels in part b). The difference is the advantage of being able to use separable kernels and the entire Gaussian class of filters—or any linearly decomposable filter that is separable, since convolution is a linear operation.

17. Can you make a separable kernel from the filter shown in Figure 5-15? If so, show what it looks like.

6

General Image Transforms

Overview

In the previous chapter, we covered the class of image transformations that can be understood specifically in terms of convolution. Of course, there are a lot of useful operations that cannot be expressed in this way (i.e., as a little window scanning over the image doing one thing or another). In general, transformations that can be expressed as convolutions are local, meaning that even though they may change the entire image, the effect on any particular pixel is determined by only a small number of pixels around that one. The transforms we will look in this chapter generally will not have this property.

Some very useful *image transforms* are very simple, and you will use them all of the time—*resize* for example. Others are somewhat more special purpose. In many cases, the image transforms we will look at in this chapter have the purpose of converting an image into some entirely different representation. This different representation will usually still be an array of values, but those values might be quite different in meaning than the intensity values in the input image. An example of this would be the frequency representation resulting from a *Fourier transform*. In a few cases, the result of the transformation will be something like a list of components, as would be the case for the *Hough Line Transform*.

There are a number of useful transforms that arise repeatedly in computer vision. OpenCV provides complete implementations of some of the more common ones as well as building blocks to help you implement your own transformations.

Stretch, Shrink, Warp, and Rotate

The simplest image transforms we will encounter are those that resize an image, either to make it larger or smaller. These operations are a little less trivial than you might think, because resizing immediately implies questions about how pixels are interpolated (for enlargement) or merged (for reduction).

Uniform Resize

We often encounter an image of some size that we would like to convert to an image of some other size. We may want to upsize (zoom in) or downsize (zoom out) the image; both of these tasks are accomplished by the same function.

`cv::resize()`

The `cv::resize()` function handles all of our resizing needs. We provide our input image, and the size we would like it be converted to, and it will generate a new image of exactly the desired size.

```
void cv::resize(
  cv::InputArray  src,                          // Input Image
  cv::OutputArray dst,                          // Result image
  cv::Size        dsize,                        // New Size
  double          fx            = 0,            // x-rescale
  double          fy            = 0,            // y-rescale
  int             interpolation = cv::INTER_LINEAR // interpolation method
);
```

We can specify the size of the output image in two ways. One way is to use *absolute sizing*; in this case, the `dsize` argument directly sets the size we would like the result image `dst` to be. The other option is to use *relative sizing*; in this case, we set `dsize` to `cv::Size(0,0)`, and instead set `fx` and `fy` to the scale factors we would like to apply to the x- and y-axes respectively.[1] The last argument is the `interpolation` method, which defaults to linear interpolation. The other available options are shown in Table 6-1.

Table 6-1: `cv::resize()` interpolation options

Interpolation	Meaning
`cv::INTER_NEAREST`	Nearest neighbor
`cv::INTER_LINEAR`	Bilinear
`cv::INTER_AREA`	Pixel area re-sampling
`cv::INTER_CUBIC`	Bicubic interpolation
`cv::INTER_LANCZOS4`	Lanczos interpolation over 8-by-8 neighborhood.

Interpolation is an important issue here. Pixels in the source image sit on an integer grid; for example, we can refer to a pixel at location (20, 17). When these integer locations are mapped to a new image, there can be gaps—either because the integer source pixel locations are mapped to float locations in the destination image and must be rounded to the nearest integer pixel location, or because there are some locations to which no pixels at all are mapped (think about doubling the image size by stretching it; then every other destination pixel would be left blank). These problems are generally referred to as *forward projection* problems. To deal with such rounding problems and destination gaps, we actually solve the problem backwards: we step through each pixel of the destination image and ask, "Which pixels in the source are needed to fill in this destination pixel?" These source pixels will almost always be on fractional pixel locations, so we must interpolate the source pixels to derive the correct value for our destination value. The default method is bilinear interpolation, but you may choose other methods (as shown in Table 6-1).

The easiest approach is to take the resized pixel's value from its closest pixel in the source image; this is the effect of choosing the `interpolation` value `cv::INTER_NEAREST`. Alternatively, we can linearly weight the 2-by-2 surrounding source pixel values according to how close they are to the destination pixel, which is what `cv::INTER_LINEAR` does. We can also virtually place the new resized pixel over the old pixels and then average the covered pixel values, as done with `cv::INTER_AREA`.[2] For yet smoother interpolation, we have the option of fitting a cubic spline between the 4-by-4 surrounding pixels in the source image and then reading off the corresponding destination value from the fitted spline; this is the result of choosing the `cv::INTER_CUBIC` interpolation method. Finally, we have the Lanczos interpolation, which is similar to the cubic method, but uses information from an 8-by-8 area around the pixel.[3]

---

[1] Either `dsize` must be `cv::Size(0,0)` or `fx` and `fy` must both be zero.

[2] At least that's what happens when `cv::resize()` shrinks an image. When it expands an image, `cv::INTER_AREA` amounts to the same thing as `cv::INTER_NEAREST`.

[3] The subtleties of the Lanczos filter are beyond the scope of this book, but this filter is commonly used in processing digital images because it has the effect of increasing the *perceived* sharpness of the image.

It is important to notice the difference between `cv::resize()` and the similarly named `cv::Mat::resize()` member function of the `cv::Mat` class. `cv::resize()` creates a new image, of a different size, over which the original pixels are mapped. The `cv::Mat::resize()` member function resizes the image whose member you are calling, and it crops that image to the new size. Pixels are not interpolated (or extrapolated) in the case of `cv::Mat::resize()`.

Image Pyramids

Image pyramids [Adelson84] are heavily used in a wide variety of vision applications. An image pyramid is a collection of images—all arising from a single original image—that are successively downsampled until some desired stopping point is reached. (Of course, this stopping point could be a single-pixel image!)

There are two kinds of image pyramids that arise often in the literature and in applications: the Gaussian [Rosenfeld80] and Laplacian [Burt83] pyramids [Adelson84]. The *Gaussian pyramid* is used to downsample images, and the Laplacian pyramid (to be discussed shortly) is required when we want to reconstruct an upsampled image from an image lower in the pyramid.

`cv::pyrDown()`

Normally, we produce layer $(i+1)$ in the Gaussian pyramid (we denote this layer G_{i+1}) from layer G_i of the pyramid, by first convolving G_i with a Gaussian kernel and then removing every even-numbered row and column. Of course, in this case, it follows immediately that each image is exactly one-quarter the area of its predecessor. Iterating this process on the input image G_0 produces the entire pyramid. OpenCV provides us with a method for generating each pyramid stage from its predecessor:

```
void cv::pyrDown(
  cv::InputArray  src,                        // Input Image
  cv::OutputArray dst,                        // Result image
  const cv::Size& dstsize = cv::Size()        // Output image size
);
```

The `cv::pyrDown()` method will do exactly this for us if we leave the destination size argument `dstsize` set to its default value of `cv::Size()`. To be even a little more specific, the default size of the output image is `( (src.cols+1)/2, (src.rows+1)/2 )`.[4] Alternatively, we can supply a `dstsize`, which will indicate the size we would like for the output image; `dstsize`, however, must obey some very strict constraints. Specifically:

$$|dstsize.width * 2 - src.cols| \leq 2$$

$$|dstsize.height * 2 - src.rows| \leq 2$$

This restriction means that the destination image is *very close to* half the size of the source image. The use of the `dstsize` argument is only for handling somewhat esoteric cases in which very tight control is needed on how the pyramid is constructed.

`cv::buildPyramid()`

It is a relatively common situation that you have an image, and wish to build a sequence of new images that are each downscaled from their predecessor. The function `cv::buildPyramid()` creates such a stack of images for you in a single call.

```
void cv::buildPyramid(
  cv::InputArray          src,                // Input Image
  cv::OutputArrayOfArrays dst,                // Output Images from pyramid
  int                     maxlevel            // Number of pyramid levels
);
```

[4] The `+1`s are there to make sure odd sized images are handled correctly. They have no effect if the image was even sized to begin with.

The argument `src` is the source image. The argument `dst` is of a somewhat unusual looking type `cv::OutputArrayOfArrays`, but you can think of this as just being an STL `vector<>` or objects of type `cv::OutputArray`. The most common example of this would be `vector<cv::Mat>`. The argument `maxlevel` indicates how many pyramid levels are to be constructed.

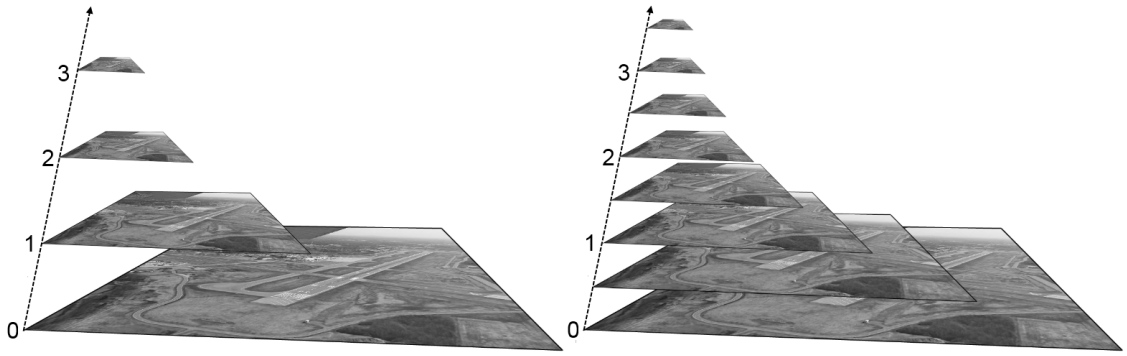


Figure 6-1: An image pyramid generated with `maxlevel=3` (left); two pyramids interleaved together to create a $\sqrt{2}$ pyramid (right)

The argument `maxlevel` is any integer greater than or equal to zero, and indicates the number of pyramid images to be generated. When `cv::buildPyramid()` runs, it will return a vector in `dst` that is of length `maxlevel+1`. The first entry in `dst` will be identical to `src`. The second will be half as large (i.e., as would result from calling `cv::pyrDown()`). The third will be half the size of the second, and so on (left-hand panel of Figure 6-1).

In practice, one often wants a pyramid with a finer logarithmic scaling than factors of two. One way to achieve this is to simply call `cv::resize()` yourself as many times as needed for whatever scale factor you want to use—but this can be quite slow. An alternative (for some common scale factors) is to call `cv::resize()` only once for each interleaved set of images you want, and then call `cv::buildPyramid()` on each of those resized "bases." You can then interleave these results together for one large finer-grained pyramid. Figure 6-1 (right) shows an example in which two pyramids are generated. The original image is first rescaled by a factor of $\sqrt{2}$, and then `cv::buildPyramid()` is called on that one image to make a second pyramid of four intermediate images. Once combined with the original pyramid, the result is a finer pyramid with scale factor of $\sqrt{2}$ across the entire pyramid.

`cv::pyrUp()`

Similarly, we can convert an existing image to an image that is twice as large in each direction by the following analogous (but not inverse!) operation:

```
void cv::pyrUp(
  cv::InputArray  src,                          // Input Image
  cv::OutputArray dst,                          // Result image
  const cv::Size& dstsize = cv::Size()          // Output image size
);
```

In this case, the image is first upsized to twice the original in each dimension, with the new (even) rows filled with 0s. Thereafter, a convolution is performed with the Gaussian filter[5] to approximate the values of the "missing" pixels.

---

[5] This filter is also normalized to four, rather than to one. This is appropriate because the inserted rows have 0s in all of their pixels before the convolution. (Normally, the sum of Gaussian kernel elements would be 1, but in case of 2x

Analogous to `cv::PyrDown()`, if `dstsize` is set to its default value of `cv::Size()`, the resulting image will be exactly twice the size (in each dimension) as `src`. Again, we can supply a `dstsize` that will indicate the size we would like for the output image `dstsize`, but it must again obey some very strict constraints. Specifically:

$$|dstsize.width * 2 - src.cols| \leq (dstsize.width \% 2)$$

$$|dstsize.height * 2 - src.rows| \leq (dstsize.height \% 2)$$

This restriction means that the destination image is *very close to* double the size of the source image. As before, the use of the `dstsize` argument is only for handling somewhat esoteric cases in which very tight control is needed on how the pyramid is constructed.

The Laplacian Pyramid

We noted previously that the operator `cv::pyrUp()` is not the inverse of `cv::pyrDown()`. This should be evident because `cv::pyrDown()` is an operator that loses information. In order to restore the original (higher-resolution) image, we would require access to the information that was discarded by the downsampling. This data forms the *Laplacian pyramid*. The i^{th} layer of the Laplacian pyramid is defined by the relation:

$$L_i = G_i - UP(G_{i+1}) \otimes g_{5x5}$$

Here the operator $UP()$ upsizes by mapping each pixel in location (x, y) in the original image to pixel $(2x + 1, 2y + 1)$ in the destination image; the $\otimes$ symbol denotes convolution; and g_{5x5} is a 5-by-5 Gaussian kernel. Of course, $UP(G_{i+1}) \otimes g_{5x5}$ is the definition of the `cv::pyrUp()` operator provided by OpenCV. Hence, we can use OpenCV to compute the Laplacian operator directly as:

$$L_i = G_i - pyrUp(G_{i+1})$$

The Gaussian and Laplacian pyramids are shown diagrammatically in Table 6-2, which also shows the inverse process for recovering the original image from the sub-images. Note how the Laplacian is really an approximation that uses the difference of Gaussians, as revealed in the preceding equation and diagrammed in the figure.

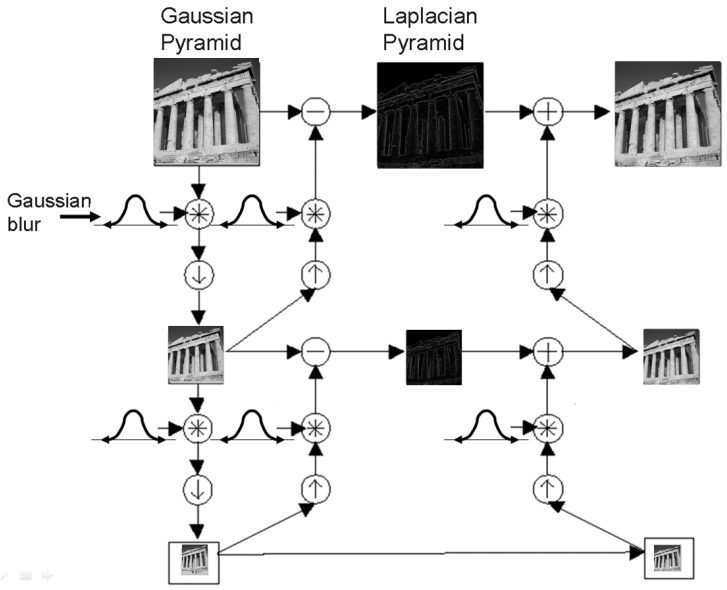


Figure 6-2: The Gaussian pyramid and its inverse, the Laplacian pyramid

---

pyramid up-sampling—in the 2D case—all the kernel elements are multiplied by 4 to recover the average brightness after inserting zero rows and columns.)

Nonuniform Mappings

In this section, we turn to *geometric* manipulations of images, i.e., those transformations that have their origin at the intersection of three-dimensional geometry and projective geometry.[6] Such manipulations include both uniform and nonuniform resizing (the latter is known as *warping*). There are many reasons to perform these operations: for example, warping and rotating an image so that it can be superimposed on a wall in an existing scene, or artificially enlarging a set of training images used for object recognition. [7] The functions that can stretch, shrink, warp, and/or rotate an image are called *geometric transforms* (for an early exposition, see [Semple79]). For planar areas, there are two flavors of geometric transforms: transforms that use a 2-by-3 matrix, which are called *affine transforms*; and transforms based on a 3-by-3 matrix, which are called *perspective transforms* or *homographies*. You can think of the latter transformation as a method for computing the way in which a plane in three dimensions is perceived by a particular observer, who might not be looking straight on at that plane.

An affine transformation is any transformation that can be expressed in the form of a matrix multiplication followed by a vector addition. In OpenCV, the standard style of representing such a transformation is as a 2-by-3 matrix. We define:

$$A \equiv \begin{bmatrix} a_{00} & a_{01} \\ a_{10} & a_{11} \end{bmatrix} \quad B \equiv \begin{bmatrix} b_0 \\ b_1 \end{bmatrix} \quad T \equiv [A \quad B] \quad X \equiv \begin{bmatrix} x \\ y \end{bmatrix} \quad X' \equiv \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

It is easily seen that the effect of the affine transformation $A \cdot X + B$ is exactly equivalent to extending the vector X into the vector X' and simply left-multiplying X' by T.

Affine transformations can be visualized as follows. Any parallelogram $ABCD$ in a plane can be mapped to any other parallelogram $A'B'C'D'$ by some affine transformation. If the areas of these parallelograms are nonzero, then the implied affine transformation is defined uniquely by (three vertices of) the two parallelograms. If you like, you can think of an affine transformation as drawing your image into a big rubber sheet and then deforming the sheet by pushing or pulling[8] on the corners to make different kinds of parallelograms.

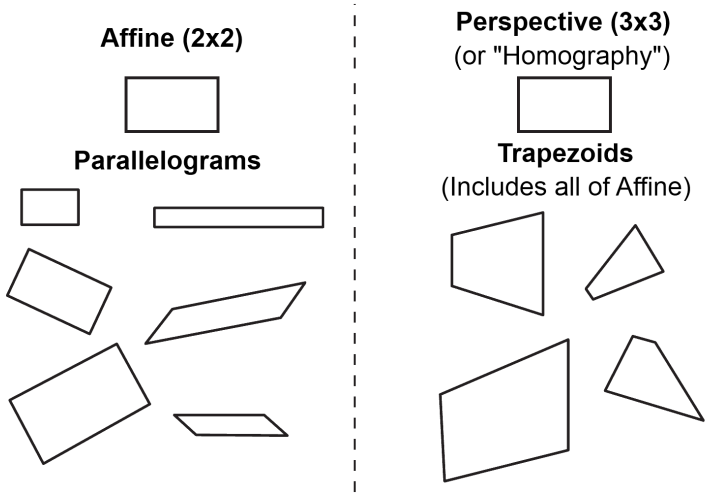


Figure 6-3: Affine and perspective transformations

[6] We will cover these transformations in detail here, and will return to them in Chapter 11 when we discuss how they can be used in the context of three-dimensional vision techniques.

[7] This activity might seem a bit dodgy; after all, wouldn't it be better to just use a recognition method that's invariant to local affine distortions? Nonetheless, this method has a long history and is quite useful in practice.

[8] One can even pull in such a manner as to invert the parallelogram.

When we have multiple images that we know to be slightly different views of the same object, we might want to compute the actual transforms that relate the different views. In this case, affine transformations are often used to model the views because, having fewer parameters, they are easier to solve for. The downside is that true perspective distortions can only be modeled by a *homography*,[9] so affine transforms yield a representation that cannot accommodate all possible relationships between the views. On the other hand, for small changes in viewpoint the resulting distortion is affine, so in some circumstances, an affine transformation may be sufficient.

Affine transforms can convert rectangles to parallelograms. They can squash the shape but must keep the sides parallel; they can rotate it and/or scale it. Perspective transformations offer more flexibility; a perspective transform can turn a rectangle into a trapezoid or any general quadrilateral. Of course, since parallelograms are also trapezoids, affine transformations are a subset of perspective transformations. Figure 6-3 shows examples of various affine and perspective transformations.

Affine Transformation

There are two situations that arise when working with affine transformations. In the first case, we have an image (or a region of interest) we'd like to transform; in the second case, we have a list of points for which we'd like to compute the result of a transformation. Each of these cases if very similar in concept, but quite different in terms of practical implementation. As a result, OpenCV has two different functions for these situations.

`cv::warpAffine()`, Dense affine transformations

In the first case, the obvious input and output formats are images, and the implicit requirement is that the warping assumes the pixels are a *dense representation* of the underlying image. This means that image warping must necessarily handle interpolations so that the output images are smooth and look natural. The affine transformation function provided by OpenCV for dense transformations is `cv::warpAffine()`:

```
void cv::warpAffine(
  cv::InputArray    src,                       // Input Image
  cv::OutputArray   dst,                       // Result image
  cv::InputArray    M,                         // 2-by-3 transformation matrix
  cv::Size          dsize,                     // Destination image size
  int               flags       = cv::INTER_LINEAR,    // Interpolation, and inverse
  int               borderMode  = cv::BORDER_CONSTANT, // Pixel extrapolation method
  const cv::Scalar& borderValue = cv::Scalar() // Used for constant borders
);
```

Here `src` and `dst` are your source and destination arrays. The input `M` is the 2-by-3 matrix we introduced earlier that quantifies the desired transformation. Each element in the destination array is computed from the element of the source array at the location given by:

$$dst(x,y) = src(M_{00}x + M_{01}y + M_{02}, M_{10}x + M_{11}y + M_{12})$$

In general, however, the location indicated by the right-hand side of this equation will not be an integer pixel. In this case, it is necessary to use interpolation to find an appropriate value for $dst(x,y)$. The next argument, `flags`, selects the interpolation method. The available interpolation methods are those in Table 6-1, the same as `cv::resize()`, plus one additional option, `cv::WARP_INVERSE_MAP` (which may be added with the usual Boolean OR). This option is a convenience that allows for inverse warping from `dst` to `src` instead of from `src` to `dst`. The final two arguments are for border extrapolation, and have the same meaning as similar arguments in image convolutions (See Chapter 5).

---

[9] "Homography" is the mathematical term for mapping points on one surface to points on another. In this sense, it is a more general term than used here. In the context of computer vision, homography almost always refers to mapping between points on two image planes that correspond to the same location on a planar object in the real world. It can be shown that such a mapping is representable by a single 3-by-3 orthogonal matrix (more on this in Chapter 11).

`cv::getAffineTransform()`, Computing an Affine Map Matrix

OpenCV provides two functions to help you generate the map matrix `M`. The first is used when you already have two images that you know to be related by an affine transformation or that you'd like to approximate in that way:

```
cv::Mat cv::getAffineTransform(              // Return 2-by-3 matrix
  const cv::Point2f* src,                    // Coordinates three of vertices
  const cv::Point2f* dst                     // Target coordinates three of vertices
);
```

Here `src` and `dst` are arrays containing three two-dimensional (x, y) points. The return value is an array that is the affine transform computed from those points.

The `src` and `dst` in `cv::getAffineTransform()` are just arrays of three points[10] defining two parallelograms. The simplest way to define an affine transform is thus to set `src` to three corners in the source image—for example, the upper- and lower-left together with the upper-right of the source image. The mapping from the source to destination image is then entirely defined by specifying `dst`, the locations to which these three points will be mapped in that destination image. Once the mapping of these three independent corners (which, in effect, specify a "representative" parallelogram) is established, all the other points can be warped accordingly.

Example 6-1 shows some code that uses these functions. In the example, we obtain the `cv::warpAffine()` matrix parameters by first constructing two three-component arrays of points (the corners of our representative parallelogram) and then convert that to the actual transformation matrix using `cv::getAffineTransform()`. We then do an affine warp followed by a rotation of the image. For our array of representative points in the source image, called `srcTri[]`, we take the three points: `(0,0)`, `(0,height-1)`, and `(width-1,0)`. We then specify the locations to which these points will be mapped in the corresponding array `dstTri[]`.

Example 6-1: An affine transformation

```
#include <opencv2/opencv.hpp>
#include <iostream>

using namespace std;

int main(int argc, char** argv) {

  if(argc != 2) {
    cout << "Warp affine\nUsage: ch6_ex6_2 <imagename>\n" << endl;
    return -1;
  }

  cv::Mat src = cv::imread(argv[1],1);
  if( src.empty() ) { cout << "Can not load " << argv[1] << endl; return -1; }

  cv::Point3f srcTri[] = {
     cv::Point2f(0,0),           // src Top left
     cv::Point2f(src.cols-1, 0), // src Top right
     cv::Point2f(0, src.rows-1)  // src Bottom left
  };

  cv::Point2f dstTri[] = {
     cv::Point2f(src.cols*0.f,   src.rows*0.33f), // dst Top left
     cv::Point2f(src.cols*0.85f, src.rows*0.25f), // dst Top right
     cv::Point2f(src.cols*0.15f, src.rows*0.7f)   // dst Bottom left
  };
```

[10] We need just three points because, for an affine transformation, we are only representing a parallelogram. We will need four points to represent a general trapezoid when we address perspective transformations.

```
  // COMPUTE AFFINE MATRIX
  //
  cv::Mat warp_mat = cv::getAffineTransform(srcTri, dstTri);
  cv::Mat dst, dst2;
  cv::warpAffine(
    src,
    dst,
    warp_mat,
    src.size(),
    cv::INTER_LINEAR,
    cv::BORDER_CONSTANT,
    cv::Scalar()
  );
  for( int i = 0; i < 3; ++i )
    cv::circle(dst, dstTri[i], 5, cv::Scalar(255, 0, 255), -1, cv::AA);

  cv::imshow("Affine Transform Test", dst);
  cv::waitKey();

  for(int frame=0;;++frame) {
    // COMPUTE ROTATION MATRIX
    cv::Point2f center(src.cols*0.5f, src.rows*0.5f);
    double angle = frame*3 % 360, scale = (cos((angle - 60)* cv::PI/180) + 1.05)*0.8;

    cv::Mat rot_mat = cv::getRotationMatrix2D(center, angle, scale);

    cv::warpAffine(
      src,
      dst,
      rot_mat,
      src.size(),
      cv::INTER_LINEAR,
      cv::BORDER_CONSTANT,
      cv::Scalar()
    );
    cv::imshow("Rotated Image", dst);
    if(cv::waitKey(30) >= 0 )
      break;
  }
  return 0;
}
```

The second way to compute the map matrix `M` is to use `cv::getRotationMatrix2D()`, which computes the map matrix for a rotation around some arbitrary point, combined with an optional rescaling. This is just one possible kind of affine transformation, but it represents an important subset that has an alternative (and more intuitive) representation that's easier to work with in your head:

```
cv::Mat cv::getRotationMatrix2D(              // Return 2-by-3 matrix
  cv::Point2f  center                         // Center of rotation
  double       angle,                         // Angle of rotation
  double       scale                          // Rescale after rotation
);
```

The first argument, `center`, is the center point of the rotation. The next two arguments give the magnitude of the rotation and the overall rescaling. The function returns the map matrix M, which (as always) is a 2-by-3 matrix of floating-point numbers).

If we define $\alpha = \text{scale} * \cos(\text{angle})$ and $\beta = \text{scale} * \sin(\text{angle})$, then this function computes the matrix `M` to be:

$$\begin{bmatrix} \alpha & \beta & (1-\alpha)\cdot center_x - \beta \cdot center_y \\ -\beta & \alpha & \beta \cdot center_x - (1-\alpha)\cdot center_y \end{bmatrix}$$

You can combine these methods of setting the `map_matrix` to obtain, for example, an image that is rotated, scaled, *and* warped.

`cv::transform()` for Sparse Affine Transformations

We have explained that `cv::warpAffine()` is the right way to handle dense mappings. For sparse mappings (i.e., mappings of lists of individual points), it is best to use `cv::transform()`. You will recall from Chapter 3 that the transform method has the following prototype:

```
void cv::transform(
  cv::InputArray  src,                          // Input N-by-1 array (Ds channele)
  cv::OutputArray dst,                          // Output N-by-1 array (Dd channels)
  cv::InputArray  mtx                           // Transfor matrux (Ds-by-Dd)
);
```

In general, `src` is an N-by-1 array with D_s channels, where N is the number of points to be transformed and D_s is the dimension of those source points. The output array `dst` will be the same size but may have a different number of channels, D_d. The transformation matrix `mtx` is a D_s-by-D_d matrix that is then applied to every element of `src`, after which the results are placed into `dst`.

> Note that `cv::transform()` acts on the channel indices of every point in an array. For the current problem, we assume that the array is essentially a large vector (N-by-1 or 1-by-N) of these multichannel objects. The important thing to remember is that the index that the transformation matrix is relative to is the channel index, not the "vector" index of the large array.

In the case of transformations that are simple rotations, our transformation matrix `mtx` will be a 2-by-2 matrix only, and it can be applied directly to the two-channel indices of `src`. In fact this is true for rotations, stretch, and warp as well in some simple cases. Usually, however, to do a general affine transformation, including translations and rotations about arbitrary centers, and so on, it is necessary to extend the number of channels in `src` to three, so that the action of the more usual 2-by-3 affine transformation matrix is defined. In this case, all of the third-channel entries must be set to `1.0` (i.e., the points must be supplied in homogeneous coordinates). Of course, the output array will still be a two-channel array.

`cv::invertAffineTransform()`, Inverting an Affine Transformation

Given an affine transformation represented as a 2-by-3 matrix, it is often desirable to be able to compute the inverse transformation, which will "put back" all of the transformed points to where they came from. This is done with `cv::invertAffineTransform()`:

```
void cv::invertAffineTransform(
  cv::InputArray  M,                            // Input 2-by-3 matrix
  cv::OutputArray iM                            // Output also a 2-by-3 matrix
);
```

This function takes a 2-by-3 array M and returns another 2-by-3 array `iM` that inverts M. Note that `cv::invertAffineTransform()` does not actually act on any image, it just supplies the inverse transform. Once you have `iM`, you can use it as you would have used `M`, with either `cv::warpAffine()` or `cv::transform()`.

Perspective Transformation

To gain the greater flexibility offered by perspective transforms (homographies), we need a new function that will allow us to express this broader class of transformations. First we remark that, even though a perspective projection is specified completely by a single matrix, the projection is not actually a linear

transformation. This is because the transformation requires division by the final dimension (usually Z; see Chapter 11) and thus loses a dimension in the process.

As with affine transformations, image operations (dense transformations) are handled by different functions than transformations on point sets (sparse transformations).

`cv::warpPerspective()`, Dense perspective transform

The dense perspective transform uses an OpenCV function that is analogous to the one provided for dense affine transformations. Specifically, `cv::warpPerspective()` has all of the same arguments as `cv::warpAffine()`, except with the small, but crucial, distinction that the map matrix must now be 3-by-3.

```
void cv::warpPerspective(
  cv::InputArray    src,                       // Input Image
  cv::OutputArray   dst,                       // Result image
  cv::InputArray    M,                         // 3-by-3 transformation matrix
  cv::Size          dsize,                     // Destination image size
  int               flags       = cv::INTER_LINEAR,    // Interpolation, and inverse
  int               borderMode  = cv::BORDER_CONSTANT, // Pixel extrapolation method
  const cv::Scalar& borderValue = cv::Scalar() // Used for constant borders
);
```

Each element in the destination array is computed from the element of the source array at the location given by:

$$dst(x,y) = src\left(\frac{M_{00}x + M_{01}y + M_{02}}{M_{20}x + M_{21}y + M_{22}}, \frac{M_{10}x + M_{11}y + M_{12}}{M_{20}x + M_{21}y + M_{22}}\right)$$

As with the affine transformation, the location indicated by the right side of this equation will not (generally) be an integer location. Again the flags argument is used to select the desired interpolation method, and has the same possible values as the corresponding argument to `cv::warpAffine()`.

`cv::getPerspectiveTransform()`, Computing the perspective map matrix

As with the affine transformation, for filling the `map_matrix` in the preceding code we have a convenience function that can compute the transformation matrix from a list of point correspondences:

```
cv::Mat cv::getPerspectiveTransform(           // Return 3-by-3 matrix
  const cv::Point2f* src,                      // Coordinates of four vertices
  const cv::Point2f* dst                       // Target coordinates of four vertices
);
```

The `src` and `dst` argument are now arrays of four (not three) points, so we can independently control how the corners of (typically) a rectangle in `src` are mapped to (generally) some rhombus in `dst`. Our transformation is completely defined by the specified destinations of the four source points. As mentioned earlier, for perspective transformations, the return value will be a 3-by-3 array; see Example 6-2 for sample code. Other than the 3-by-3 matrix and the shift from three to four control points, the perspective transformation is otherwise exactly analogous to the affine transformation we already introduced.

Example 6-2: Code for perspective transformation

```
#include <opencv2/opencv.hpp>
#include <iostream>

using namespace std;

int main(int argc, char** argv) {

  if(argc != 2) {
    cout << "Perspective Warp\nUsage: ch6_ex6_3 <imagename>\n" << endl;
    return -1;
  }
```

```
  Mat src = cv::imread(argv[1],1);
  if( src.empty() ) { cout << "Can not load " << argv[1] << endl; return -1; }

  cv::Point2f srcQuad[] = {
    cv::Point2f(0,          0),          // src Top left
    cv::Point2f(src.cols-1, 0),          // src Top right
    cv::Point2f(src.cols-1, src.rows-1), // src Bottom right
    cv::Point2f(0,          src.rows-1)  // src Bottom left
  };

  cv::Point2f dstQuad[] = {
    cv::Point2f(src.cols*0.05f, src.rows*0.33f),
    cv::Point2f(src.cols*0.9f,  src.rows*0.25f),
    cv::Point2f(src.cols*0.8f,  src.rows*0.9f),
    cv::Point2f(src.cols*0.2f,  src.rows*0.7f)
  };

  // COMPUTE PERSPECTIVE MATRIX
  cv::Mat warp_mat = cv::getPerspectiveTransform(srcQuad, dstQuad);
  cv::Mat dst;
  cv::warpPerspective(src, dst, warp_mat, src.size(), cv::INTER_LINEAR,
                      cv::BORDER_CONSTANT, cv::Scalar());
  for( int i = 0; i < 4; i++ )
    cv::circle(dst, dstQuad[i], 5, cv::Scalar(255, 0, 255), -1, cv::AA);

  cv::imshow("Perspective Transform Test", dst);
  cv::waitKey();
  return 0;
}
```

`cv::perspectiveTransform()`, Sparse perspective transformations

There is a special function, `cv::perspectiveTransform()`, that performs perspective transformations on lists of points. Because `cv::transform()` is limited to linear operations, it does not conveniently handle perspective transforms. This is because such transformations require division by the third coordinate of the homogeneous representation ($x = f * X/Z, y = f * Y/Z$). The special function `cv::perspectiveTransform()` takes care of this for us:

```
void cv::perspectiveTransform(
  cv::InputArray  src,                      // Input N-by-1 array (2 or 3 channele)
  cv::OutputArray dst,                      // Output N-by-1 array (2 or 3 channels)
  cv::InputArray  mtx                       // Transfor matrux (3-by-3 or 4-by-4)
);
```

As usual, the `src` and `dst` arguments are, respectively, the array of source points to be transformed and the array of destination points resulting from the transformation. These arrays should be two- or three-channel arrays. The matrix `mat` can be either a 3-by-3 or a 4-by-4 matrix. If it is 3-by-3, then the projection is from two dimensions to two; if the matrix is 4-by-4, then the projection is from three dimensions to three.

In the current context, we are transforming a set of points in an image to another set of points in an image, which sounds like a mapping from two dimensions to two dimensions. This is not exactly correct, however, because the perspective transformation is actually mapping points on a two-dimensional plane embedded in a three-dimensional space back down to a (different) two-dimensional subspace. Think of this as being just what a camera does (we will return to this topic in greater detail when discussing cameras in later chapters). The camera takes points in three dimensions and maps them to the two dimensions of the camera imager. This is essentially what is meant when the source points are taken to be in "homogeneous coordinates." We are adding a dimension to those points by introducing the Z dimension and then setting all of the Z values to 1. The projective transformation is then projecting back out of that space onto the two-dimensional space of our output. This is a rather long-winded way of explaining why, when mapping points in one image to points in another, you will need a 3-by-3 matrix.

Outputs of the code in Example 6-1 and Example 6-2 are shown in Figure 6-4 for affine and perspective transformations. In these examples, we transform actual images; you can compare these with the simple diagrams of Figure 6-3.

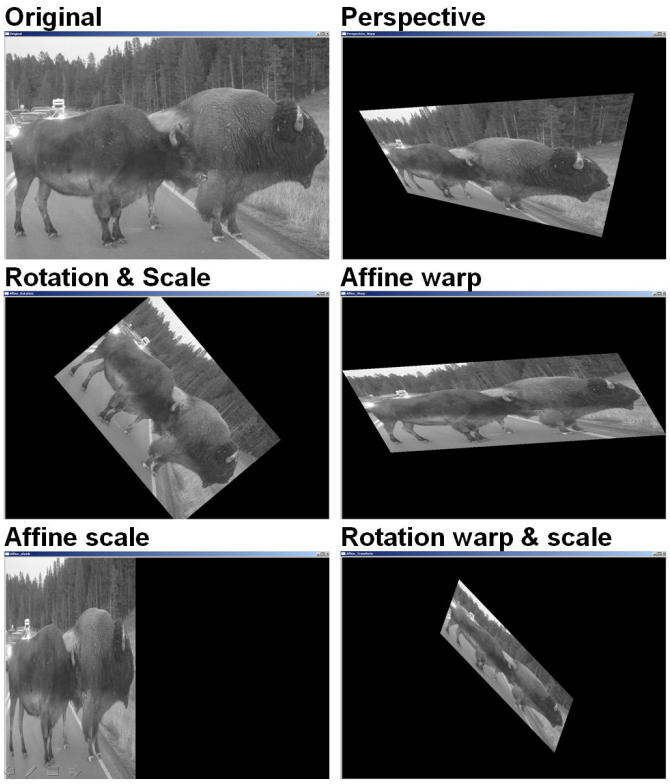


Figure 6-4: Perspective and affine mapping of an image

General Remappings

The affine and perspective transformations we have seen so far are actually specific cases of a more general process. Under the hood, those two both have the same basic behavior. They take pixels from one place in the source image and map them to another place in the destination image. In fact, there are other useful operations that have the same structure. In this section, we will look at another few transformations of this kind, and then look at how OpenCV makes it possible to implement your own general mapping transformations.

Polar Mappings

In Chapter 3, we briefly encountered two functions, `cv::cartToPolar()` and `cv::polarToCart()`, which could be used to convert arrays of points in an x-y Cartesian representation to (or from) arrays of points in an r-θ polar representation.

There is a slight style inconsistency here between the polar mapping functions and the perspective and affine transformation functions. The polar mapping functions expect pairs of single-channel arrays, rather than double-channel arrays as their way of representing two-dimensional vectors. This difference has its origin in the way the two functions are traditionally used, rather than any intrinsic difference between what they are doing.

The functions `cv::cartToPolar()` and `cv::polarToCart()` are employed by more complex routines such as `cv::logPolar()` (described later) but are also useful in their own right.

`cv::cartToPolar()`, Converting from Cartesian to Polar Coordinates

For the case of mapping from Cartesian coordinates to polar coordinates, we have the function `cv::cartToPolar()`:

```
void cv::cartToPolar(
  cv::InputArray  x,                          // Input single channel x-array
  cv::InputArray  y,                          // Input single channel y-array
  cv::OutputArray magnitude,                  // Output single channel mag-array
  cv::OutputArray angle,                      // Output single channel angle-array
  bool            angleInDegrees = false      // Set true for degrees, else radians
);
```

The first two arguments `x`, and `y`, are single-channel arrays. Conceptually, what is being represented here is not just a list of points, but a *vector field*[11]—with the x-component of the vector field at any given point being represented by the value of the array `x` at that point, and the y-component of the vector field at any given point being represented by the value of the array `y` at that point. Similarly, the result of this function appears in the arrays `magnitude` and `angle`, with each point in `magnitude` representing the length of the vector at that point in `x` and `y`, and each point in `angle` representing the orientation of that vector. The angles recorded in `angle` will, by default, be in radians, i.e., $[0, 2\pi)$. If the argument `angleInDegrees` is set to `true`, however, then the angles array will be recorded in degrees $[0, 360)$.

As an example of where you might use this function, suppose you have already taken the x- and y-derivatives of an image, either by using `cv::Sobel()` or by using convolution functions via `cv::DFT()` or `cv::filter2D()`. If you stored the x-derivatives in an image `dx_img` and the y-derivatives in `dy_img`, you could now create an edge-angle recognition histogram. That is, you could then collect all the angles provided the magnitude or strength of the edge pixel is above some desired threshold. To calculate this, we would first create two new destination images (and call them `img_mag` and `img_angle`, for example) for the directional derivatives and then use the function `cvCartToPolar( dx_img, dy_img, img_mag, img_angle, 1 )`. We would then fill a histogram from `img_angle` as long as the corresponding "pixel" in `img_mag` is above our desired threshold.

> In Chapter 13, we will discuss image recognition and image features. This process is actually the basis of how an important image feature used in object recognition, called *HOG* (histogram of oriented gradients), is calculated.

`cv::polarToCart()`, Converting from Polar to Cartesian Coordinates

The function `cv::cartToPolar()` performs the reverse mapping from polar coordinates to Cartesian coordinates.

```
void cv::polarToCart(
  cv::InputArray  magnitude,                  // Output single channel mag-array
  cv::InputArray  angle,                      // Output single channel angle-array
  cv::OutputArray x,                          // Input single channel x-array
  cv::OutputArray y,                          // Input single channel y-array
  bool            angleInDegrees = false      // Set true for degrees, else radians
);
```

The inverse operation is also often useful, allowing us to convert from polar back to Cartesian coordinates. It takes essentially the same arguments as `cv::cartToPolar()`, with the exception that `magnitude` and `angle` are now inputs, and `x` and `y` are now the results.

[11] If you are not familiar with the concept of a vector field, it is sufficient for our purposes to just think of this as a two component vector associated with every point in "image."

LogPolar

For two-dimensional images, the log-polar transform [Schwartz80] is a change from Cartesian to *log-polar* coordinates: $(x, y) \leftrightarrow re^{i\theta}$, where $r = \sqrt{x^2 + y^2}$ and $\theta = atan2(y, x)$. Next, to separate out the polar coordinates into a (ρ, θ) space that is relative to some center point (x_c, y_c); we take the log so that $\rho = \log(\sqrt{(x - x_c)^2 + (y - y_c)^2})$ and $\theta = atan2(y - y_c, x - x_c)$. For image purposes—when we need to "fit" the interesting stuff into the available image memory—we typically apply a scaling factor m to ρ. Figure 6-5 shows a square object on the left and its encoding in log-polar space.

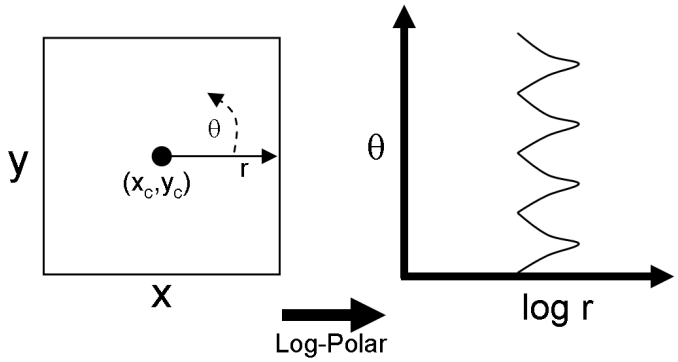


Figure 6-5: The log-polar transform maps (x, y) into $(log(r), \theta)$. Here, a square is displayed in the log-polar coordinate system

The next question is, of course, "Why bother?" The log-polar transform takes its inspiration from the human visual system. Your eye has a small but dense center of photoreceptors in its center (the *fovea*), and the density of receptors falls off rapidly (exponentially) from there. Try staring at a spot on the wall and holding your finger at arm's length in your line of sight. Then, keep staring at the spot and move your finger slowly away; note how the detail rapidly decreases as the image of your finger moves away from your fovea. This structure also has certain nice mathematical properties (beyond the scope of this book) that concern preserving the angles of line intersections.

More important for us is that the log-polar transform can be used to create two-dimensional invariant representations of object views by shifting the transformed image's center of mass to a fixed point in the log-polar plane; see

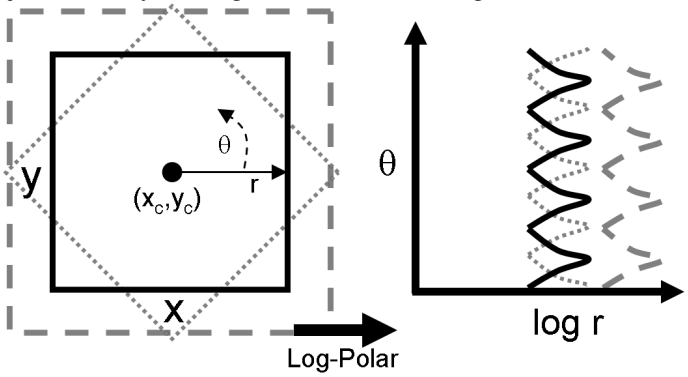


Figure 6-6. On the left are three shapes that we want to recognize as "square." The problem is, they look very different. One is much larger than the others and another is rotated. The log-polar transform appears on the right in

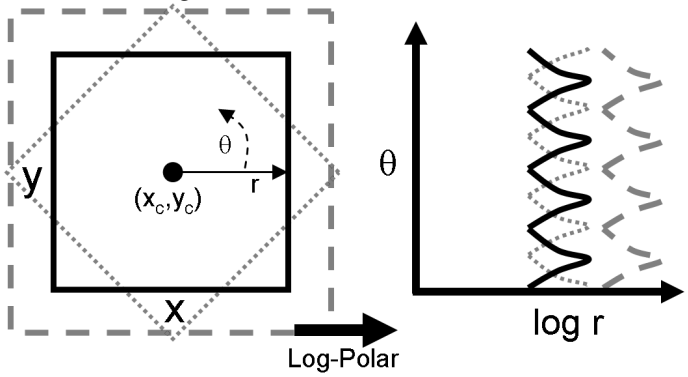


Figure 6-6. Observe that size differences in the (x, y) plane are converted to shifts along the $\log(r)$ axis of the log-polar plane and that the rotation differences are converted to shifts along the θ-axis in the log-polar plane. If we take the transformed center of each transformed square in the log-polar plane and then re-center that point to a certain fixed position, then all the squares will show up identically in the log-polar plane. This yields a type of invariance to two-dimensional rotation and scaling.[12]

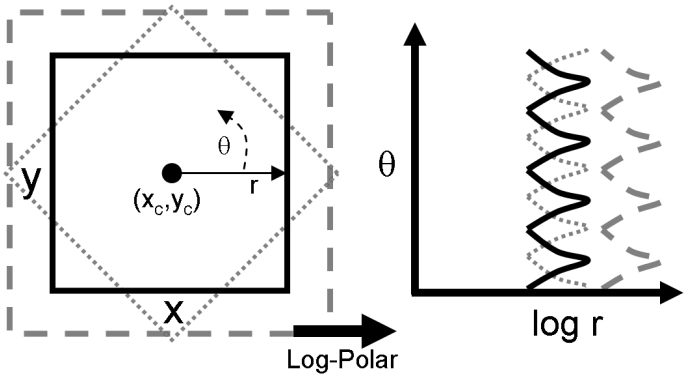


Figure 6-6: Log-polar transform of rotated and scaled squares: size goes to a shift on the $\log(r)$ axis and rotation to a shift on the θ-axis

cv::logPolar()

The OpenCV function for a log-polar transform is `cv::logPolar()`:

```
void cv::logPolar(
  cv::InputArray  src,                          // Input image
  cv::OutputArray dst,                          // Output image
  cv::Point2f     center,                       // Center of transform
  double          m,                            // Scale factor
  int             flags = cv::INTER_LINEAR
                        | cv::WARP_FILL_OUTLIERS // interpolation and fill modes
);
```

[12] In Chapter 13, we'll learn about recognition. For now, simply note that it wouldn't be a good idea to derive a log-polar transform for a whole object because such transforms are quite sensitive to the exact location of their center points. What is more likely to work for object recognition is to detect a collection of key points (such as corners or blob locations) around an object, truncate the extent of such views, and then use the centers of those key points as log-polar centers. These local log-polar transforms could then be used to create local features that are (partially) scale- and rotation-invariant and that can be associated with a visual object.

The `src` and `dst` are the usual input and output images. The parameter `center` is the center point (x_c, y_c) of the log-polar transform; `m` is the scale factor, which should be set so that the features of interest dominate the available image area. The `flags` parameter allows for different interpolation methods. The interpolation methods are the same set of standard interpolations available in OpenCV (Table 6-1). The interpolation methods can be combined with either or both of the flags `cv::WARP_FILL_OUTLIERS` (to fill points that would otherwise be undefined) or `cv::WARP_INVERSE_MAP` (to compute the reverse mapping from log-polar to Cartesian coordinates).

Sample log-polar coding is given in Example 6-4, which demonstrates the forward and backward (inverse) log-polar transform. The results on a photographic image are shown in Figure 6-7.

Figure 6-7: Log-polar example on an elk with transform centered at the white circle on the left; the output is on the right

Example 6-3: Log-polar transform example

```
#include <opencv2/opencv.hpp>
#include <iostream>

using namespace std;

int main(int argc, char** argv) {

  if(argc != 3) {
    cout << "LogPolar\nUsage: ch6_ex6_4 <imagename> <M value>\n"
      <<"<M value>~30 is usually good enough\n";
    return -1;
  }

  cv::Mat src = cv::imread(argv[1],1);

  if( src.empty() ) { cout << "Can not load " << argv[1] << endl; return -1; }

  double M = atof(argv[2]);
  cv::Mat dst(src.size(), src.type()), src2(src.size(), src.type());

  cv::logPolar(
    src,
    dst,
    cv::Point2f(src.cols*0.5f, src.rows*0.5f),
    M,
    cv::INTER_LINEAR | cv::WARP_FILL_OUTLIERS
  );
  cv::logPolar(
    dst,
    src2,
    cv::Point2f(src.cols*0.5f, src.rows*0.5f),
    M,
```

```
    cv::INTER_LINEAR | cv::WARP_INVERSE_MAP
  );
  cv::imshow( "log-polar", dst );
  cv::imshow( "inverse log-polar", src2 );
  cv::waitKey();
  return 0;
}
```

Arbitrary Mappings

We sometimes want to accomplish interpolation programmatically; that is, we'd like to apply some known algorithm that will determine the mapping. In other cases, however, we'd like to do this mapping ourselves. Before diving into some methods that will compute (and apply) these mappings for us, let's take a moment to look at the function responsible for applying the mappings that these other methods rely upon.

One common use of `cv::remap()` is to rectify (correct distortions in) calibrated and stereo images. We will see functions in Chapters 11 and 12 that convert calculated camera distortions and alignments into `mapx` and `mapy` parameters.

The OpenCV function we want is called `cv:remap()`:

`cv::remap()`, General Image Remapping

```
void cv::remap(
  cv::InputArray    src,                        // Input image
  cv::OutputArray   dst,                        // Output image
  cv::InputArray    map1,                       // target x-location for src pixels
  cv::InputArray    map2,                       // target y-location for src pixels
  int               interpolation = cv::INTER_LINEAR,    // Interpolation, and inverse
  int               borderMode    = cv::BORDER_CONSTANT, // Pixel extrapolation method
  const cv::Scalar& borderValue   = cv::Scalar()         // Used for constant borders
);
```

The first two arguments of `cv::remap()` are the source and destination images, respectively. The next two arguments, `map1` and `map2`, indicate where any particular pixel is to be relocated. This is how you specify your own general mapping. These should be the same size as the source and destination images, and must be one of the following data types: `cv::S16C2`, `cv::F32C1`, or `cv::F32C2`. Non-integer mappings are allowed: `cv::remap()` will do the interpolation calculations for you automatically.

The next argument, `interpolation`, contains flags that tell `cv::remap()` exactly how that interpolation is to be done. Any one of the values listed in Table 6-1 will work – except for `cv::INTER_AREA`, which is not implemented for `cv::remap()`.

Discrete Fourier Transform

For any set of values that are indexed by a discrete (integer) parameter, it is possible to define a *discrete Fourier transform* (DFT)[13] in a manner analogous to the Fourier transform of a continuous function. For N complex numbers $x_0, x_1, x_2, \ldots, x_{N-1}$, the one-dimensional DFT is defined by the following formula (where $i = \sqrt{-1}$):

$$g_k = \sum_{n=0}^{N-1} f_n e^{-\frac{2\pi i}{N} kn}$$

[13] Joseph Fourier [Fourier] was the first to find that some functions can be decomposed into an infinite series of other functions, and doing so became a field known as Fourier analysis. Some key text on methods of decomposing functions into their Fourier series are Morse for physics [Morse53] and Papoulis in general [Papoulis62]. The fast Fourier transform was invented by Cooley and Tukey in 1965 [Cooley65] though Carl Gauss worked out the key steps as early as 1805 [Johnson84]. Early use in computer vision is described by Ballard and Brown [Ballard82].

A similar transform can be defined for a two-dimensional array of numbers (of course higher-dimensional analogues exist also):

$$g_{k_x,k_y} = \sum_{n_x=0}^{N_x-1} \sum_{n_y=0}^{N_y-1} f_{n_x,n_y} e^{-\frac{2\pi i}{N}(k_x n_x + k_y n_y)}$$

In general, one might expect that the computation of the N different terms g_k would require $O(N^2)$ operations. In fact, there are several *fast Fourier transform* (FFT) algorithms capable of computing these values in $O(N \log N)$ time.

`cv::dft()`, the Discrete Fourier Transform

The OpenCV function `cv::dft()` implements one such FFT algorithm. The function `cv::dft()` can compute FFTs for one- and two-dimensional arrays of inputs. In the latter case, the two-dimensional transform can be computed or, if desired, only the one-dimensional transforms of each individual row can be computed (this operation is much faster than calling `cv::dft()` several times):

```
void cv::dft(
  cv::InputArray    src,                    // Input array (real or complex)
  cv::OutputArray   dst,                    // Output array
  int               flags       = 0,        // for inverse, or other variations
  int               nonzeroRows = 0         // number of rows to not ignore
);
```

The input array must be of floating-point type and may be single- or double-channel. In the single-channel case, the entries are assumed to be real numbers and the output will be packed in a special space-saving format called *CCS* or *Complex Conjugate Symmetrical*.[14] If the source and channel are two-channel matrices or images, then the two channels will be interpreted as the real and imaginary components of the input data. In this case, there will be no special packing of the results, and some space will be wasted with a lot of 0s in both the input and output arrays.[15]

The special packing of result values that is used with single-channel CCS output is as follows.

For a one-dimensional array:

$Re\ Y_0$	$Re\ Y_1$	$Im\ Y_1$	$Re\ Y_2$	$Im\ Y_2$	...	$Re\ Y_{(N/2-1)}$	$Im\ Y_{(N/2-1)}$	$Re\ Y_{(N/2)}$

For a two-dimensional array:

$Re\ Y_{00}$	$Re\ Y_{01}$	$Im\ Y_{01}$	$Re\ Y_{02}$	$Im\ Y_{02}$	...	$Re\ Y_{0,\frac{N_x}{2}-1}$	$Im\ Y_{0,\frac{N_x}{2}-1}$	$Re\ Y_{0,\frac{N_x}{2}}$
$Re\ Y_{10}$	$Re\ Y_{11}$	$Im\ Y_{11}$	$Re\ Y_{12}$	$Im\ Y_{12}$	...	$Re\ Y_{1,\frac{N_x}{2}-1}$	$Im\ Y_{1,\frac{N_x}{2}-1}$	$Re\ Y_{1,\frac{N_x}{2}}$
$Re\ Y_{20}$	$Re\ Y_{21}$	$Im\ Y_{21}$	$Re\ Y_{22}$	$Im\ Y_{22}$	...	$Re\ Y_{2,\frac{N_x}{2}-1}$	$Im\ Y_{2,\frac{N_x}{2}-1}$	$Re\ Y_{2,\frac{N_x}{2}}$
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
$Re\ Y_{\frac{N_y}{2}-1,0}$	$Re\ Y_{N_y-3,1}$	$Im\ Y_{N_y-3,1}$	$Re\ Y_{N_y-3,2}$	$Im\ Y_{N_y-3,2}$	...	$Re\ Y_{N_y-3,\frac{N_x}{2}-1}$	$Im\ Y_{N_y-3,\frac{N_x}{2}-1}$	$Re\ Y_{N_y-3,\frac{N_x}{2}}$

[14] As a result of this compact representation, the size of the output array for a single-channel image is the same as the size of the input array because the elements that are provably zero are omitted. In the case of the two-channel (complex) array, the output size will, of course, also be equal to the input size.

[15] When using this method, you must be sure to explicitly set the imaginary components to 0 in the two-channel representation. An easy way to do this is to create a matrix full of 0s using `cv::Mat::zeros()` for the imaginary part and then to call `cv::merge()` with a real-valued matrix to form a temporary complex array on which to run `cv::dft()` (possibly in-place). This procedure will result in full-size, unpacked, complex matrix of the spectrum.

$Im\ Y_{\frac{N_y}{2}-1,0}$	$Re\ Y_{N_y-2,1}$	$Im\ Y_{N_y-2,1}$	$Re\ Y_{N_y-2,2}$	$Im\ Y_{N_y-2,2}$	...	$Re\ Y_{N_y-2,\frac{N_x}{2}-1}$	$Im\ Y_{N_y-2,\frac{N_x}{2}-1}$	$Re\ Y_{N_y-2,\frac{N_x}{2}}$
$Re\ Y_{\frac{N_y}{2},0}$	$Re\ Y_{N_y-1,1}$	$Im\ Y_{N_y-1,1}$	$Re\ Y_{N_y-1,2}$	$Im\ Y_{N_y-1,2}$	...	$Re\ Y_{N_y-1,\frac{N_x}{2}-1}$	$Im\ Y_{N_y-1,\frac{N_x}{2}-1}$	$Re\ Y_{N_y-1,\frac{N_x}{2}}$

It is worth taking a moment to look closely at the indices of these arrays. Certain values in the array are guaranteed to be 0 (more accurately, certain values of f_k are guaranteed to be real). It should also be noted that the last row listed in the table will be present only if N_y is even and that the last column will be present only if N_x is even. In the case of the two-dimensional array being treated as N_y separate one-dimensional arrays rather than a full two-dimensional transform (we'll take a look at how to do this), all of the result rows will be analogous to the single row listed for the output of the one-dimensional array.

The third argument, called `flags`, indicates exactly what operation is to be done. As usual, `flags` is treated as a bit array, so you can combine any flags you need with Boolean OR. The transformation we started with is known as a *forward transform* and is selected by default. The inverse transform[16] is defined in exactly the same way except for a change of sign in the exponential and a scale factor. To perform the inverse transform without the scale factor, use the flag `cv::DFT_INVERSE`. The flag for the scale factor is `cv::DFT_SCALE`, which results in all of the output being scaled by a factor of N^{-1} (or $\left(N_x N_y\right)^{-1}$ for a two-dimensional transform). This scaling is necessary if the sequential application of the forward transform and the inverse transform is to bring us back to where we started. Because one often wants to combine `cv::DFT_INVERSE` with `cv::DFT_SCALE`, there are several shorthand notations for this kind of operation. In addition to just combining the two operations, you can use `cv::DFT_INV_SCALE` (or `cv::DFT_INVERSE_SCALE` if you're not into that brevity thing). The last flag you may want to have handy is `cv::DFT_ROWS`, which allows you to tell `cv::dft()` to treat a two-dimensional array as a collection of one-dimensional arrays that should each be transformed separately as if they were N_y distinct vectors of length N_x. This can significantly reduce overhead when doing many transformations at a time. By using `cv::DFT_ROWS` it is also possible to implement three-dimensional (and higher) DFT.

Though the default behavior of the forward transform is to produce results in CCS format (which results in an output array exactly the same size as the input array), you can explicitly ask OpenCV to not do this with the flag `cv::DFT_COMPLEX_OUTPUT`. The result will be the full complex array (with all of the zeros in it). Conversely, when performing an inverse transformation on a complex array, the result is normally also a complex array. If the source array had complex conjugate symmetry,[17] you can ask OpenCV to produce a purely real array (which will be smaller than the input array) by passing the `cv::DFT_REAL_OUTPUT` flag.

In order to understand the last argument, `nonzero_rows`, we must digress for a moment to explain that in general, DFT algorithms strongly prefer input vectors of some lengths over input vectors of other lengths; similarly for arrays of some sizes over arrays of other sizes. In most DFT algorithms, the preferred sizes are powers of 2 (i.e., 2^n for some integer n). In the case of the algorithm used by OpenCV, the preference is that the vector lengths, or array dimensions, be $2^p 3^q 5^r$, for some integers p, q, and r. Hence the usual procedure is to create a somewhat larger array and then to copy your array into that somewhat roomier zero-padded array. For convenience, there is a handy utility function, `cv::getOptimalDFTSize()`, which takes the (integer) length of your vector and returns the first equal or larger size that can be expressed in the form given (i.e., $2^p 3^q 5^r$). Despite the need for this padding, it is possible to indicate to `cv::dft()` that you really do not care about the transform of those rows that you had to add down below your actual data (or, if you are doing an inverse transform, which

[16] With the inverse transform, the input is packed in the special format described previously. This makes sense because, if we first called the forward DFT and then ran the inverse DFT on the results, we would expect to wind up with the original data—that is, of course, if we remember to use the `cv::DFT_SCALE` flag!

[17] This is not to say that it is in CCS format, only that it possesses the symmetry, as it would if (for example) it was the result of a forward transform of a purely real array in the first place. Also, take note that you are *telling* OpenCV that the input array has this symmetry—it will trust you. It does not actually check to verify that this symmetry is present.

rows in the result you do not care about). In either case, you can use `nonzero_rows` to indicate how many rows contain meaningful data. This will provide some savings in computation time.

`cv::idft()`, the Inverse Discrete Fourier Transform

As we saw earlier, the function `cv::dft()` can be made to implement not only the discrete Fourier transform, but also the inverse operation (with the provision of the correct `flags` argument). It is often preferable, if only for code readability, to have a separate function that does this inverse operation by default.

```
void cv::idft(
  cv::InputArray  src,                          // Input array (real or complex)
  cv::OutputArray dst,                          // Output array
  int             flags       = 0,              // for variations
  int             nonzeroRows = 0               // number of rows to not ignore
);
```

Calling `cv::idft()` is exactly equivalent to calling `cv::dft()` with the `cv::DFT_INVERSE` flag (in addition to whatever flags you supply to `cv::idft()`, of course.)

`cv::mulSpectrums()`, Spectrum Multiplication

In many applications that involve computing DFTs, one must also compute the per-element multiplication of the two resulting spectra. Because such results are complex numbers, typically packed in their special high-density CCS format, it would be tedious to unpack them and handle the multiplication via the "usual" matrix operations. Fortunately, OpenCV provides the handy `cv::mulSpectrums()` routine, which performs exactly this function for us:

```
void cv::mulSpectrums(
  cv::InputArray  src1,                         // First input array (ccs or complex)
  cv::InputArray  src2,                         // Second input array (ccs or complex)
  cv::OutputArray dst,                          // Result array
  int             flags,                        // for row-by-row computation
  bool            conj = false                  // true to conjugate src2
);
```

Note that the first two arguments are arrays, which may be either CCS packed single-channel spectra or two-channel complex spectra (as you would get from calls to `cv::dft()`). The third argument is the destination array, which will be of the same size and type as the source arrays. The final argument, `conj`, tells `cv::mulSpectrums()` exactly what you want done. In particular, it may be set to `false` for implementing the above pair multiplication or set to `true` if the element from the first array is to be multiplied by the complex conjugate of the corresponding element of the second array[18].

Convolution Using Discrete Fourier Transforms

It is possible to greatly increase the speed of a convolution by using DFT via the convolution theorem [Titchmarsh26] that relates convolution in the spatial domain to multiplication in the Fourier domain [Morse53; Bracewell65; Arfken85].[19] To accomplish this, one first computes the Fourier transform of the image and then the Fourier transform of the convolution filter. Once this is done, the convolution can be performed in the transform space in linear time with respect to the number of pixels in the image. It is worthwhile to look at the source code for computing such a convolution, as it will also provide us with many good examples of using `cv::dft()`. The code is shown in Example 6-4, which is taken directly from the OpenCV reference.

---

[18] The primary usage of this argument is the implementation of a correlation in Fourier space. It turns out that the only difference between convolution (which we will discuss in the next section), and correlation, is the conjugation of the second array in the spectrum multiplication.

[19] Recall that OpenCV's DFT algorithm implements the FFT whenever the data size makes the FFT faster.

Example 6-4: Use of `cv::dft()` and `cv::idft()` to accelerate the computation of convolutions

```
#include <opencv2/opencv.hpp>
#include <iostream>

using namespace std;

int main(int argc, char** argv) {

  if(argc != 2) {
    cout << "Fourier Transform\nUsage: ch6_ex6_5 <imagename>" << endl;
    return -1;
  }

  cv::Mat A = cv::imread(argv[1],0);

  if( A.empty() ) { cout << "Can not load " << argv[1] << endl; return -1; }

  cv::Size patchSize( 100, 100 );
  cv::Point topleft( A.cols/2, A.rows/2 );
  cv::Rect roi( topleft.x, topleft.y, patchSize.width, patchSize.height );
  cv::Mat B = A( roi );

  int dft_M = cv::getOptimalDFTSize( A.rows+B.rows-1 );
  int dft_N = cv::getOptimalDFTSize( A.cols+B.cols-1 );

  cv::Mat dft_A = cv::Mat::zeros( dft_M, dft_N, cv::F32 );
  cv::Mat dft_B = cv::Mat::zeros( dft_M, dft_N, cv::F32 );

  cv::Mat dft_A_part = dft_A( Rect(0, 0, A.cols,A.rows) );
  cv::Mat dft_B_part = dft_B( Rect(0, 0, B.cols,B.rows) );

  A.convertTo( dft_A_part, dft_A_part.type(), 1, -mean(A)[0] );
  B.convertTo( dft_B_part, dft_B_part.type(), 1, -mean(B)[0] );

  cv::dft( dft_A, dft_A, 0, A.rows );
  cv::dft( dft_B, dft_B, 0, B.rows );

  // set the last parameter to false to compute convolution instead of correlation
  cv::mulSpectrums( dft_A, dft_B, dft_A, 0, true );
  cv::idft( dft_A, dft_A, DFT_SCALE, A.rows + B.rows - 1 );

  cv::Mat corr = dft_A( Rect(0, 0, A.cols + B.cols - 1, A.rows + B.rows - 1) );
  cv::normalize( corr, corr, 0, 1, NORM_MINMAX, corr.type() );
  cv::pow( corr, 3., corr );

  cv::B ^= cv::Scalar::all( 255 );

  cv::imshow( "Image", A );
  cv::imshow( "Correlation", corr );
  cv::waitKey();
  return 0;
}
```

In Example 6-4, we can see that the input arrays are first created and then initialized. Next, two new arrays are created whose dimensions are optimal for the DFT algorithm. The original arrays are copied into these new arrays and the transforms are then computed. Finally, the spectra are multiplied together and the inverse transform is applied to the product. The transforms are the slowest[20] part of this operation; an N-by-

[20] By "slowest" we mean "asymptotically slowest"—in other words, that this portion of the algorithm takes the most time for very large N. This is an important distinction. In practice, as we saw in the earlier section on convolutions, it is

N image takes $O(N^2 \log N)$ time and so the entire process is also completed in that time (assuming that $N > M$ for an M-by-M convolution kernel). This time is much faster than $O(N^2M^2)$, the non-DFT convolution time required by the more naïve method.

`cv::dct()`, the Discrete Cosine Transform

For real-valued data, it is often sufficient to compute what is, in effect, only half of the discrete Fourier transform. The *discrete cosine transform* (DCT) [Ahmed74; Jain77] is defined analogously to the full DFT by the following formula:

$$c_k = \left(\frac{1}{N}\right)^{1/2} x_0 + \sum_{n=1}^{N-1} \left(\frac{2}{N}\right)^{1/2} x_n \cos\left((k + \frac{1}{2})\frac{n}{N}\pi\right)$$

Of course, there is a similar transform for higher dimensions. Note that, by convention, the normalization factor is applied to both the cosine transform and its inverse (which is not the convention for the discrete Fourier transform).

The basic ideas of the DFT apply also to the DCT, but now all the coefficients are real-valued. Astute readers might object that the cosine transform is being applied to a vector that is not a manifestly even function. However, with `cv::dct()`, the algorithm simply treats the vector as if it were extended to negative indices in a mirrored manner.

The actual OpenCV call is:

```
void cv::dct(
  cv::InputArray  src,                          // Input array (even size)
  cv::OutputArray dst,                          // Output array
  int             flags       = 0               // for row-by-row or inverse
);
```

The `cv::dct()` function expects arguments like those for `cv::dft()` except that, because the results are real-valued, there is no need for any special packing of the result array (or of the input array in the case of an inverse transform). Unlike `cv::dft()`, however, the input array must have an even number of elements (you can pad the last element with a zero if necessary to achieve this). The `flags` argument can be set to `cv::DCT_INVERSE` to generate the inverse transformation, and either may be combined with `cv::DCT_ROWS` with the same effect as with `cv::dft()`. Because of the different normalization convention, both the forward and inverse cosine transforms always contain their respective contribution to the overall normalization of the transform; hence there is no analog to `cv::DFT_SCALE` for `cv::dct()`.

As with `cv::dft()`, there is a strong dependence of performance on array size. In fact, deep down, the implementation of `cv::dct()` actually calls `cv::dft()` on an array exactly half the size of your input array. For this reason, the optimal size of an array to pass to `cv::dct()` is exactly double the size of the optimal array you would pass to `cv::dft()`. Putting everything together, the best way to get an optimal size for `cv::dct()` is to compute:

```
size_t optimal_dct_size = 2 * cv::getOptimalDFTSize( (N+1)/2 );
```

where N is the actual size of your data that you want to transform.

`cv::idct()`, the Inverse Discrete Cosine Transform

Just as with `cv::idft()` and `cv::dft()`, `cv::dct()` can be asked to compute the inverse cosine transform using the `flags` argument. As before, code readability is often improved with the use of a separate function that does this inverse operation by default.

```
void cv::idct(
  cv::InputArray  src,                          // Input array
```

---

not always optimal to pay the overhead for conversion to Fourier space. In general, when convolving with a small kernel it will not be worth the trouble to make this transformation.

```
    cv::OutputArray dst,                        // Output array
    int             flags = 0,                  // for row-by-row computation
);
```

Calling `cv::idct()` is exactly equivalent to calling `cv::dct()` with the `cv::DCT_INVERSE` flag (in addition to any other flags you supply to `cv::idct()`).

Integral Images

OpenCV allows you to calculate an integral image easily with the appropriately named `cv::integral()` function. An *integral image* [Viola04] is a data structure that allows rapid summing of sub-regions[21]. Such summations are useful in many applications; a notable one is the computation of *Haar wavelets*, which are used in face detection and similar algorithms.

There are three variations of the integral image that are supported by OpenCV. They are the *sum*, the *square-sum*, and the *tilted-sum*.

A standard integral image sum has the form:

$$sum(x,y) = \sum_{y'<y} \sum_{x'<x} image(x',y')$$

The square-sum image is the sum of squares:

$$sum_{square}(x,y) = \sum_{y'<y} \sum_{x'<x} [image(x',y')]^2$$

The tilted-sum is like the sum except that it is for the image rotated by 45 degrees:

$$sum_{tilted}(x,y) = \sum_{y'<y} \sum_{abs(x'-x)<y} image(x',y')$$

Using these integral images, one may calculate sums, means, and standard deviations over arbitrary upright or "tilted" rectangular regions of the image. As a simple example, to sum over a simple rectangular region described by the corner points (x_1, y_1) and (x_2, y_2), where $x_2 > x_1$ and $y_2 > y_1$, we'd compute:

$$\sum_{y_1 \le y < y_2} \sum_{x_1 \le x < x_2} image(x,y) = [sum(x_2,y_2) - sum(x_1,y_2) - sum(x_2,y_1) + sum(x_1,y_1)]$$

In this way, it is possible to do fast blurring, approximate gradients, compute means and standard deviations, and perform fast block correlations even for variable window sizes.

[21] The citation above is the best for more details on the method, but it was actually introduced in computer vision in 2001 in a paper *Robust Real-time Object Detection* by the same authors. The method was previously used as early as 1984 in computer graphics, where the integral image is known as a *Summed Area Table*.

To make this all a little more clear, consider the 7-by-5 image shown in

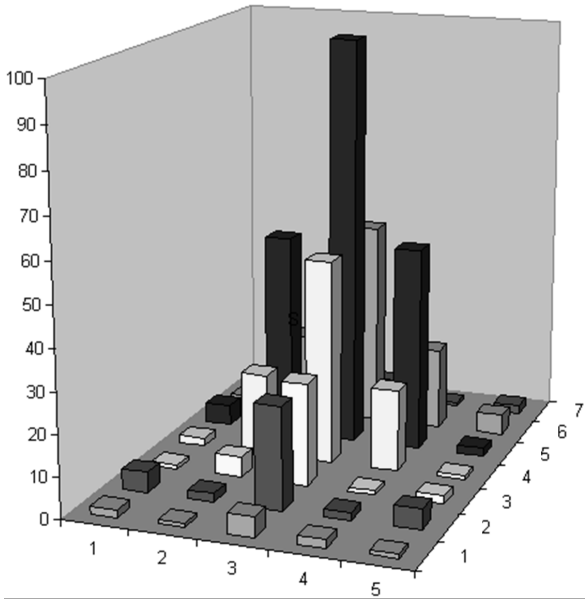


Figure 6-8; the region is shown as a bar chart in which the height associated with the pixels represents the brightness of those pixel values. The same information is shown in Figure 6-9, numerically on the left and in integral form on the right. Integral images I' are computed by going across rows, proceeding row by row using the previously computed integral image values together with the current raw image I pixel value $I(x, y)$ to calculate the next integral image value as follows:

$$I'(x, y) = [I(x, y) - I(x - 1, y) - I(x, y - 1) + I(x - 1, y - 1)]$$

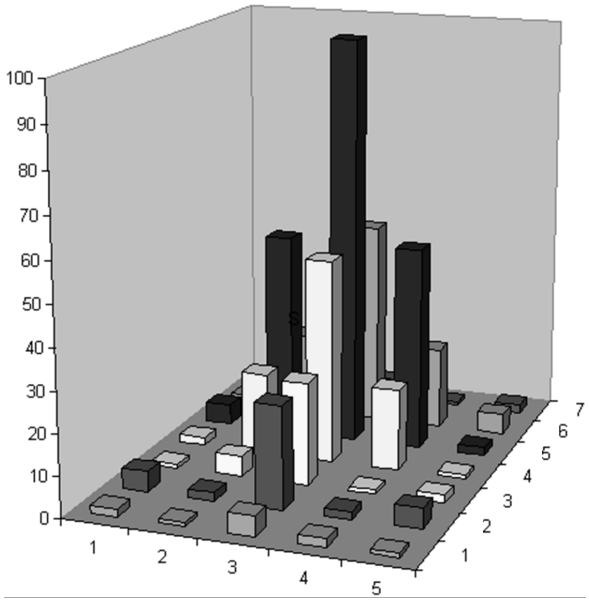


Figure 6-8: Simple 7-by-5 image shown as a bar chart with x, y, and height equal to pixel value

The last term is subtracted off because this value is double-counted when adding the second and third terms. You can verify that this works by testing some values in Figure 6-9.

When using the integral image to compute a region, we can see by Figure 6-9 that, in order to compute the central rectangular area bounded by the 20s in the original image, we'd calculate $398 - 9 - 10 + 1 = 380$. Thus, a rectangle of any size can be computed using four measurements (resulting in $O(1)$ computational complexity).

1	2	5	1	2
2	**20**	**50**	**20**	5
5	**50**	**100**	**50**	2
2	**20**	**50**	**20**	1
1	5	25	1	2
5	2	25	2	5
2	1	5	2	1

0	0	0	0	0	0
0	1	3	8	9	11
0	3	**25**	**80**	**101**	108
0	8	**80**	**235**	**306**	315
0	10	**102**	**307**	**398**	408
0	11	108	338	430	442
0	16	115	370	464	481
0	18	118	378	474	492

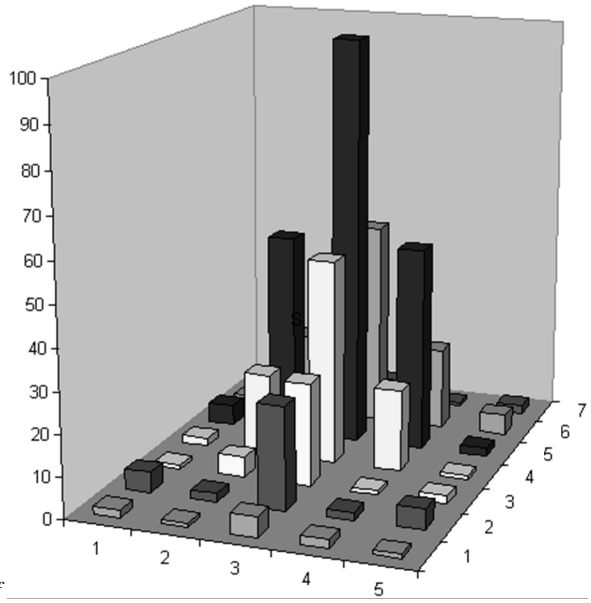


Figure 6-9: The 7-by-5 image of _____

Figure 6-8 shown numerically at left (with the origin assumed to be the upper-left) and converted to an (8-by-6) integral image at right

`cv::integral()`, for Standard Summation Integral

The different forms of integral are (somewhat confusingly) distinguished in the C++ API only by their arguments. The form that computes the basic sum has only three.

```
void cv::integral(
  cv::InputArray  image,                    // Input array
  cv::OutputArray sum,                      // Output sum results
  int             sdepth = -1               // Depth for results (e.g., cv::F32)
);
```

The first and second are the input and result images. If the input image is of size W-by-H, then the output image will be of size $(W+1)$-by-$(H+1)$.[22] The third argument `sdepth` specifies the desired depth of the sum (destination) image. `sdepth` can be any of `cv::S32`, `cv::F32`, or `cv::F64`.[23]

[22] This allows for the rows of zeros which are implied by the fact that summing zero terms results in a sum of zero.

[23] It is worth noting that even though `sum` and `tilted_sum` allow 32-bit float as output for input images of 32-bit float type, it is recommended to use 64-bit float, particularly for larger images. After all, a modern large image can be many millions of pixels.

`cv::integral()`, for Squared Summation Integral

The squared sum is computed with the same function as the regular sum, except that the provision of an additional output argument for the squared sum.

```
void cv::integral(
  cv::InputArray  image,                          // Input array
  cv::OutputArray sum,                            // Output sum results
  cv::OutputArray sqsum,                          // Output sum of squares results
  int             sdepth = -1                     // Depth for results (e.g., cv::F32)
);
```

The `cv::OutputArray` argument `sqsum` indicates to `cv::integral()` that the square sum should be computed in addition to the regular sum. As before, `sdepth` specifies the desired depth of the resulting images. `sdepth` can be any of `cv::S32`, `cv::F32`, or `cv::F64`.

`cv::integral()`, for Standard Summation Integral

Similar to the squared sum, the tilted sum integral is essentially the same function, with an additional argument for the additional result.

```
void cv::integral(
  cv::InputArray  image,                          // Input array
  cv::OutputArray sum,                            // Output sum results
  cv::OutputArray sqsum,                          // Output sum of squares results
  cv::OutputArray tilted,                         // Output tilted sum results
  int             sdepth = -1                     // Depth for results (e.g., cv::F32)
);
```

The additional `cv::OutputArray` argument `tilted` is computed by this form of `cv::integral()`, in addition to the other sums, thus all of the other arguments are the same.

The Canny Edge Detector

Though it is possible to expose edges in images with simple filters such as the Laplace filter, it is possible to improve on this method substantially. The simple Laplace filter method was refined by J. Canny in 1986 into what is now commonly called the *Canny edge detector* [Canny86]. One of the differences between the Canny algorithm and the simpler, Laplace-based algorithm in the previous chapter is that, in the Canny algorithm, the first derivatives are computed in x and y and then combined into four directional derivatives. The points where these directional derivatives are local maxima are then candidates for assembling into edges.

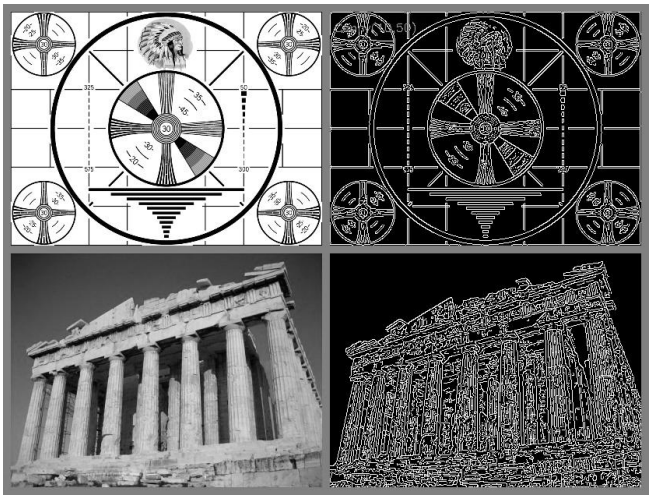


Figure 6-10: Results of Canny edge detection for two different images when the high and low thresholds are set to `50` and `10`, respectively.

However, the most significant new dimension to the Canny algorithm is that it tries to assemble the individual edge candidate pixels into *contours*.[24] These contours are formed by applying a *hysteresis threshold* to the pixels. This means that there are two thresholds, an upper and a lower. If a pixel has a gradient larger than the upper threshold, then it is accepted as an edge pixel; if a pixel is below the lower threshold, it is rejected. If the pixel's gradient is between the thresholds, then it will be accepted only if it is connected to a pixel that is above the high threshold. Canny recommended a ratio of high:low threshold between 2:1 and 3:1.

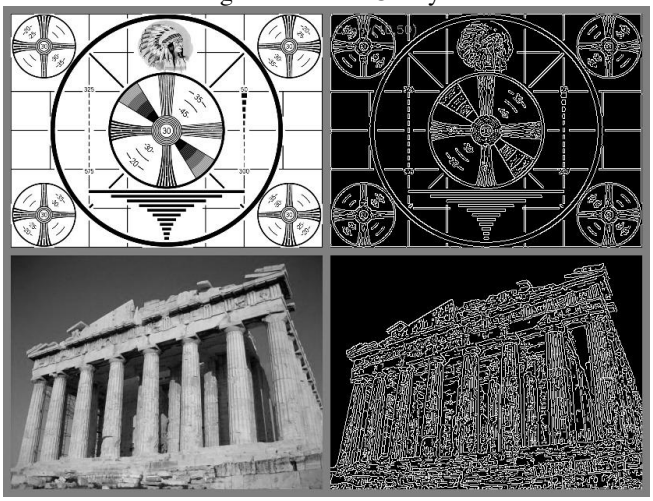
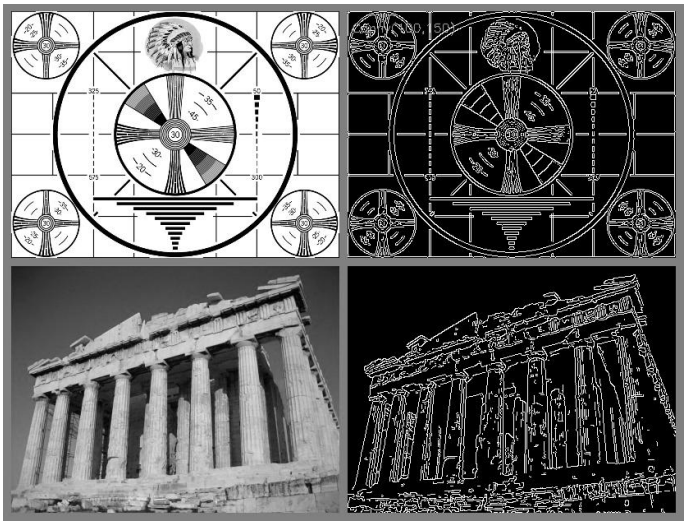


Figure 6-10 and Figure 6-11 show the results of applying `cv::Canny()` to a test pattern and a photograph using high:low hysteresis threshold ratios of 5:1 and 3:2, respectively.



[24] We'll have much more to say about contours later. As you await those revelations, keep in mind that the `cv::Canny()` routine does not actually return objects of a contour type; we will have to build those from the output of `cv::Canny()` if we want them by using `cv::findContours()`. Everything you ever wanted to know about contours will be covered in Chapter 8.

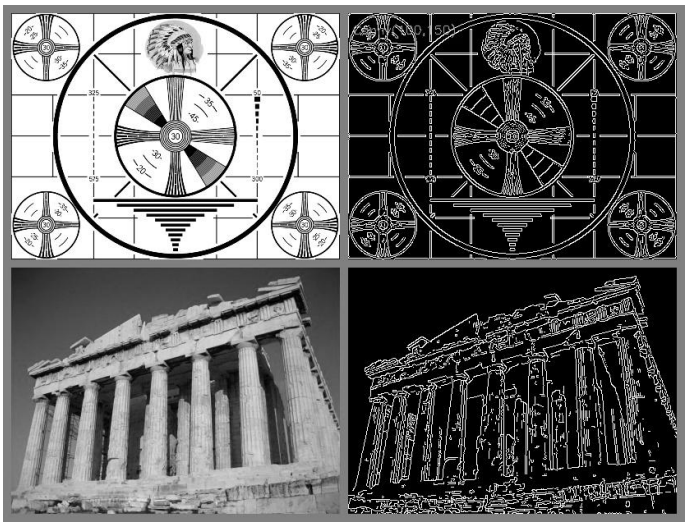


Figure 6-11: Results of Canny edge detection for two different images when the high and low thresholds are set to `150` and `100`, respectively

`cv::Canny()`

The OpenCV implementation of the Canny edge detection algorithm converts an input image into an "edge image".

```
void cv::Canny(
  cv::InputArray  image,                      // Input single channel image
  cv::OutputArray edges,                      // Output edge image
  double          threshold1,                 // "lower" threshold
  double          threshold2,                 // "upper" threshold
  int             apertureSize = 3,           // Sobel aperture
  bool            L2gradient   = false        // true for more accurate L2-norm
);
```

The `cv::Canny()` function expects an input image, which must be single-channel, and an output image, which will also be grayscale (but which will actually be a Boolean image). The next two arguments are the low and high thresholds. The next to last argument `apertureSize` is the aperture used by the Sobel derivative operators that are called inside of the implementation of `cv::Canny()`. The final argument `L2gradient` is used to select between computing the gradient "correctly" using the proper L_2-norm, or if a faster less accurate L_1-norm based method should be used. If the argument `L2gradient` is set to true, the more accurate form is used:

$$|grad(x,y)|_{L_2} = \sqrt{\left(\frac{dI}{dx}\right)^2 + \left(\frac{dI}{dy}\right)^2}$$

If `L2gradient` is set to `false`, the faster form is used:

$$|grad(x,y)|_{L_1} = \left|\frac{dI}{dx}\right| + \left|\frac{dI}{dy}\right|$$

Line Segment Detection

In a great number of practical image processing applications, it is useful to be able to find either the dominant line segments, or all of the lines in an image. In particular, straight lines often provide valuable clues about perspective and the structure of objects. OpenCV provides several different algorithms which address this problem in different ways, each with their own distinct strengths and weaknesses.

The *Hough transform*[25] is a method for finding lines, circles, or other simple forms in an image. The original Hough transform was a line transform, which is a relatively fast way of searching a binary image for straight lines. The transform can be further generalized to cases other than just simple lines (we will return to this in the next section).

In addition to the *Hough Line Transform*, another more recent algorithm called *LSD* (for *Line Segment Detector*) provides a fast technique which is, in general, more robust than the Hough Line Transform.

Hough Line Transform

The basic theory of the Hough line transform is that any point in a binary image could be part of some set of possible lines. If we parameterize each line by, for example, a slope a and an intercept b, then a point in the original image is transformed to a locus of points in the (a, b) plane corresponding to all of the lines passing through that point (see

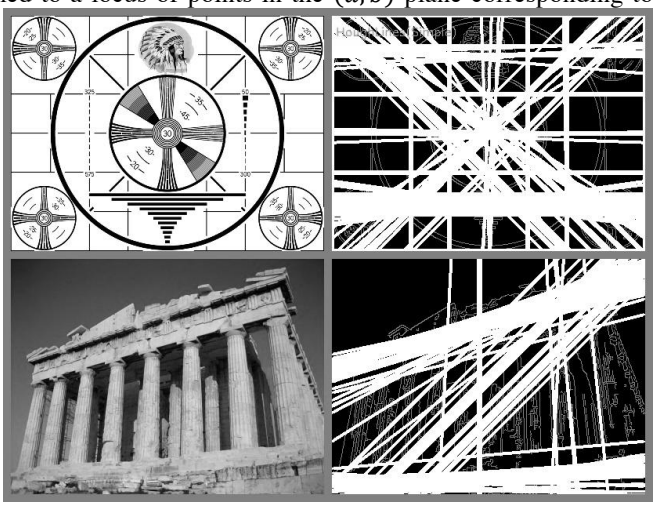
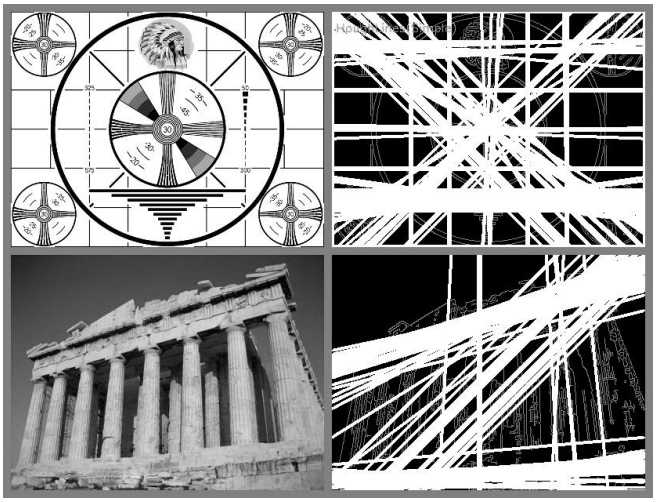


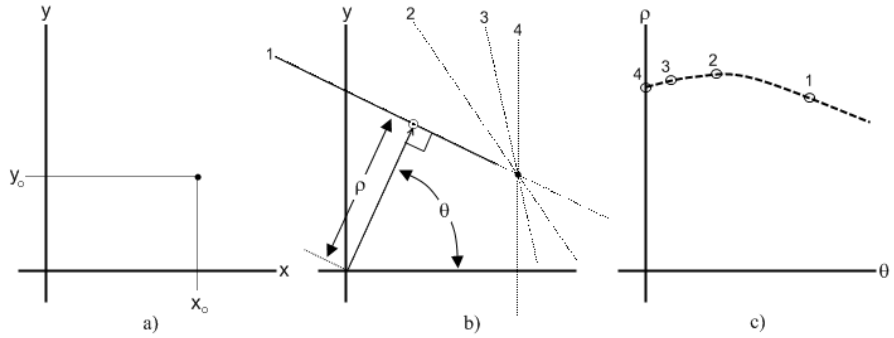
Figure 6-12). If we convert every nonzero pixel in the input image into such a set of points in the output image and sum over all such contributions, then lines that appear in the input (i.e., (x, y) plane) image will appear as local maxima in the output (i.e., (a, b) plane) image. Because we are summing the contributions from each point, the (a, b) plane is commonly called the *accumulator plane*.



[25] Hough developed the transform for use in physics experiments [Hough59]; its use in vision was introduced by Duda and Hart [Duda72].

Figure 6-12: The Hough line transform finds many lines in each image; some of the lines found are expected, but others may not be.

It might occur to you that the slope-intercept form is not really the best way to represent all of the lines passing through a point (because of the considerably different density of lines as a function of the slope, and the related fact that the interval of possible slopes goes from $-\infty$ to $+\infty$). It is for this reason that the actual parameterization of the transform image used in numerical computation is somewhat different. The preferred parameterization represents each line as a point in polar coordinates (ρ, θ), with the implied line being the line passing through the indicated point but perpendicular to the radial from the origin to that



point (see

Figure 6-13). The equation for such a line is:

$$\rho = x \cos\theta + y \sin\theta$$

The OpenCV Hough transform algorithm does not make this computation explicit to the user. Instead, it simply returns the local maxima in the (ρ, θ) plane. However, you will need to understand this process in order to understand the arguments to the OpenCV Hough line transform function.

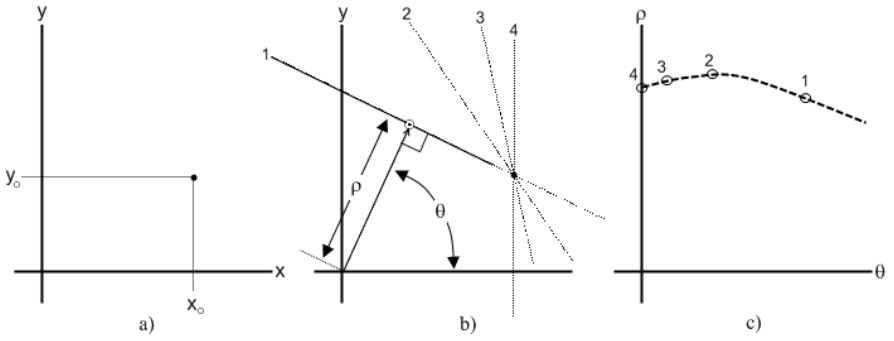


Figure 6-13: A point (x_0, y_0) in the image plane (a) implies many lines each parameterized by a different ρ and θ (b); these lines in the (ρ, θ) plane, which taken together form a curve of characteristic shape (c).

OpenCV supports three different kinds of Hough line transform: the *standard Hough transform* (SHT) [Duda72], the *multiscale Hough transform* (MHT), and the *progressive probabilistic Hough transform* (PPHT).[26] The SHT is the algorithm we just looked at. The MHT algorithm is a slight refinement that gives

[26] The "probabilistic Hough transform" (PHT) was introduced by Kiryati, Eldar, and Bruckshtein in 1991 [Kiryati91]; the PPHT was introduced by Matas, Galambosy, and Kittler in 1999 [Matas99].

more accurate values for the matched lines. The PPHT is a variation of this algorithm that, among other things, computes an extent for individual lines in addition to the orientation (as shown in

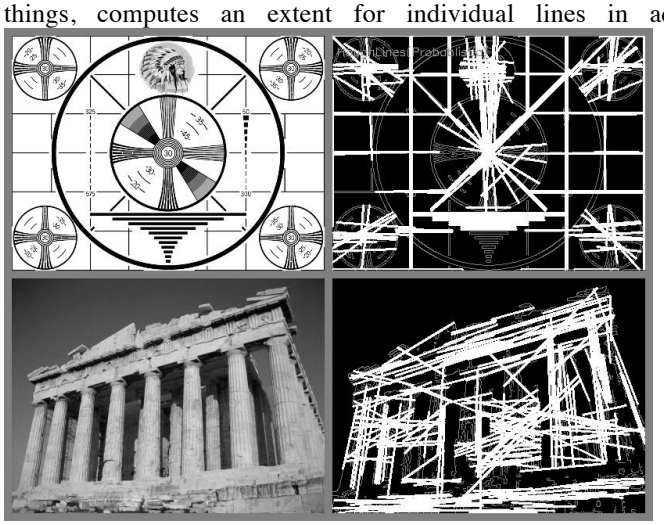


Figure 6-14). It is "probabilistic" because, rather than accumulating every possible point in the accumulator plane, it accumulates only a fraction of them. The idea is that if the peak is going to be high enough anyhow, then hitting it only a fraction of the time will be enough to find it; the result of this conjecture can be a substantial reduction in computation time.

`cv::HoughLines()`, the Standard and Multi-scale Hough Transforms

The standard and multi-scale Hough transforms are both implemented in a single function: `cv::HoughLines()` – with the distinction being in the use (or nonuse) of two optional parameters.

```
void cv::HoughLines(
  cv::InputArray  image,                    // Input single channel image
  cv::OutputArray lines,                    // N-by-1 2-channel array
  double          rho,                      // rho resolution (pixels)
  double          theta,                    // theta resolution (radians)
  int             threshold,                // Unnormalized accumulator threshold
  double          srn      = 0,             // rho refinement (for MHT)
  double          stn      = 0              // theta refinement (for MHT)
);
```

The first argument is the input image. It must be an 8-bit image, but the input is treated as binary information (i.e., all nonzero pixels are considered to be equivalent). The second argument is the place where the found lines will be stored. It will be an N-by-1 two-channel array of floating point type (the number of columns N, will be the number of lines returned).[27] The two channels will contain the ρ and θ values for each found line.

The next two arguments, `rho` and `theta`, set the resolution desired for the lines (i.e., the resolution of the accumulator plane). The units of `rho` are pixels and the units of `theta` are radians; thus, the accumulator plane can be thought of as a two-dimensional histogram with cells of dimension `rho` pixels by `theta` radians. The `threshold` value is the value in the accumulator plane that must be reached for the algorithm to report a line. This last argument is a bit tricky in practice; it is not normalized, so you should expect to scale it up with the image size for SHT. Remember that this argument is, in effect, indicating the number of points (in the edge image) that must support the line for the line to be returned.

[27] As usual, depending on the object type you pass to `lines`, this could be either a 1-by-N array with two channels, or if you like, a `std::vector<>` with N entries, with each of those entries being of type `Vec2f`.

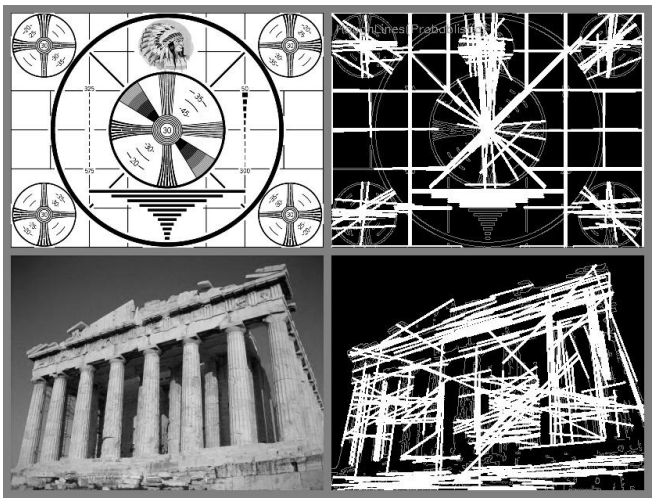


Figure 6-14: The Canny edge detector (`param1=50, param2=150`) is run first, with the results shown in gray, and the progressive probabilistic Hough transform (`param1=50, param2=10`) is run next, with the results overlaid in white; you can see that the strong lines are generally picked up by the Hough transform

The parameters `srn` and `stn` are not used by the standard Hough transform; they control an extension of the SHT algorithm called the multi-scale Hough transform (MHT). For MHT, these two parameters indicate higher resolutions to which the parameters for the lines should be computed. MHT first computes the locations of the lines to the accuracy given by the `rho` and `theta` parameters and then goes on to refine those results by a factor of `srn` and `stn`, respectively (i.e., the final resolution in `rho` is `rho` divided by `srn` and the final resolution in `theta` is `theta` divided by `stn`). Leaving these parameters set to zero causes the SHT algorithm to be run.

`cv::HoughLinesP()`, the Progressive Probabilistic Hough Transform

```
void cv::HoughLinesP(
  cv::InputArray  image,                        // Input single channel image
  cv::OutputArray lines,                        // N-by-1 4-channel array
  double          rho,                          // rho resolution (pixels)
  double          theta,                        // theta resolution (radians)
  int             threshold,                    // Un-normalized accumulator threshold
  double          minLineLength = 0,            // required line length
  double          maxLineGap    = 0             // required line separation
);
```

The `cv::HoughLinesP()` function works very much like `cv::HoughLines()`, with two important differences. The first is that the lines argument will be a four-channel array (or a vector of objects all of type `Vec4i`). The four channels will be the (x_0, y_0) and (x_1, y_1) (in that order), the (x, y) locations of the two endpoints of the found line segment. The second important difference is the meaning of the two parameters. For the PPHT, the `minLineLength` and `maxLineGap` arguments set the minimum length of a line segment that will be returned, and the separation between collinear segments required for the algorithm not to join them into a single longer segment.

Line Segment Detection (LSD): cv::LineSegmentDetector

The LSD algorithm, originally developed by Rafael Grompone von Gioi, Jérémie Jakubowicz, Jean-Michel Morel, and Gregory Randall [vonGioi10][vonGioi12], is a linear time line segment finding algorithm. The LSD algorithm works by first analyzing the image in a local way and determining what kind of line might be supported by the pixels in these small patches. The patches are then agglomerated into larger hypotheses which are then validated before being accepted.

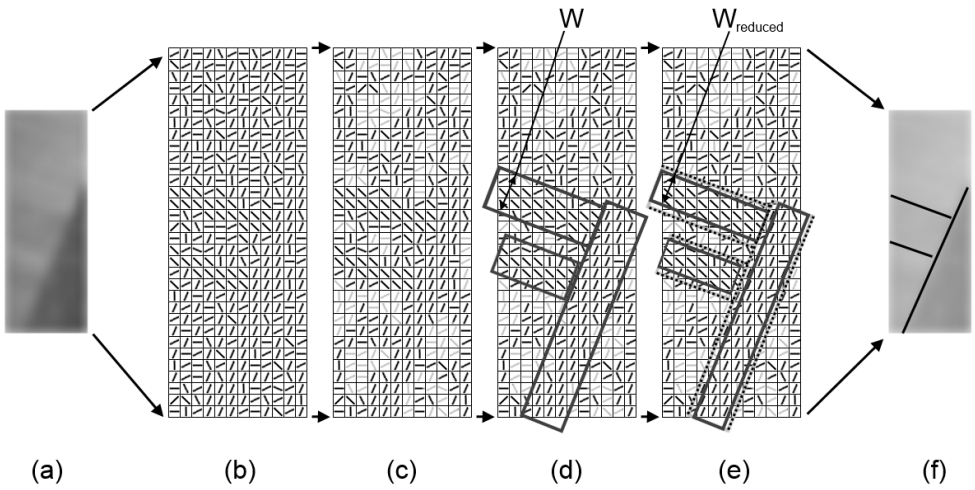


Figure 6-15: Beginning with an image (a), the LSD algorithm first computes the local level line field (b). Level lines associated with weak gradients are rejected (c). Line support regions are associated with rectangles (d) whose width W is the approximate width of the line. In some cases, the widths are reduced to produce lines with lower NFA scores (e). The final resulting lines are shown here on top of the original image (f).

The local procedure is based on what is called a *level line* (or, collectively across the entire image, the *level line field* Figure 6-15b). The level line field is computed by first computing the gradient of the image at every point, the level lines are perpendicular to those local gradients.

Once the level line field has been computed, connected regions can be generated which contain level lines of the same or similar orientation (i.e. within some tolerance τ). This is a very powerful idea, because it does not require the level lines to be associated with equally strong gradients[28]. In this way, aliasing effects can be naturally accounted for, as even in an aliased line, the orientation of the local level lines will be approximately parallel.

Collectively, the regions containing similar level lines are called *line support regions*. A rectangle is fit to each such line support region; this rectangle represents a hypothesis for a line Figure 6-15d.

The LSD algorithm does not double count individual level lines. To avoid doing so, the level lines are first sorted based on the magnitudes of their associated gradients, on the assumption that the strongest gradients go with the most important lines. The line support regions are then constructed by building out from those strong gradient points in descending order of their strength. Because an exact ordering is not necessary for this process, the gradient magnitudes are instead binned into N_{bins} different bins in between zero and the largest observed gradient. The advantage of this *pseudo-ordering* is that it takes $O(N)$ time, rather than $O(N \log N)$ time (as would be expected of a full sorting).

Once the rectangle hypotheses have been formed, each can be tested; if a large number of the points contained are consistent with the hypothesis, then a line segment has probably been found. If, on the other hand, this bounding rectangle contains many points which are not part of the original line support region, the proposed segment is probably not real.

The actual test is based on what is called the *number of false alarms* (NFA) for a particular rectangle. The NFA is a function of the total number of pixels in the rectangle n as well as the number of aligned pixels in that rectangle k. Conceptually, the NFA is the number of rectangles we would expect to find in a random

[28] It is however, common practice to exclude level lines whose associated gradient is below some threshold (as shown in Figure 6-15c).

image with similar significance to the hypothesis rectangle (i.e. similar n and k)[29]. Rectangle hypotheses are rejected if their NFA is below some threshold: $NFA_{I,r}(n,k) < \epsilon$. Optionally, a refinement pass can attempt to tune the rectangle hypotheses to minimize their NFA Figure 6-15e.

Using the `cv::lineSegmentDetector` Object

The LSD algorithm is implemented in OpenCV using the more contemporary style in which features of the library are implemented as objects which hold their various parameters and thereafter provide an algorithmic service by means of their member functions. The object which implements the LSD algorithm is called `cv::lineSegmentDetector`. The following function creates one of these objects and returns you a `cv::Ptr<>` to it:

```
cv::Ptr<LineSegmentDetector> cv::createLineSegmentDetector(
  int    refine      = cv::LSD_REFINE_STD,
  double scale       = 0.8,
  double sigma_scale = 0.6,
  double quant       = 2.0,
  double ang_th      = 22.5,
  double log_eps     = 0,
  double density_th  = 0.7,
  int    n_bins      = 1024
);
```

You should first notice that every argument to this function has a default value. This is not an accident. These values were determined by the designers of the LSD algorithm to be broadly optimal. As a result, you are unlikely to want to change them. For experts (and the experimentally minded) however, we describe them here.

The first argument is `refine`, which controls how the algorithm will attempt to refine (though not necessarily reduce) its initial findings to the best final output. The three refinement options are to do nothing (`cv::LSD_REFINE_NONE`), to attempt to break long arcs into multiple shorted but straighter segments (`cv::LSD_REFINE_STD`), and to both do arc decomposition as well as automatic parameter tuning, which should give better results at the expense of longer runtime (`cv::LSD_REFINE_ADV`).

The second argument is the `scale` argument. Any incoming image to the LSD algorithm is automatically rescaled in both dimensions by this factor. The utility of this is that, just as with human perception, lines which are apparent at large scales will often be invisible when the image is viewed at high resolution; similarly, at reduced resolution, lines in fine details will be lost.

As part of the reduction produced by `scale`, the image is first convolved with a Gaussian kernel. The size of this kernel is specified by `sigma_scale`; the size in pixels is actually the ratio of `sigma_scale` divided by `scale`. The default `sigma` value of `0.8` along with the default value for `sigma_scale` of `0.6` are chosen to give good results even in images with substantial aliasing effects on lines, without substantially altering the return from what would be expected at full scale.

The argument `quant` is a bound on the possible error in the gradient value due to quantization effects. It is used to determine which gradients are too small to use for the computation of level lines. This value would naturally be `1.0` (because the pixel values are integer values), but it has been found empirically that one gets better results by doubling this to `2.0` (the default).

The `ang_th` argument sets the angular tolerance within which level lines can be considered to have the same orientation as their bounding rectangle. The default value for this parameter is `22.5` (degrees).

The `log_eps` argument is only used with advanced refinement (`cv::LSD_REFINE_ADV`). In this case, it is used to set the log of the cutoff on the *number of false alarms* (NFA). This is the (log of the) parameter

[29] We are hand waving away some math here. The actual concept of significance is based on the probability of generating a similar rectangle on a noise image. As such, "similar n and k" means "those n and k which would give a similar probability", which in fact means that they give the same value for $B(n,k,p) = \sum_{j=k}^{n} \binom{n}{j} p^j (1-p)^{n-j}$. (Here $p = \tau/\pi$.)

ϵ we encountered earlier. The default value for $\log(\epsilon)$ is `0`, which corresponds to one false alarm (per image). You can set this to `-1` or `-2` to prune out some of the more marginal lines.

The next argument, `density_th` is straight forward; it is simply the fraction of aligned points in a particular rectangle required for that rectangle to be considered. The default for this parameter is `0.7`.

The final argument, `n_bins`, is the number of bins in the pseudo-ordering of the gradient modulus. The default number of bins is `1024`.

Once you have constructed your `cv::lineSegmentDetector` argument, you can use it to perform detections on images:

```
void cv::LineSegmentDetector::detect(
  const InputArray image,
  cv::OutputArray  lines,
  cv::OutputArray  width = cv::noArray(),
  cv::OutputArray  prec  = cv::noArray(),
  cv::OutputArray  nfa   = cv::noArray()
);
```

The `input` image should be a grayscale image (`cv::U8C1`). From this image, the resulting lines will be computed and returned. The `lines` output array will be filled with `cv::Vec4i` objects, each of which will contain the x and y location of the point at one end of the line followed by the x and y location of the point at the other end.

The optional parameters `width`, `prec`, and `nfa` return the widths, precisions, and NFA's of each of the found lines. Each of these will be a single column array with the same number of rows as `lines`. The `width` is the transverse width of the associated rectangle, and thus an approximation of the width of the line. The precision (angular tolerance) is initially set to τ/π, and so starts out proportional to `ang_th`. However, the LSD algorithm will attempt to reduce both the width and the precision if doing so will improve the NFA.[30] The final NFA's associated with each line can be returned in the array `nfa`, but only when advanced refinement (`cv::LSD_REFINE_ADV`) is used. The values in `nfa` are the negative log of the NFA for that rectangle (e.g. `+1` corresponds to an NFA of 0.1).

Once you have your lines, there is a convenient method for drawing them on top of your image:

```
void cv::LineSegmentDetector::drawSegments(
  cv::InputOutputArray image,
  cv::InputArray       lines
);
```

Given an `image`, and the `lines` you computed with `cv::LineSegmentDetector::detect()`, an image is produced by adding the lines on top of `image` (in red if `image` is in color).

If you should have two sets of lines, you can also compare them using the method `cv::LineSegmentDetector::compareSegments()`, which has the following prototype:

```
int cv::LineSegmentDetector::compareSegments(
  const cv::Size&      size,
  cv::InputArray       lines1,
  cv::InputArray       lines2,
  cv::InputOutputArray image = noArray()
);
```

In order to use the comparison method, you must provide `size`, which is the size of the original image you found the two sets of lines in (they must come from an image of the same size). You can then provide two sets of lines: `lines1` and `lines2`, in the same format (array of `cv::Vec4i`) that you them from the detector. If you provide an `image` array, the line segment detector will draw both sets of lines (one in blue,

[30] If you think about a line without a smooth edge, reducing the width of the rectangle might well reduce the area of the rectangle much faster than it reduces the number of aligned points. In a similar idea, reducing the precision will reduce the number of aligned points, but may reduce the number of false alarms even faster.

the other in red) on your image. Even if you do not supply an image for visualization, the return value of the segment comparison method tells you how many pixels were not matching in the two line images (i.e. a return value of `0` corresponds to a perfect match).

The Circle Detection

Algorithms such as the Hough Transform can be generalized to other contexts as well. One particularly useful example is circle detection which, as we will see, employs a very similar mechanism to the Hough Line Transform to find circles in an image.

Hough Circle Transform

The *Hough circle transform* [Kimme75] (see

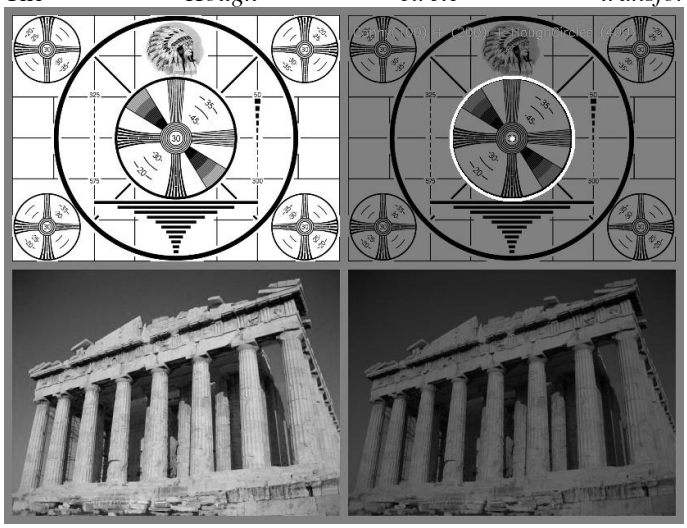


Figure 6-16) works in a manner roughly analogous to the Hough line transforms just described. The reason it is only "roughly" is that—if one were to try doing the exactly analogous thing—the accumulator *plane* would have to be replaced with an accumulator *volume* with three dimensions: one for x and one for y (the location of the circle center), and another for the circle radius r. This would mean far greater memory requirements and much slower speed. The implementation of the circle transform in OpenCV avoids this problem by using a somewhat more tricky method called the *Hough gradient method*.

The Hough gradient method works as follows. First, the image is passed through an edge detection phase (in this case, `cv::Canny()`). Next, for every nonzero point in the edge image, the local gradient is considered (the gradient is computed by first computing the first-order Sobel x- and y-derivatives via `cv::Sobel()`). Using this gradient, every point along the line indicated by this slope—from a specified minimum to a specified maximum distance—is incremented in the accumulator. At the same time, the location of every one of these nonzero pixels in the edge image is noted. The candidate centers are then selected from those points in this (two-dimensional) accumulator that are both above some given threshold and larger than all of their immediate neighbors. These candidate centers are sorted in descending order of their accumulator values, so that the centers with the most supporting pixels appear first. Next, for each center, all of the nonzero pixels (recall that this list was built earlier) are considered. These pixels are sorted according to their distance from the center. Working out from the smallest distances to the maximum radius, a single radius is selected that is best supported by the nonzero pixels. A center is kept if it has sufficient support from the nonzero pixels in the edge image *and* if it is a sufficient distance from any previously selected center.

This implementation enables the algorithm to run much faster and, perhaps more importantly, helps overcome the problem of the otherwise sparse population of a three-dimensional accumulator, which would

lead to a lot of noise and render the results unstable. On the other hand, this algorithm has several shortcomings that you should be aware of.

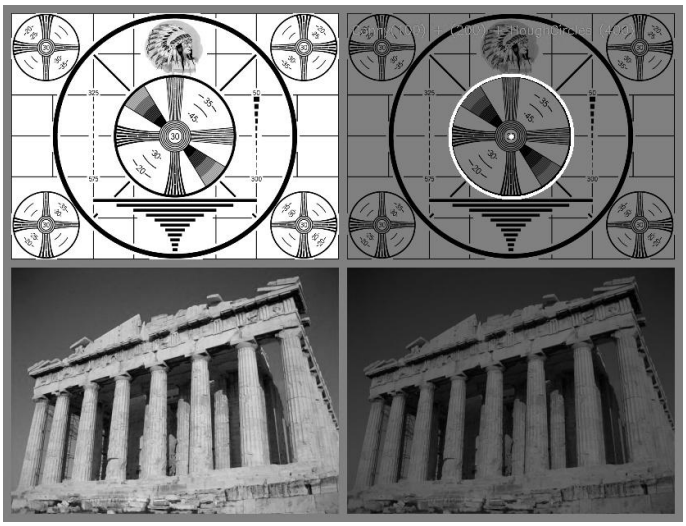


Figure 6-16: The Hough circle transform finds some of the circles in the test pattern and (correctly) finds none in the photograph

First, the use of the Sobel derivatives to compute the local gradient—and the attendant assumption that this can be considered equivalent to a local tangent—is not a numerically stable proposition. It might be true "most of the time," but you should expect this to generate some noise in the output.

Second, the entire set of nonzero pixels in the edge image is considered for every candidate center; hence, if you make the accumulator threshold too low, the algorithm will take a long time to run. Third, because only one circle is selected for every center, if there are concentric circles then you will get only one of them.

Finally, because centers are considered in ascending order of their associated accumulator value and because new centers are not kept if they are too close to previously accepted centers, there is a bias toward keeping the larger circles when multiple circles are concentric or approximately concentric. (It is only a "bias" because of the noise arising from the Sobel derivatives; in a smooth image at infinite resolution, it would be a certainty.)

`cv::HoughCircles()`, the Hough Circle Transform

The Hough circle transform function `cv::HoughCircles()` has similar arguments to the line transform.

```
void cv::HoughCircles(
  cv::InputArray  image,                        // Input single channel image
  cv::OutputArray circles,                      // N-by-1 3-channel or vector of Vec3f
  int             method,                       // Always cv::HOUGH_GRADIENT
  double          dp,                           // Accumulator resolution (ratio)
  double          minDist,                      // Required separation (between lines)
  double          param1    = 100,              // Upper Canny threshold
  double          param2    = 100,              // Un-normalized accumulator threshold
  int             minRadius = 0,                // Smallest radius to consider
  int             maxRadius = 0                 // Largest radius to consider
);
```

The input `image` is again an 8-bit image. One significant difference between `cv::HoughCircles()` and `cv::HoughLines()` is that the latter requires a binary image. The `cv::HoughCircles()`

function will internally (automatically) call `cv::Sobel()`[31] for you, so you can provide a more general grayscale image.

The result array `circles` will be either a matrix-array or a vector, depending on what you pass to `cv::HoughCircles()`. If a matrix is used, it will be a one-dimensional array of type `cv::F32C3`; the three channels will be used to encode the location of the circle and its radius. If a vector is used, it must be of type `std::vector<Vec3f>`. The `method` argument must always be set to `cv::HOUGH_GRADIENT`.

The parameter `dp` is the resolution of the accumulator image used. This parameter allows us to create an accumulator of a lower resolution than the input image. It makes sense to do this because there is no reason to expect the circles that exist in the image to fall naturally into the same number of bins as the width or height of the image itself. If `dp` is set to `1` then the resolutions will be the same; if set to a larger number (e.g., `2`), then the accumulator resolution will be smaller by that factor (in this case, half). The value of `dp` cannot be less than `1`.

The parameter `minDist` is the minimum distance that must exist between two circles in order for the algorithm to consider them distinct circles.

For the (currently required) case of the method being set to `cv::HOUGH_GRADIENT`, the next two arguments, `param1` and `param2`, are the edge (Canny) threshold and the accumulator threshold, respectively. You may recall that the Canny edge detector actually takes two different thresholds itself. When `cv::Canny()` is called internally, the first (higher) threshold is set to the value of `param1` passed into `cv::HoughCircles()`, and the second (lower) threshold is set to exactly half that value. The parameter `param2` is the one used to threshold the accumulator and is exactly analogous to the `threshold` argument of `cv::HoughLines()`.

The final two parameters are the minimum and maximum radius of circles that can be found. This means that these are the radii of circles for which the accumulator has a representation. Example 6-5 shows an example program using `cv::HoughCircles()`.

Example 6-5: Using `cv::HoughCircles()` to return a sequence of circles found in a grayscale image

```
#include <opencv2/opencv.hpp>
#include <iostream>
#include <math.h>

using namespace cv;
using namespace std;

int main(int argc, char** argv) {

  if(argc != 2) {
    cout << "Hough Circle detect\nUsage: ch6_ex6_1 <imagename>\n" << endl;
    return -1;
  }

  cv::Mat src = cv::imread(argv[1], 1), image;
  if( src.empty() ) { cout << "Can not load " << argv[1] << endl; return -1; }
  cv::cvtColor(src, image, cv::BGR2GRAY);

  cv::GaussianBlur(image, image, Size(5,5), 0, 0);

  vector<cv::Vec3f> circles;
  cv::HoughCircles(image, circles, cv::HOUGH_GRADIENT, 2, image.cols/10);

  for( size_t i = 0; i < circles.size(); ++i ) {
    cv::circle(
```

[31] The function `cv::Sobel()`, not `cv::Canny()`, is called internally. The reason is that `cv::HoughCircles()` needs to estimate the orientation of a gradient at each pixel, and this is difficult to do with binary edge map.

```
            src,
            cv::Point(cvRound(circles[i][0]), cvRound(circles[i][1])),
            cvRound(circles[i][2]),
            cv::Scalar(0,0,255),
            2,
            cv::AA
          );
        }
        cv::imshow( "Hough Circles", src);
        cv::waitKey(0);
        return 0;
    }
```

It is worth reflecting momentarily on the fact that, no matter what tricks we employ, there is no getting around the requirement that circles be described by three degrees of freedom (x, y, and r), in contrast to only two degrees of freedom (ρ and θ) for lines. The result will invariably be that any circle-finding algorithm requires more memory and computation time than the line-finding algorithms we looked at previously. With this in mind, it's a good idea to bound the radius parameter as tightly as circumstances allow in order to keep these costs under control.[32] The Hough transform was extended to arbitrary shapes by Ballard in 1981 [Ballard81] basically by considering objects as collections of gradient edges.

Distance Transformation

The *distance transform* of an image is defined as a new image in which every output pixel is set to a value equal to the distance to the nearest zero pixel in the input image—according to some specific distance metric. It should be immediately obvious that the typical input to a distance transform should be some kind of edge image. In most applications the input to the distance transform is an output of an edge detector such as the Canny edge detector that has been inverted (so that the edges have value zero and the non-edges are nonzero).

There are two methods available to compute the distance transform. The first method uses a mask that is typically a 3-by-3 or 5-by-5 array. Each point in the array defines the "distance" to be associated with a point in that particular position relative to the center of the mask. Larger distances are built up (and thus approximated) as sequences of "moves" defined by the entries in the mask. This means that using a larger mask will yield more accurate distances. When using this method, given a specific distance metric, the appropriate mask is automatically selected from a set known to OpenCV. This is the "original" method developed by Borgefors (1986) [Borgefors86]. The second method computes exact distances, and is due to Felzenszwalb [Felzenszwalb04]. Both methods run in time linear in the total number of pixels, but the exact algorithm is still slower.

[32] Although `cv::HoughCircles()` catches centers of the circles quite well, it sometimes fails to find the correct radius. Therefore, in an application where only a center must be found (or where some different technique can be used to find the actual radius), the radius returned by `cv::HoughCircles()` can be ignored.

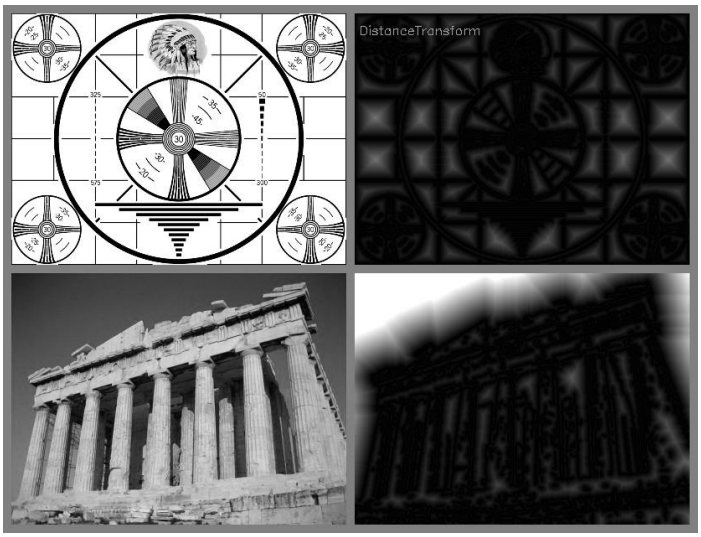


Figure 6-17: First a Canny edge detector was run with `param1=100` and `param2=200`; then the distance transform was run with the output scaled by a factor of `5` to increase visibility

The distance metric can be any of several different types, including the classic L2 (Cartesian) distance metric.

cv::distanceTransform(), for Unlabeled Distance Transform

When calling the OpenCV distance transform function, the output image will be a 32-bit floating-point image (i.e., `cv::F32`).

```
void cv::distanceTransform(
  cv::InputArray  src,                    // Input Image
  cv::OutputArray dst,                    // Result image
  int             distanceType,           // Distance metric to use
  int             maskSize                // Mask to use (3, 5, or see below)
);
```

`cv::distanceTransform()` has some additional parameters. The first is `distanceType`, which indicates the distance metric to be used. Your choices here are `cv::DIST_C`, `cv::DIST_L1` and `cv::DIST_L2`. These methods compute the distance to the nearest zero based on integer steps along a grid. The difference between the methods is that `cv::DIST_C` is the distance when the steps are counted on a 4-connected grid (i.e., diagonal moves are not allowed), and `cv::DIST_L1` gives the number of steps on an 8-connected grid (i.e., diagonal moves are allowed). When `distanceType` is set to `cv::DIST_L2`, `cv::distanceTransform()` attempts to compute the exact Euclidian distances.

After the distance type is the `maskSize`, which may be `3`, `5`, or `cv::DIST_MASK_PRECISE`. In the case of 3 or 5, this argument indicates that a 3-by-3 or 5-by-5 mask should be used with the Borgefors method. If you are using `cv::DIST_L1` or `cv::DIST_C`, you can always use a 3-by-3 mask and you will get exact results. If you are using `cv::DIST_L2`, the Borgefors method is always approximate, and using the larger 5-by-5 mask will result in a better approximation to the L_2 distance, at the cost of a slightly slower computation. Alternatively, `cv::DIST_MASK_PRECISE` can be used to indicate the Felzenszwalb algorithm (when used with `cv::DIST_L2`).

cv::distanceTransform(), for Labeled Distance Transform

It is also possible to ask the distance transform algorithm to not only calculate the distances, but to also report which object that minimum distance is to. These "objects" are called *connected components*. We will have a lot more to say about connected components in Chapter 8, but for now, you can think of them as exactly what they sound like; they are structures made of continuously connected groups of zeros in the source image.

```
void cv::distanceTransform(
  cv::InputArray  src,                          // Input Image
  cv::OutputArray dst,                          // Result image
  cv::OutputArray labels,                       // Output connected componet id's
  int             distanceType,                 // Distance metric to use
  int             maskSize,                     // Mask to use (3, 5, or see below)
  int             labelType = cv::DIST_LABEL_CCOMP  // How to label
);
```

If a `labels` array is provided, then as a result of running `cv::distanceTransform()` it will be of the same size as `dst`. In this case, the computation of connected components will be done automatically, and the label associated with the nearest[33] such component will be placed in each pixel of `labels`.

> If you are wondering how to differentiate labels, consider that for any pixel that is zero in `src`, then the corresponding distance must also be zero. In addition to this, the label for that pixel must be the label of the connected component it is part of. As a result, if you want to know what label was given to any particular zero pixel, you need only look up that pixel in `labels`.

The argument `labelType` can be set either to `cv::DIST_LABEL_CCOMP` or `cv::DIST_LABEL_PIXEL`. In the former case, the function automatically finds connected components of zero pixels in the input image and gives each one a unique label. In the latter case, all zero pixels are given distinct labels.

Histogram Equalization

Cameras and image sensors must usually deal not only with the contrast in a scene but also with the image sensors' exposure to the resulting light in that scene. In a standard camera, the shutter and lens aperture settings juggle between exposing the sensors to too much or too little light. Often the range of contrasts is too much for the sensors to deal with; hence, there is a trade-off between capturing the dark areas (e.g., shadows), which requires a longer exposure time, and the bright areas, which require shorter exposure to avoid saturating "whiteouts."

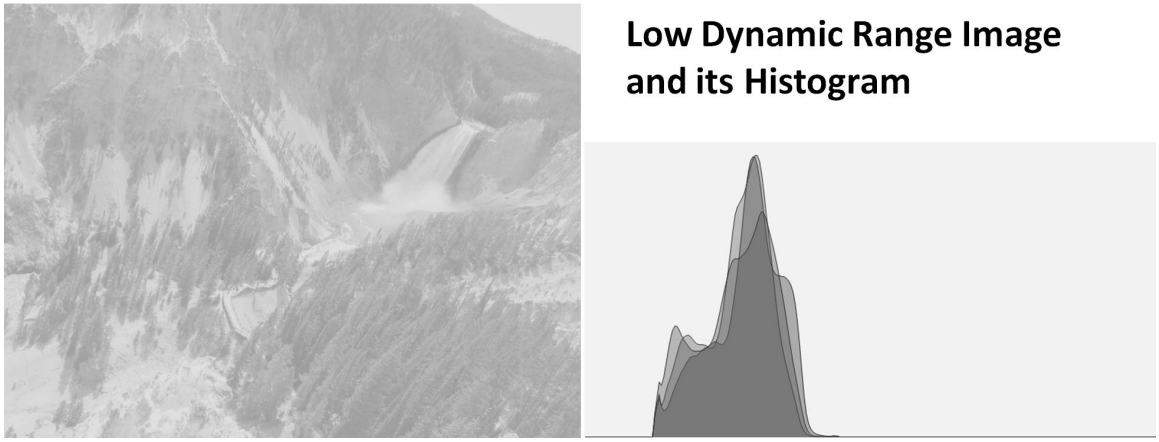


Figure 6-18: The image on the left has poor contrast, as is confirmed by the histogram of its intensity values on the right

After the picture has been taken, there's nothing we can do about what the sensor recorded; however, we can still take what's there and try to expand the dynamic range of the image to increase its contrast. The

[33] The output "labels" array will basically be the discrete Voronoi diagram.

most commonly used technique for this is histogram equalization.[34] [35] In

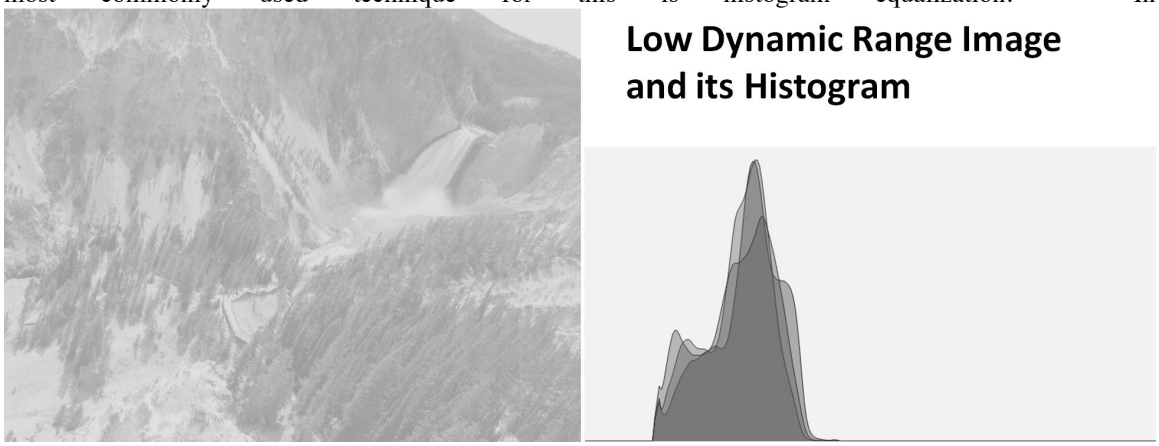


Figure 6-18 we can see that the image on the left is poor because there's not much variation of the range of values. This is evident from the histogram of its intensity values on the right. Because we are dealing with an 8-bit image, its intensity values can range from `0` to `255`, but the histogram shows that the actual intensity values are all clustered near the middle of the available range. Histogram equalization is a method for stretching this range out.

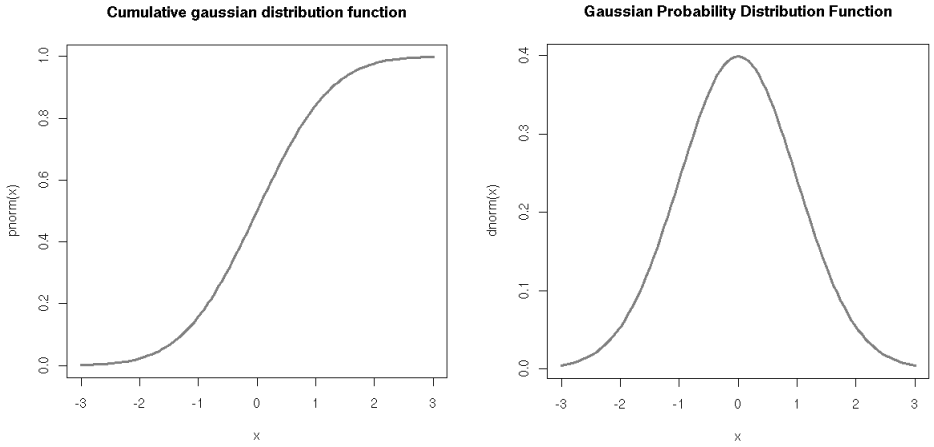


Figure 6-19: Result of cumulative distribution function (left) computed for a Gaussian distribution (right)

The underlying math behind histogram equalization involves mapping one distribution (the given histogram of intensity values) to another distribution (a wider and, ideally, uniform distribution of intensity values). That is, we want to spread out the y-values of the original distribution as evenly as possible in the new distribution. It turns out that there is a good answer to the problem of spreading out distribution values: the remapping function should be the *cumulative distribution function*. An example of the cumulative density function is shown in **Error! Reference source not found.** for the somewhat idealized case of a distribution that was originally pure Gaussian. However, cumulative density can be applied to any distribution; it is just the running sum of the original distribution from its negative to its positive bounds.

---

[34] If you are wondering why histogram equalization is not in the chapter on histograms (Chapter 7), the reason is that histogram equalization makes no explicit use of any histogram data types. Although histograms are used internally, the function (from the user's perspective) requires no histograms at all.

[35] Histogram equalization is an old mathematical technique; its use in image processing is described in various textbooks [Jain86; Russ02; Acharya05], conference papers [Schwarz78], and even in biological vision [Laughlin81].

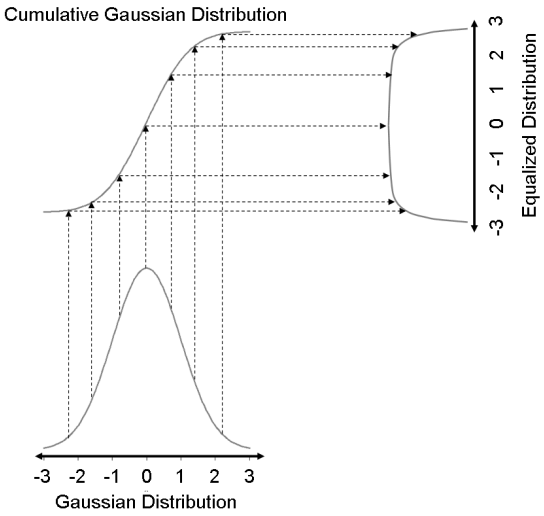
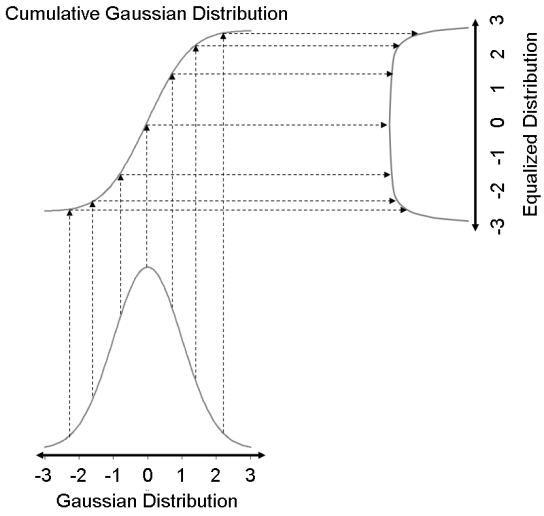


Figure 6-20: Using the cumulative density function to equalize a Gaussian distribution

We may use the cumulative distribution function to remap the original distribution to an equally spread



distribution (see

Figure 6-20) simply by looking up each y-value in the original distribution and seeing where it should go in the equalized distribution.

For continuous distributions the result will be an exact equalization, but for digitized/discrete distributions the results may be far from uniform.

Applying this equalization process to

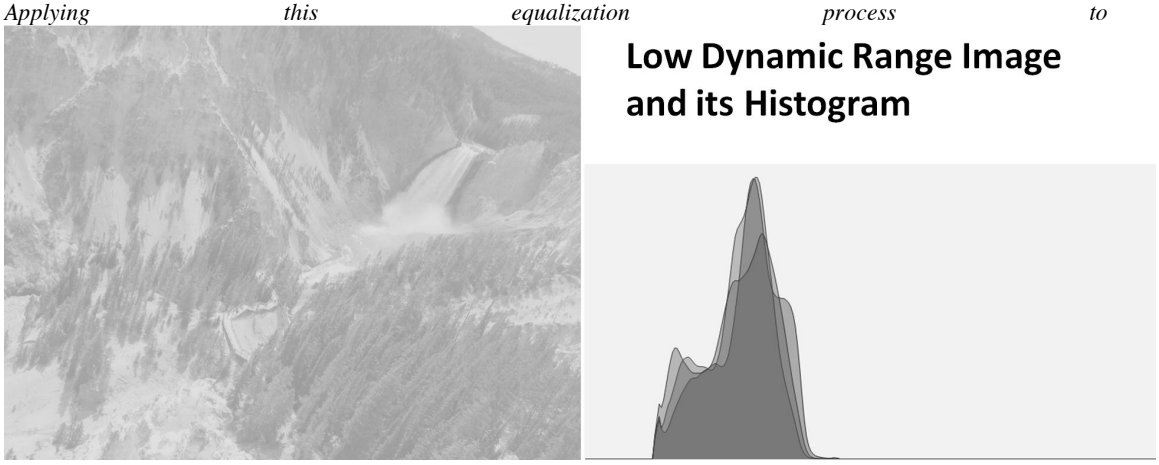


Figure 6-18 yields the equalized intensity distribution histogram and resulting image in **Error! Reference source not found.**.

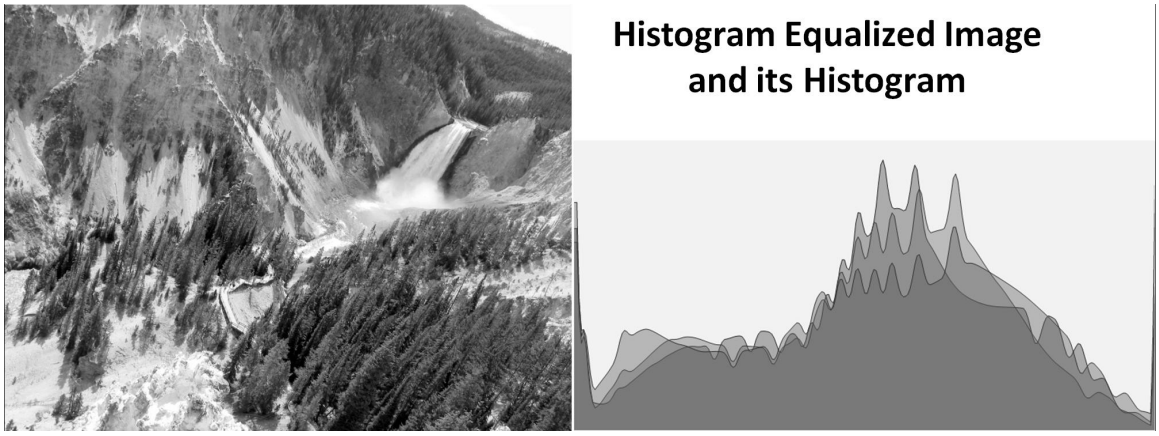


Figure 6-21: Histogram equalized results: the spectrum has been spread out

`cv::equalizeHist()`, Contrast Equalization

OpenCV wraps this whole process up in one neat function.

```
void cv::equalizeHist(
  const cv::InputArray  src,                    // Input Image
  cv::OutputArray dst                           // Result image
);
```

In `cv::equalizeHist()`, the source `src` must be a single-channel, 8-bit image. The destination image `dst` will be the same. For color images you will have to separate the channels and process them one by one.

Segmentation

The topic of image segmentation is a large one, which we have touched on in several places already, and will return to in more sophisticated contexts later in the book as well. Here, we will focus on several methods of the library that specifically implement techniques that are either segmentation methods in themselves, or primitives that will be used later by more sophisticated tactics. It should be noted that at this time, there is no general "magic" solution for image segmentation, and it remains a very active area in

computer vision research. Despite this, many good techniques have been developed that are reliable at least in some specific domain, and in practice can yield very good results.

Flood Fill

Flood fill [Heckbert00; Shaw04; Vandevenne04] is an extremely useful function that is often used to mark or isolate portions of an image for further processing or analysis. Flood fill can also be used to derive, from an input image, masks that can be used for subsequent routines to speed or restrict processing to only those pixels indicated by the mask. The function `cv::floodFill()` itself takes an optional mask that can be further used to control where filling is done (e.g., when doing multiple fills of the same image).

In OpenCV, flood fill is a more general version of the sort of fill functionality that you probably already associate with typical computer painting programs. For both, a *seed point* is selected from an image and then all similar neighboring points are colored with a uniform color. The difference is that the neighboring pixels need not all be identical in color.[36] The result of a flood fill operation will always be a single contiguous region. The `cv::floodFill()` function will color a neighboring pixel either if it is within a specified range (`loDiff` to `upDiff`) of the current pixel or if (depending on the settings of `flags`) the neighboring pixel is within a specified range of the original `seed` value.

The `cv::floodFill()` function will color a neighboring pixel if it is within a specified range (`loDiff` to `upDiff`) of either the current pixel or if (depending on the settings of `flags`) the neighboring pixel is within a specified range of the original `seed` value. Flood filling can also be constrained by an optional mask argument. There are two different prototypes for the `cv::floodFill()` routine, one that accepts an explicit `mask` parameter, and one that does not.

```
int cv::floodFill(
  cv::InputOutputArray image,                  // Input image, 1 or 3 channels
  cv::Point            seed,                   // Start point for flood
  cv::Scalar           newVal,                 // Value for painted pixels
  cv::Rect*            rect,                   // Output bounds painted domain
  cv::Scalar           lowDiff  = cv::Scalar(),// Maximum down color distance
  cv::Scalar           highDiff = cv::Scalar(),// Maximum up color distance
  int                  flags                   // Local vs. global, and mask-only
);

int cv::floodFill(
  cv::InputOutputArray image,                  // Input w-by-h image, 1 or 3 channels
  cv::InputOutputArray mask,                   // Singl-channel 8-bit, w+2-by-h+2
  cv::Point            seed,                   // Start point for flood
  cv::Scalar           newVal,                 // Value for painted pixels
  cv::Rect*            rect,                   // Output bounds painted domain
  cv::Scalar           lowDiff  = cv::Scalar(),// Maximum down color distance
  cv::Scalar           highDiff = cv::Scalar(),// Maximum up color distance
  int                  flags                   // Local vs. global, and mask-only
);
```

The parameter `image` is the input image, which can be 8-bit or a floating-point type, and must either have one or three channels. In general, this `image` array will be modified by `cv::floodFill()`. The flood fill process begins at the location `seed`. The seed will be set to value `newVal`, as will all subsequent pixels colored by the algorithm. A pixel will be colorized if its intensity is not less than a colorized neighbor's intensity minus `loDiff` and not greater than the colorized neighbor's intensity plus `upDiff`. If the `flags` argument includes `cv::FLOODFILL_FIXED_RANGE`, then a pixel will be compared to the original seed point rather than to its neighbors. Generally, the `flags` argument controls the connectivity of the fill, what the fill is relative to, whether we are filling only a mask, and what values are used to fill the

[36] Users of contemporary painting and drawing programs should note that most of them now employ a filling algorithm very much like `cv::floodFill()`.

mask. Our first example of flood fill is shown in

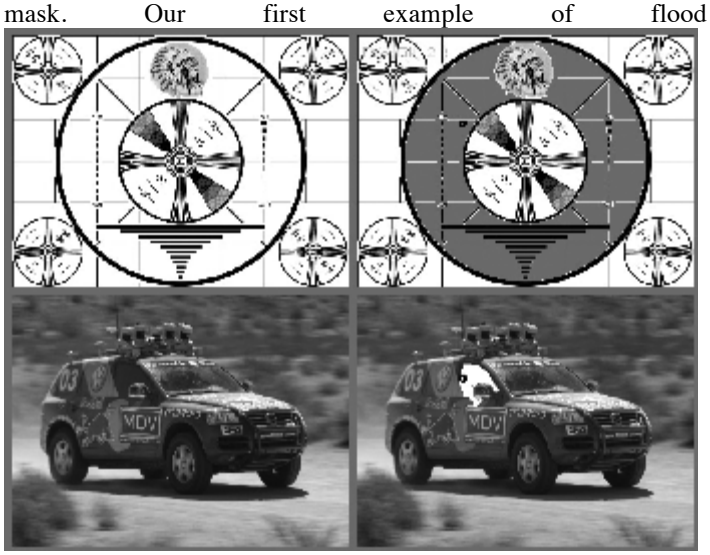
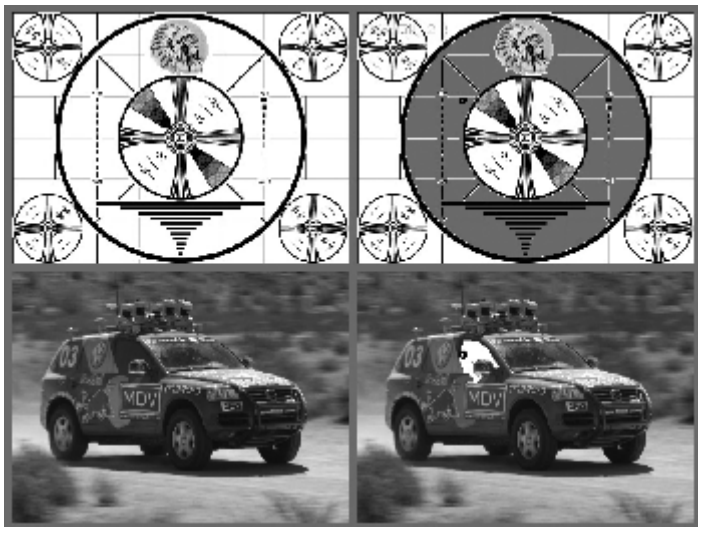


Figure 6-22.

The argument `mask` indicates a mask that can function both as input to `cv::floodFill()` (in which case, it constrains the regions that can be filled) and as output from `cv::floodFill()` (in which case, it will indicate the regions that actually were filled). `mask` must be a single-channel 8-bit image whose size is exactly two pixels larger in width and height than the source image[37].

In the sense that `mask` is an *input* to `cv::floodFill()`, the algorithm will not flood across nonzero pixels in the mask. As a result you should zero it before use if you don't want masking to block the flooding operation.

When the `mask` is present, it will also be used as an *output*. When the algorithm runs, every "filled" pixel will be set to a nonzero value in the mask. You have the option of adding the value `cv::FLOODFILL_MASK_ONLY` to flags (using the usual Boolean OR operator). In this case, the input `image` will not be modified at all. Instead, only `mask` will be modified.



[37] This is done to make processing easier and faster for the internal algorithm. Note that since the mask is larger than the original image, pixel (x, y) in `image` corresponds to pixel $(x + 1, y + 1)$ in `mask`. This is, however, an excellent opportunity to use `cv::Mat::getSubRect()`.

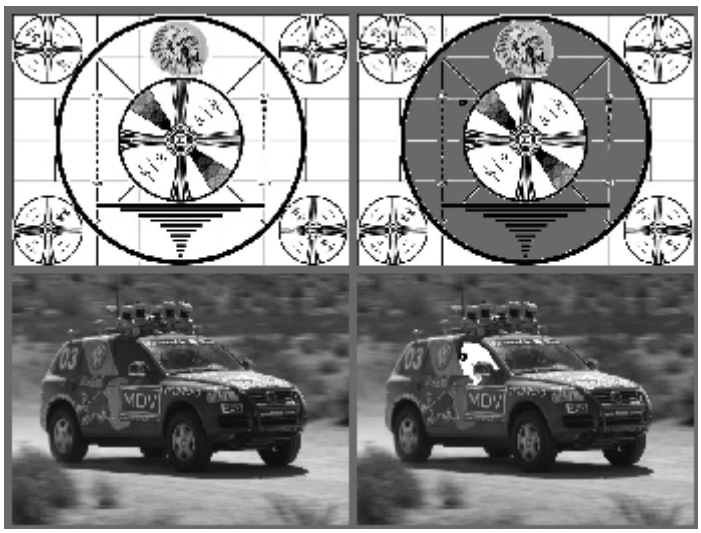
Figure 6-22: Results of flood fill (top image is filled with gray, bottom image with white) from the dark circle located just off center in both images; in this case, the `upDiff` and `loDiff` parameters were each set to `7.0`

> If the flood-fill mask is used, then the mask pixels corresponding to the repainted image pixels are set to `1`. Don't be confused if you fill the mask and see nothing but black upon display; the filled values are there, but the mask image needs to be rescaled if you want to display it in a way you can actually see it on the screen. After all, the difference between `0` and `1` is pretty small on an intensity scale of `0` to `255`.

Two possible values for `flags` have already been mentioned: `cv::FLOODFILL_FIXED_RANGE` and `cv::FLOODFILL_MASK_ONLY`. In addition to these, you can also add the numerical values `4` or `8`.[38] In this case, you are specifying whether the flood fill algorithm should consider the pixel array to be *four-connected* or *eight-connected*. In the former case, a four-connected array is one in which pixels are only connected to their four nearest neighbors (left, right, above, and below). In the latter eight-connected case, pixels are considered to be connected to diagonally neighboring pixels as well.

It's time to clarify the `flags` argument, which is tricky because it has three parts. The *low* 8 bits (0–7) can be set to `4` or `8`. This controls the connectivity considered by the filling algorithm. If set to `4`, only horizontal and vertical neighbors to the current pixel are considered in the filling process; if set to `8`, flood fill will additionally include diagonal neighbors. The *high* 8 bits (16–23) can be set with the flags `cv::FLOODFILL_FIXED_RANGE` (fill relative to the seed point pixel value; otherwise, fill relative to the neighbor's value), and/or `cv::FLOODFILL_MASK_ONLY` (fill the mask location instead of the source image location). Obviously, you must supply an appropriate mask if `cv::FLOODFILL_MASK_ONLY` is set. The *middle* bits (8–15) of `flags` can be set to the value with which you want the mask to be filled. If the middle bits of `flags` are `0`s, the mask will be filled with `1`s. All these flags may be linked together via OR. For example, if you want an 8-way connectivity fill, filling only a fixed range, filling the mask not the image, and filling using a value of `47`, then the parameter to pass in would be:

```
flags = 8
      | cv::FLOODFILL_MASK_ONLY
      | cv::FLOODFILL_FIXED_RANGE
      | (47<<8);
```



[38] The text here reads "add," but recall that `flags` is really a bit-field argument. Conveniently, however, `4` and `8` are single bits. So you can use "add" or "OR," whichever you prefer (e.g., `flags = 8 | cv::FLOODFILL_MASK_ONLY`).

Figure 6-22 shows flood fill in action on a sample image. Using `cv::FLOODFILL_FIXED_RANGE` with a wide range resulted in most of the image being filled (starting at the center). We should note that `newVal`, `loDiff`, and `upDiff` are prototyped as type `cv::Scalar` so they can be set for three channels at once. For example, `loDiff = cv::Scalar(20,30,40)` will set `lo_iff` thresholds of `20` for red, `30` for green, and `40` for blue.

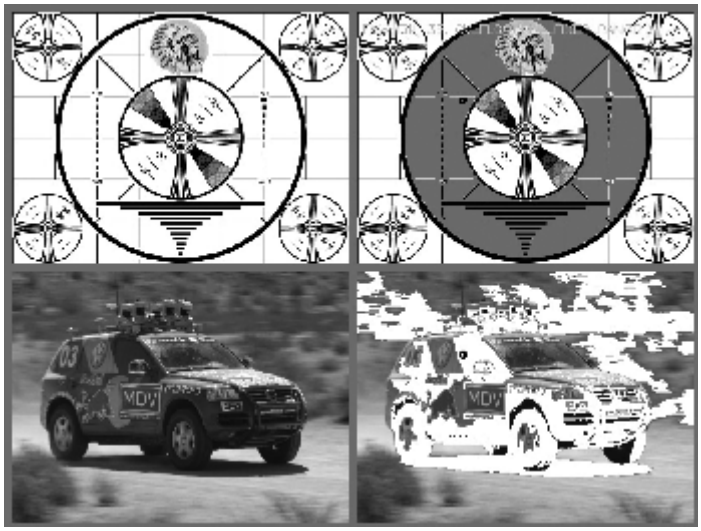


Figure 6-23: Results of flood fill (top image is filled with gray, bottom image with white) from the dark circle located just off center in both images; in this case, flood fill was done with a fixed range and with a high and low difference of `25.0`

Watershed Algorithm

In many practical contexts, we would like to segment an image but do not have the benefit of a separate background image. One technique that is often effective in this context is the *watershed algorithm [Meyer92]*. This algorithm converts lines in an image into "mountains" and uniform regions into "valleys" that can be used to help segment objects. The watershed algorithm first takes the gradient of the intensity image; this has the effect of forming valleys or *basins* (the low points) where there is no texture and of forming mountains or *ranges* (high ridges corresponding to edges) where there are dominant lines in the image. It then successively floods basins starting from user-specified (or algorithm-specified) points until these regions meet. Regions that merge across the marks so generated are segmented as belonging together as the image "fills up." In this way, the basins connected to the marker point become "owned" by that marker. We then segment the image into the corresponding marked regions.

More specifically, the watershed algorithm allows a user (or another algorithm!) to mark parts of an object or background that are known to be part of the object or background. The user or algorithm can draw a simple line that effectively tells the watershed algorithm to "group points like these together." The watershed algorithm then segments the image by allowing marked regions to "own" the edge-defined

valleys in the gradient image that are connected with the segments.

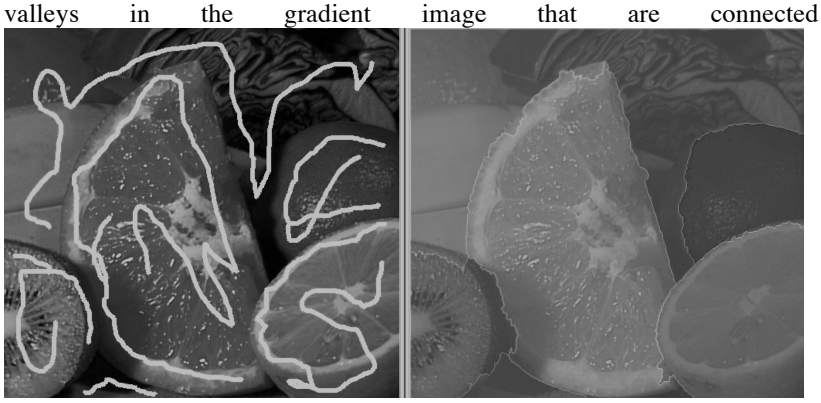


Figure 6-24 clarifies this process.

The function specification of the watershed segmentation algorithm is:

```
void cv::watershed(
  cv::InputArray image,                          // Input 8-bit, 3 channels
  cv::InputOutputArray markers                   // I/O 32-bit float, single channel
);
```

Here, `image` must be an 8-bit, 3-channel (color) image and `markers` is a single-channel integer (`cv::S32`) image of the same (x, y) dimensions. On input, the value of `markers` is `0` *except* where the user (or an algorithm) has indicated by using positive numbers that some regions belong together. For example, in the left panel of

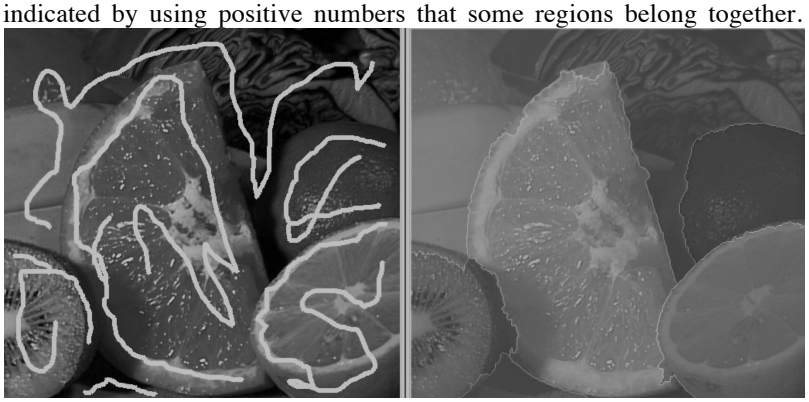


Figure 6-24, the orange might have been marked with a `1`, the lemon with a `2`, the lime with `3`, the upper background with `4`, and so on.

After the algorithm has run, all of the former zero value pixels in `markers` will be set to one of the given markers (i.e., all of the pixels of the orange are hoped to come out with a `1` on them, the pixels of the lemon with a `2`, etc.), except the boundary pixels between regions, which will be set to `-1`.

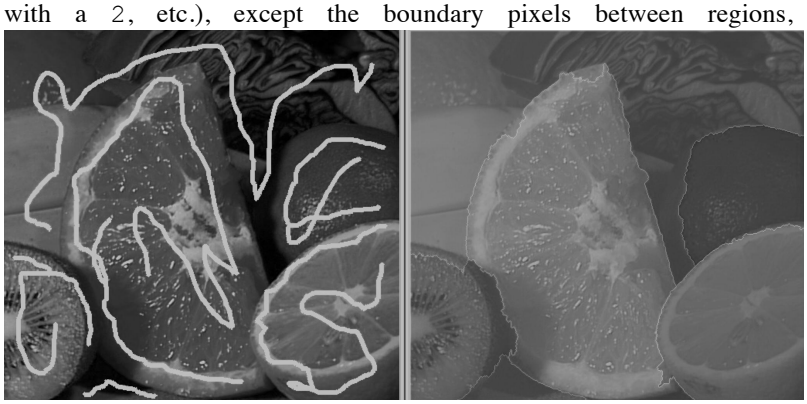


Figure 6-24 (right) shows an example of such a segmentation.

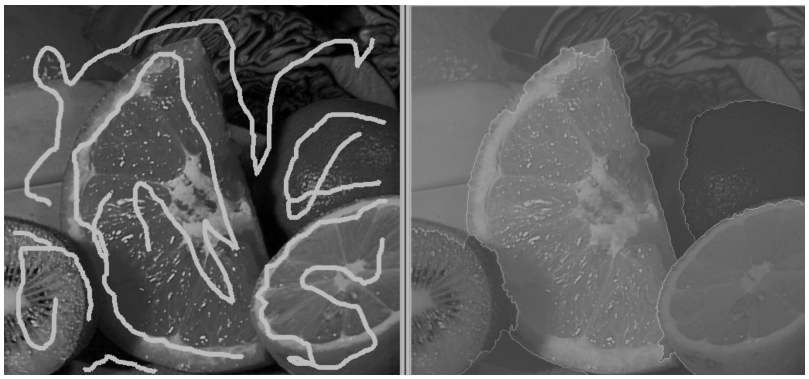


Figure 6-24: Watershed algorithm: after a user has marked objects that belong together (left), the algorithm then merges the marked area into segments (right)

One small word of warning is in order here. It is tempting to think that all regions will be separated by pixels with marker value `-1` at their boundaries. This is not actually the case, however. Notably, if two neighboring pixels were input originally with nonzero but distinct values, they will remain touching and not separated by a `-1` pixel on output.

Grabcuts

The Grabcuts algorithm, introduced by Rother, Kolmogorov, and Blake [Rother04], extends the Graphcuts algorithm [Boykov01] for use in user-directed image segmentation. The grabcuts algorithm is capable of obtaining excellent segmentations, often with no more than a bounding rectangle around the foreground object to be segmented.

This Graphcuts algorithm uses user-labeled foreground and background regions to establish distribution histograms for those two classes of image regions. It then combines the assertion that unlabeled foreground or background should conform to similar distributions, with the idea that these regions should tend to be smooth and connected (i.e., a bunch of blobs). These assertions are combined into an *energy functional* which gives a low energy (i.e., cost) to solutions which conform to these assertions and a high energy to solutions which violate them. The final result is obtained by minimizing this energy function[39].

The Grabcuts algorithm extends Graphcuts in several important ways. The first is that it replaces the histogram models with a Gaussian mixture model, enabling the algorithm to work on color images. In addition, it solves the energy functional minimization problem in an iterative manner, which provides better overall results, and allows much greater flexibility in the labeling provided by the user. Notably, this latter point allows even one-sided labelings which identify either background or foreground pixels (while Graphcuts required both).

The implementation in OpenCV allows the caller to either just provide a rectangle around the object to be segmented, in which case the pixels "under" the rectangle's boundary are taken to be background and no foreground is specified. Alternatively, the caller can specify an overall mask in which pixels are categorized as being either definitely foreground, definitely background, probably foreground, and probably background. In this case, the definite regions will be used to classify the other regions, with the latter being classified into the definite categories by the algorithm.

The OpenCV implementation of Grabcuts is implemented by the `cv::Grabcuts()` function:

```
void cv::grabCut(
  cv::InputArray       img,
  cv::InputOutputArray mask,
  cv::Rect             rect,
```

---

[39] This minimization is a non-trivial problem. In practice it is performed using a technique called *Mincut*, which is how both the Graphcuts and Grabcuts algorithms get their respective names.

```
  cv::InputOutputArray bgdModel,
  cv::InputOutputArray fgdModel,
  int                  iterCount,
  int                  mode     = cv::GC_EVAL
);
```

Given an input image `img`, the resulting labeling will be computed by `cv::grabCut()` and placed in the output array `mask`. This `mask` array can also be used as an input. This is determined by the `mode` variable. If `mode` contains the flag `cv::GC_INIT_WITH_MASK`[40], then the values currently in `mask` when it is called will be used to initialize the labeling of the image. The mask is expected to be a single channel image of `cv::U8` type in which every value is one of the following enumerations:

Enumerated value	Numerical value	Meaning
`cv::GC_BGD`	0	Background (definitely)
`cv::GC_FGD`	1	Foreground (definitely)
`cv::PR_GC_BGD`	2	Probably Background
`cv::PR_GC_FGD`	3	Probably Foreground

The argument `rect` is only used when you are not using mask initialization. When the mode contains the flag `cv::GC_INIT_WITH_RECT`, the entire region outside of the provided rectangle is taken to be "definitely background", while the rest is automatically set to "probably foreground".

The next two arrays are essentially temporary buffers. When you first call `cv::grabCut()`, they can be empty arrays. However, if for some reason you should run the Grabcuts algorithm for some number of iterations and then want to restart the algorithm for more iterations (possibly after allowing a user to provide additional "definite" pixels to guide the algorithm) you will need to pass in the same (unmodified) buffers that were filled by the previous run (in addition to using the mask you got back from the previous run as input for the next run).

The Grabcuts algorithm essentially runs the Graphcuts algorithm some number of times. Between each such run, the mixture models are recomputed. The `itercount` argument determines how many such iterations will be applied. Typical values for `itercount` are `10` or `12`, though the number required may depend on the size and nature of the image being processed.

Mean-Shift Segmentation

Mean-shift segmentation finds the peaks of color distributions over space [Comaniciu99]. It is related to the "Mean Shift Algorithm," which we will discuss in Chapter 10, when we discuss tracking and motion. The main difference between the two is that the former looks at spatial distributions of color (and is thus related to our current topic of segmentation), while the latter tracks those distributions through time in successive

[40] Actually, you do not need to explicitly provide the `cv::GC_INIT_WITH_MASK` flag, because mask initialization is the default behavior. So as long as you do not provide the `cv::GC_INIT_WITH_RECT` flag, you will get mask initialization. However, this is not implemented as a default argument, but rather a default in the procedural logic of the function, and is therefore not guaranteed to remain unchanged in future releases of the library. It is best to either use the `cv::GC_INIT_WITH_MASK` flag or the `cv::GC_INIT_WITH_RECT` flag explicitly, both for future proofing, and for general clarity.

frames. The function that does this segmentation based on the color distribution peaks is `cv::pyrMeanShiftFiltering()`.

Given a set of multidimensional data points whose dimensions are $(x, y, I_{blue}, I_{green}, I_{red})$, mean shift can find the highest density "clumps" of data in this space by scanning a *window* over the space. Notice, however, that the spatial variables (x, y) can have very different ranges from the color magnitude ranges $(I_{blue}, I_{green}, I_{red})$. Therefore, mean shift needs to allow for different window radii in different dimensions. In this case we should have one radius for the spatial variables (`spatialRadius`) and one radius for the color magnitudes (`colorRadius`). As mean-shift windows move, all the points traversed by the windows that converge at a peak in the data become connected to or "owned" by that peak. This ownership, radiating out from the densest peaks, forms the segmentation of the image. The segmentation is actually done over a scale pyramid (`cv::pyrUp(), cv::pyrDown()`), as described in Chapter 5, so that color clusters at a high level in the pyramid (shrunken image) have their boundaries refined at lower pyramid levels in the pyramid.

The output of the mean-shift segmentation algorithm is a new image that has been "posterized," meaning that the fine texture is removed and the gradients in color are mostly flattened. Such images can then be further segmented using whatever algorithm is appropriate for your needs (e.g., `cv::Canny()` combined with `cv::findContours()`, if in fact a contour segmentation is what you ultimately want).

The function prototype for `cv::pyrMeanShiftFiltering()` looks like this:

```
void cv::pyrMeanShiftFiltering(
  cv::InputArray   src,                      // 8-bit, 3-channel image
  cv::OutputArray  dst,                      // 8-bit, 3-channels, same size as src
  cv::double       sp,                       // spatial window radius
  cv::double       sr,                       // color window radius
  int              maxLevel = 1,             // max pyramid level
  cv::TermCriteria termcrit = TermCriteria(
    cv::TermCriteria::MAX_ITER | cv::TermCriteria::EPS,
    5,
    1
  )
);
```

In `cv::pyrMeanShiftFiltering()` we have an input image `src` and an output image `dst`. Both must be 8-bit, three-channel color images of the same width and height. The `spatialRadius` and `colorRadius` define how the mean-shift algorithm averages color and space together to form a segmentation. For a 640-by-480 color image, it works well to set `spatialRadius` equal to `2` and `colorRadius` equal to `40`. The next parameter of this algorithm is `max_level`, which describes how many levels of scale pyramid you want used for segmentation. A `max_level` of `2` or `3` works well for a 640-by-480 color image.

The final parameter is `cv::TermCriteria`, which we have seen in some previous algorithms. `cv::TermCriteria` is used for all iterative algorithms in OpenCV. The mean-shift segmentation function comes with good defaults if you just want to leave this parameter blank. Otherwise, `cv::TermCriteria` has the following constructor:

```
cv::TermCriteria(
    int    type;       // cv::TermCriteria::MAX_ITER, cv::TermCriteria::EPS, or both
    int    max_iter,   // maximum iterations when cv::TermCriteria::MAX_ITER is used
    double epsilon     // convergence value when cv::TermCriteria::EPS is used
);
```

Typically, we just use the `cv::TermCriteria()` constructor inline to generate the `cv::TermCriteria` object that we need. The first argument is either `cv::TermCriteria::MAX_ITER` or `cv::TermCriteria::EPS`, which tells the algorithm that we want to terminate either after some fixed number of iterations or when the convergence metric reaches some small value, respectively. The next two arguments set the values at which these criteria should terminate the algorithm. The reason we have both options is because we can set the type to

`cv::TermCriteria::MAX_ITER | cv::TermCriteria::EPS` to stop when either limit is reached. The parameter `max_iter` limits the number of iterations if `cv::TermCriteria::MAX_ITER` is set, whereas `epsilon` sets the error limit if `cv::TermCriteria::EPS` is set. Of course the exact meaning of `epsilon` depends on the algorithm.

Figure 6-25 shows an example of mean-shift segmentation using the following values:

```
cv::pyrMeanShiftFiltering( src, dst, 20, 40, 2);
```

Figure 6-25: Mean-shift segmentation over scale using `cv::pyrMeanShiftFiltering()` with parameters `max_level=2`, `spatialRadius=20`, and `colorRadius=40`; similar areas now have similar values and so can be treated as super pixels, which can speed up subsequent processing significantly

Image Repair

Images are often corrupted by noise. There may be dust or water spots on the lens, scratches on the older images, or parts of an image that were vandalized. *Inpainting* [Telea04] is a method for removing such damage by taking the color and texture at the border of the damaged area and propagating and mixing it inside the damaged area. See

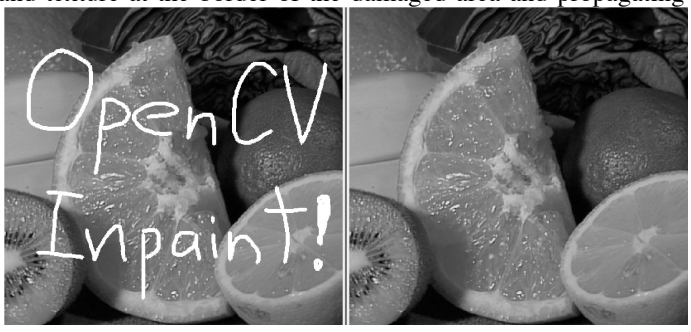


Figure 6-26 for an application that involves the removal of writing from an image.

Inpainting

Inpainting works provided the damaged area is not too “thick” and enough of the original texture and color remains around the boundaries of the damage.

Figure 6-27 shows what happens when the damaged area is too large.

Figure 6-26: Inpainting: an image damaged by overwritten text (left) is restored by inpainting (right)

The prototype for `cv::inpaint()` is

```
void cv::inpaint(
  cv::InputArray  src,                    // Input Image: 8-bit, 1 or 3 channels
  cv::InputArray  inpaintMask,            // 8-bit, 1 channel. Inpaint non-zeros
  cv::OutputArray dst,                    // Result image
  double          inpaintRadius,          // Range to consider around pixel
  int             flags                   // Select NS or TELEA
);
```

Figure 6-27: Inpainting cannot magically restore textures that are completely removed: the navel of the orange has been completely blotted out (left); inpainting fills it back in with mostly orangelike texture (right)

Here `src` *is an 8-bit single-channel grayscale image or a three-channel color image to be repaired, and* `inpaintMask` *is an 8-bit single-channel image of the same size as* `src` *in which the damaged areas (e.g., the writing seen in the left panel of*

Figure 6-26) have been marked by nonzero pixels; all other pixels are set to `0` in `inpaintMask`. The output image will be written to `dst`, which must have the same size and number of channels as `src`. The `inpaintRadius` is the area around each inpainted pixel that will be factored into the resulting output color of that pixel. As in

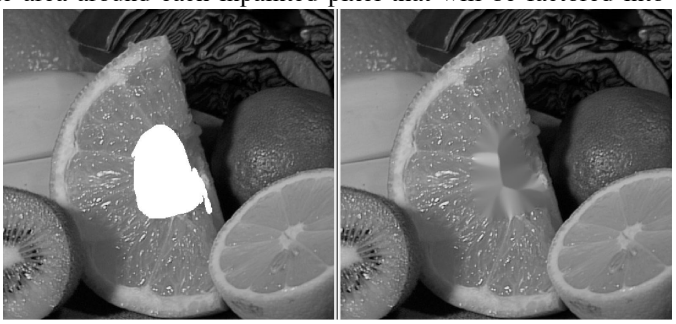


Figure 6-27, interior pixels within a thick enough inpainted region may take their color entirely from other inpainted pixels closer to the boundaries. Almost always, one uses a small radius such as `3` because too large a radius will result in a noticeable blur. Finally, the `flags` parameter allows you to experiment with two different methods of inpainting: `cv::INPAINT_NS` (Navier-Stokes method), and `cv::INPAINT_TELEA` (A. Telea's method).

Denoising

Another important problem that arises is the problem of noise in the image. In many applications, the most important source of noise is low-light conditions. In low light, the gain on the imager must be increased and the result is that noise is also amplified. The character of this kind of noise is typically random isolated pixels which appear either too bright or too dark, but discoloration is also possible in color images.

The denoising algorithm implemented in OpenCV is called Fast Non-local Means Denoising (FNLMD), and is based on work by Antoni Buades, Bartomeu Coll, and Jean-Michel Morel [Buades05]. While simple denoising algorithms essentially rely on averaging individual pixels with their neighbors, the central concept of FNLMD is to look for *similar pixels* elsewhere in the image, and average the among those. In this context a pixel is considered to be a similar pixel not because it is similar in color or intensity, but because it is similar in environment. The key logic here is that many images contain repetitive structures, and so even if your pixel is corrupted by noise, there will be many other pixels similar to it which are not.

The identification of similar pixels proceeds based on a window $B(p,s)$, centered on pixel p and of size s. Given such a window around the point we wish to update, we can compare that window with an analogous window around some other pixel q. We define the square-distance between $B(p,s)$ and $B(q,s)$ to be:

$$d^2\big(B(p,s),B(q,s)\big)=\frac{1}{3s^2}\sum_{c=1}^{3}\sum_{j\epsilon B(0,s)}\big(I_c(p+j)-I_c(q+j)\big)^2,$$

where c is the color index, $I_c(p)$ is the intensity of the image in channel c at point p, and the summation over j is over the elements of the patch. From this square-distance, a weight can be assigned to every other pixel relative to the pixel currently being updated. This weight is given by the formula:

$$w(p,q) = e^{-\frac{\max(d^2-2\sigma^2,0.0)}{h^2}}.$$

In this weight function, σ is the standard deviation expected in the noise (in intensity units), and h is a generic filtering parameter which determines how quickly patches will become irrelevant as their square-distance grows from the patch we are updating. In general, increasing the value of h will increase the noise removed but at the expense of some of the image detail. Decreasing the value if h will preserve detail, but also more of the noise.

Typically, there is a decreasing return in considering patches very far away (in pixel units) from the pixel being updated, as the number of such patches increases quadratically with the distance allowed. For this reason, a *search window* is defined and only patches in the search window contribute to the update. The update of the current pixel is then given by a simple weighted average of all other pixels in the search window using the given exponentially decaying weight function[41]. It is for this reason that the algorithm is called "non-local"; the patches which contribute to the repair of a given pixel are only loosely correlated to the location of the pixel being recomputed.

The OpenCV implementation of FNLMD has several different functions in it, each of which applies to slightly different circumstances.

Basic FNLMD with `cv::fastNlMeansDenoising()`

```
void cv::fastNlMeansDenoising(
  cv::InputArray         src,                    // Input image
  cv::OutputArray        dst,                    // Output image
  float                  h                  = 3, // Weight decay parameter
  int                    templateWindowSize = 7, // Size of patches used in comparison
  int                    searchWindowSize   = 21 // Max distance to patch considered
);
```

The first of these four functions, `cv::fastNlMeansDenoising()`, implements the algorithm as described exactly. The result array `dst` is computed from the input array `src` using a patch area of `templateWindowSize`, a decay parameter of `h`, and patches inside of `searchWindowSize` distance are considered. The image may be 1, 2, or 3 channel, but must be or type `cv::U8`.[42] Table 6-2 lists some values, provided from the authors of the algorithm, which can be used to help set the decay parameter, h.

Table 6-2: Recommended values for `cv::fastNlMeansDenoising()` for grayscale images.

Noise: σ	Patch Size: s	Search Window	Decay Parameter: h
$0 < \sigma \leq 15$	3 x 3	21 x 21	$0.40 \cdot \sigma$
$15 < \sigma \leq 30$	5 x 5	21 x 21	$0.40 \cdot \sigma$
$30 < \sigma \leq 45$	7 x 7	35 x 35	$0.35 \cdot \sigma$
$45 < \sigma \leq 75$	9 x 9	35 x 35	$0.35 \cdot \sigma$

[41] There is one subtlety here, which is that the weight of the contribution of the pixel p in its own recalculation would be $w(p,p) = e^0 = 1$. In general this results in too high a weight relative to other similar pixels and very little change occurs in the value at p. For this reason, the weight at p is normally chosen to be the maximum of the weights of the pixels within the area $B(p,s)$.

[42] Note that though this image allows for multiple channels, it is not the best way to handle color images. For color images, it is better to use `cv::fastNlMeansDenoisingColored()`.

$75 < \sigma \leq 100$	11 x 11	35 x 35	$0.30 \cdot \sigma$

FNLMD on Color Images with `cv::fastNlMeansDenoisingColored()`

```
void cv::fastNlMeansDenoisingColored(
  cv::InputArray         src,                   // Input image
  cv::OutputArray        dst,                   // Output image
  float                  h                = 3, // Weight decay parameter
  float                  hColor           = 3, // Weight decay parameter for color
  int                    templateWindowSize = 7, // Size of patches used in comparison
  int                    searchWindowSize   = 21 // Max distance to patch considered
);
```

The second variation of the FNLMD algorithm is used for color images. It accepts only images of type `cv::U8C3`. Though it would be possible in principle to apply the algorithm more or less directly to an RGB image, in practice it is better to convert the image to a different color space for the computation. The function `cv::fastNlMeansDenoisingColored()` first converts the image to the LAB color space, then applies the FNLMD algorithm, then converts the result back to RGB. The primary advantage of doing this is that in color there are, in effect, three decay parameters. In an RGB representation however it would be unlikely that you would want to set any of them to different values. In the LAB space however, it is natural to assign a different decay parameter to the luminosity component than to the color components. The function `cv::fastNlMeansDenoisingColored()` allows you to do just that. The parameter `h` is used for the luminosity decay parameter, while the new parameter `hColor` is used for the color channels. In general the value of `hColor` will be quite a bit smaller than `h`. In most contexts, `10` is a suitable value.

Table 6-3: Recommended values for `cv::fastNlMeansDenoising()` for color images.

Noise: σ	Patch Size: s	Search Window	Decay Parameter: h
$0 < \sigma \leq 25$	3 x 3	21 x 21	$0.55 \cdot \sigma$
$25 < \sigma \leq 55$	5 x 5	35 x 35	$0.40 \cdot \sigma$
$55 < \sigma \leq 100$	7 x 7	35 x 35	$0.35 \cdot \sigma$

FNLMD on Video with `cv::fastNlMeansDenoisingMulti()` and `cv::fastNlMeansDenoisingColoredMulti()`

```
void cv::fastNlMeansDenoisingMulti(
  cv::InputArrayOfArrays srcImgs,               // Sequence of several images
  cv::OutputArray        dst,                   // Output image
  int                    imgToDenoiseIndex,     // Index of image from srcImgs to denoise
  int                    temporalWindowSize,    // Number of images to use in denoising (odd)
  float                  h                = 3, // Weight decay parameter
  int                    templateWindowSize = 7, // Size of patches used for comparison
  int                    searchWindowSize   = 21 // Maximum distance to patch considered
);
void cv::fastNlMeansDenoisingColoredMulti(
  cv::InputArrayOfArrays srcImgs,               // Sequence of several images
  cv::OutputArray        dst,                   // Output image
  int                    imgToDenoiseIndex,     // Index of image from srcImgs to denoise
  int                    temporalWindowSize,    // Number of images to use in denoising (odd)
  float                  h                = 3, // Weight decay parameter
  float                  hColor           = 3, // Weight decay parameter for color dimensions
  int                    templateWindowSize = 7, // Size of patches used for comparison
  int                    searchWindowSize   = 21 // Maximum distance to patch considered
);
```

The third and fourth variations are used for sequential images, such as those which might be captured from video. In the case of sequential images, it is natural to imagine that other frames than just the current one might contain useful information for denoising a pixel. In most applications the noise will not be constant between images, while the signal will likely be similar or even identical. The functions `cv::fastNlMeansDenoisingMulti()` and `cv::fastNlMeansDenoisingColoredMulti()` expect an array of images, `srcImgs`, rather than a single image. Additionally, they must be told which image in the sequence is actually to be denoised; this is done with the parameter `imgToDenoiseIndex`. Finally, a temporal window must be provided which indicates the number of images from the sequence to be used in the denoising. This parameter must be odd, and the implied window is always centered on `imgToDenoiseIndex`. (Thus, if you were to set `imgToDenoiseIndex` to `4`, and `temporalWindowSize` to `5`, the images which would be used in the denoising would be `2`, `3`, `4`, `5`, and `6`.)

Summary

In this chapter, we learned a variety of methods that can be used to transform images. These transformations included scale transformations, as well as affine and perspective transformations. We learned how to remap vector functions from Cartesian to polar representations as well as how to perform discrete Fourier transformations. Finally, we learned how to perform edge transformations, including both low-level routines to extract edges on a pixel-by-pixel level, as well as higher-level routines which would integrate those edges into contours.

What all of these functions have in common is their conversion of one image into another through a global operation on the entire image.

Exercises

1. In this exercise, we learn to experiment with parameters by setting good `lowThresh` and `highThresh` values in `cv::Canny()`. Load an image with suitably interesting line structures. We'll use three different high:low threshold settings of 1.5:1, 2.75:1, and 4:1.
   a) Report what you see with a high setting of less than `50`.
   b) Report what you see with high settings between `50` and `100`.
   c) Report what you see with high settings between `100` and `150`.
   d) Report what you see with high settings between `150` and `200`.
   e) Report what you see with high settings between `200` and `250`.
   f) Summarize your results and explain what happens as best you can.
2. Load an image containing clear lines and circles such as a side view of a bicycle. Use the Hough line and Hough circle calls and see how they respond to your image.
3. Can you think of a way to use the Hough transform to identify any kind of shape with a distinct perimeter? Explain how.
4. Get a product or print out a 1D barcode (having a series of parallel vertical lines in a row) from the web. Use the Line Segment Detector algorithm to create a 1D barcode detector.
5. Look at the diagrams of how the log-polar function transforms a square into a wavy line.
   a) Draw the log-polar results if the log-polar center point were sitting on one of the corners of the square.
   b) What would a circle look like in a log-polar transform if the center point were inside the circle and close to the edge?
   c) Draw what the transform would look like if the center point were sitting just outside of the circle.

6. A log-polar transform takes shapes of different rotations and sizes into a space where these correspond to shifts in the θ-axis and $\log(r)$ axis. The Fourier transform is translation invariant. How can we use these facts to force shapes of different sizes and rotations to automatically give equivalent representations in the log-polar domain?
7. Draw separate pictures of large, small, large rotated, and small rotated squares. Take the log-polar transform of these each separately. Code up a two-dimensional shifter that takes the center point in the resulting log-polar domain and shifts the shapes to be as identical as possible.
8. Take the Fourier transform of a small Gaussian distribution and the Fourier transform of an image. Multiply them and take the inverse Fourier transform of the results. What have you achieved? As the filters get bigger, you will find that working in the Fourier space is much faster than in the normal space.
9. Load an interesting image, convert it to grayscale, and then take an integral image of it. Now find vertical and horizontal edges in the image by using the properties of an integral image.

   Use long skinny rectangles; subtract and add them in place.
10. A good computer vision programming interview question is to code up an integral image. Let's do that:
    - Write a routine that takes an image and returns an integral image.
    - Write a routine that takes an image and returns a 45 degree rotated integral image.
    - Can you write an integral image for 22.5 degree rotated images? If so, how?
11. Explain how you could use the distance transform to automatically align a known shape with a test shape when the scale is known and held fixed. How would this be done over multiple scales?
12. Practice histogram equalization on images that you load in, and report the results.
13. Load an image, take a perspective transform, and then rotate it. Can this transform be done in one step? In the 2005 DARPA Grand Challenge robot race, the authors on the Stanford team used a kind of color clustering algorithm to separate road from non-road. The colors were sampled from a laser-defined trapezoid of road patch in front of the car. Other colors in the scene that were close in color to this patch—and whose connected component connected to the original trapezoid—were labeled as road. See Figure 6-28, where the watershed algorithm was used to segment the road after using a trapezoid mark inside the road and an inverted "U" mark outside the road. Suppose we could automatically generate these marks. What could go wrong with this method of segmenting the road?

    Hint: Look carefully at Figure 6-28 and then consider that we are trying to extend the road trapezoid by using things that look like what's in the trapezoid.

Figure 6-28: Using the watershed algorithm to identify a road: markers are put in the original image (left), and the algorithm yields the segmented road (right)

14. In-painting works pretty well for the repair of writing over textured regions. What would happen if the writing obscured a real object edge in a picture? Try it.
15. How well does inpainting work at fixing up writing drawn over a mean-shift segmented image? Try it for various settings and show the results.
16. Take an image and add noise to it in the following ways:

- Randomly add uniform noise of successively higher ranges. Run the function `cv::fastNlMeansDenoisingColored()` on it. Report the results.
- Do the same, but use Gaussian noise instead.
- Do the same, but add *shot noise*. Change the value of every "N^{th}" pixel. For example, every 1000^{th} pixel, 500^{th}, 100^{th}, 50^{th}, and 10^{th}.

7

Histograms and Matching

In the course of analyzing images, objects, and video information, we frequently want to represent what we are looking at as a *histogram*. Histograms can be used to represent such diverse things as the color distribution of an object, an edge gradient template of an object [Freeman95], and the distribution of probabilities representing our current hypothesis about an object's location. Figure 7-1 shows the use of histograms for rapid gesture recognition. Edge gradients were collected from "up," "right," "left," "stop" and "OK" hand gestures. A webcam was then set up to watch a person who used these gestures to control web videos. In each frame, color interest regions were detected from the incoming video; then edge gradient directions were computed around these interest regions, and these directions were collected into orientation bins within a histogram. The histograms were then matched against the gesture models to recognize the gesture. The vertical bars in Figure 7-1 show the match levels of the different gestures. The gray horizontal line represents the threshold for acceptance of the "winning" vertical bar corresponding to a gesture model.

Histograms find uses in many computer vision applications. Histograms are used to detect scene transitions in videos by marking when the edge and color statistics markedly change from frame to frame. They are used to identify interest points in images by assigning each interest point a "tag" consisting of histograms of nearby features. Histograms of edges, colors, corners, and so on form a general feature type that is passed to classifiers for object recognition. Sequences of color or edge histograms are used to identify whether videos have been copied on the web, where scenes change in a movie, in image retrieval from massive databases, and the list goes on. Histograms are one of the classic tools of computer vision.

Histograms are simply collected *counts* of the underlying data organized into a set of predefined *bins*. They can be populated by counts of features computed from the data, such as gradient magnitudes and directions, color, or just about any other characteristic. In any case, they are used to obtain a statistical picture of the underlying distribution of data. The histogram usually has fewer dimensions than the source data. Figure 7-2 depicts a typical situation. The figure shows a two-dimensional distribution of points (upper-left); we impose a grid (upper-right) and count the data points in each *grid cell*, yielding a one-dimensional histogram (lower-right). Because the raw data points can represent just about anything, the histogram is a handy way of representing whatever it is that you have learned from your image.

Figure 7-1: Local histograms of gradient orientations are used to find the hand and its gesture; here the "winning" gesture (longest vertical bar) is a correct recognition of "L" (move left)

Histograms that represent continuous distributions do so by quantizing the points into each grid cell.[1] This is where problems can arise, as shown in Figure 7-3. If the grid is too wide (upper-left), then the output is too coarse and we lose the structure of the distribution. If the grid is too narrow (upper-right), then there is not enough averaging to represent the distribution accurately and we get small, "spiky" cells.

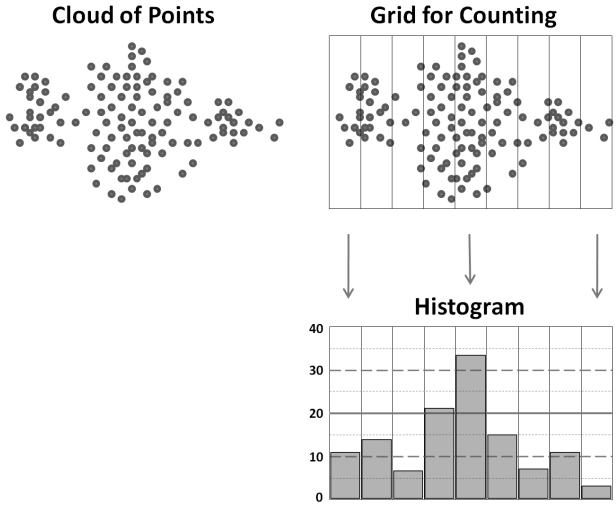


Figure 7-2: Typical histogram example: starting with a cloud of points (upper-left), a counting grid is imposed (upper-right) that yields a one-dimensional histogram of point counts (lower-right)

OpenCV has a data type for representing histograms. The histogram data structure is capable of representing histograms in one or many dimensions, and it contains all the data necessary to track bins of both uniform and non-uniform sizes. And, as you might expect, it comes equipped with a variety of useful functions that allow us to easily perform common operations on our histograms.

---

[1] This is also true of histograms representing information that falls naturally into discrete groups when the histogram uses fewer bins than the natural description would suggest or require. An example of this is representing 8-bit intensity values in a 10-bin histogram: each bin would then combine the points associated with approximately 25 different intensities, (erroneously) treating them all as equivalent.

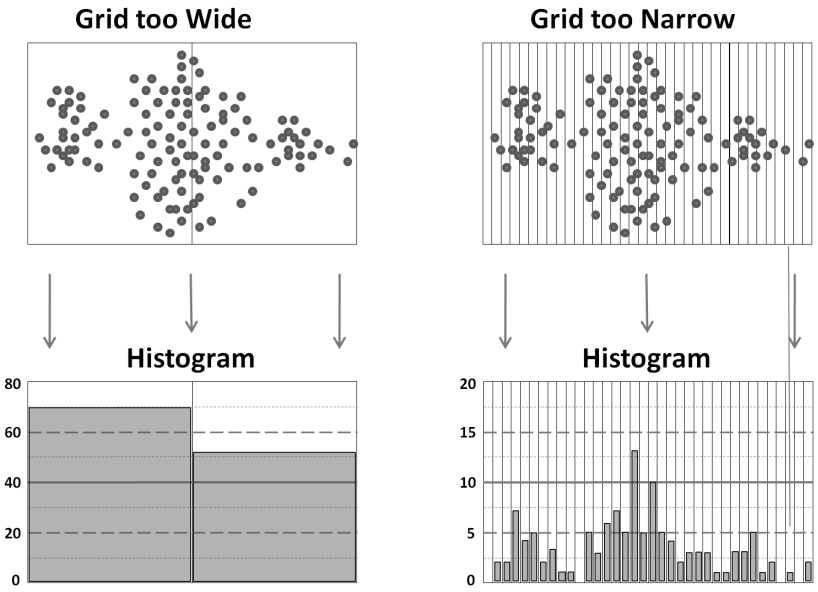


Figure 7-3: A histogram's accuracy depends on its grid size: a grid that is too wide yields too coarse quantization in the histogram values (left); a grid that is too small yields "spiky" and singleton results from too small samples (right).

Histogram Representation in OpenCV

Histograms are represented in OpenCV as arrays, using the same array structures as are used for other data.[2] This means that you can use `cv::Mat`, if you have a one- or two-dimensional array (with the array being *N*-by-1 or 1-by-*N* in the former case), `vector<>` types, or sparse matrices. Of course, the interpretation of the array is different, even if the underlying code is identical. For an n-dimensional array, the interpretation is an n-dimensional array of histogram bins, in which the value for any particular element represents the number of counts in that particular bin. This distinction is important in the sense that bins, being just indices into an array of some dimensionality, are simple integers. The identity of the bin, i.e., what it represents, is separate from the bin's integer index. Whenever you are working with histograms, you will find yourself needing to handle the conversion between measured values and histogram bin indices. Many OpenCV functions will do this task, or some part of this task, for you.

When working with higher-dimensional histograms, often most of the entries in that histogram are zero. The `cv::SparseMat` class is very good for representing such cases. In fact, histograms are the primary reason for the existence of the `cv::SparseMat` class. Most functions that work on dense arrays will also work on sparse arrays, but we will touch on a few important exceptions in the next section.

`cv::calcHist()`, Creating a Histogram from Data

The function `cv::calcHist()` computes the bin values for a histogram from one or more arrays of data. Recall that the dimensions of the histogram are not related to the dimensionality of the input arrays, or their size, but rather to their number. Each dimension of the histogram represents counting (and binning) across all pixels values in one of the channels of one of the input arrays. You are not required to use every channel in every image, however; you can pick whatever subset you would like of the channels of the arrays passed to `cv::calcHist()`.

---

[2] This is a substantial change in the C++ API relative to the C API. In the latter, there is a specific structure called a `CvHistogram` for representing histogram data. The elimination of this structure in the C++ interface creates a much simpler more unified library.

```
void cv::calcHist(
  const cv::Mat*  images,            // C-style array of images, 8U or 32F
  int             nimages,           // number of images in 'images' array
  const int*      channels,          // C-style list of int's identifying channels
  cv::InputArray  mask,              // pixels in 'images' count iff 'mask' nonzero
  cv::OutputArray hist,              // output histogram array
  int             dims,              // histogram dimensionality < cv::MAX_DIMS (32)
  const int*      histSize,          // C-style array, histogram sizes in each dim
  const float**   ranges,            // C-style array of 'dims' pairs set bin sizes
  bool            uniform    = true, // true for uniform binning
  bool            accumulate = false // if true, add to 'hist' rather than replacing
);

void cv::calcHist(
  const cv::Mat*  images,            // C-style array of images, 8U or 32F
  int             nimages,           // number of images in 'images' array
  const int*      channels,          // C-style list of int's identifying channels
  cv::InputArray  mask,              // pixels in 'images' count iff 'mask' nonzero
  cv::SparseMat&  hist,              // output histogram (sparse) array
  int             dims,              // histogram dimensionality < cv::MAX_DIMS (32)
  const int*      histSize,          // C-style array, histogram sizes in each dim
  const float**   ranges,            // C-style array of 'dims' pairs set bin sizes
  bool            uniform    = true, // true for uniform binning
  bool            accumulate = false // if true, add to 'hist' rather than replacing
);
```

There are three forms of the `cv::calcHist()` function, two of which use "old-fashioned" C-style arrays, while the third uses the now preferred STL vector template type arguments. The primary distinction between the first two is whether the computed results are to be organized into a dense or a sparse array.

The first arguments are the array data, with `images` being either a pointer to a C-style list of arrays or the more modern `cv::InputArrayOfArrays`. In either case, the role of `images` is to contain one or more arrays from which the histogram will be constructed. All of these arrays must be the same size, but each can have any number of channels. These arrays may be 8-bit integers or of 32-bit floating point type, but the type of all of the arrays must match. In the case of the C-style array input, the additional argument `narrays` indicates the number of arrays pointed to by `images`. The argument `channels` indicates which channels to consider when creating the histogram. Once again, `channels` may be a C-style array or an STL vector of integers. These integers identify which channels from the input arrays are to be used for the output histogram. The channels are numbered sequentially, so the first $N_c^{(0)}$ channels in `images[0]` are numbered zero through $N_c^{(0)} - 1$, while the next $N_c^{(1)}$ channels in `images[1]` are numbered $N_c^{(0)}$ through $N_c^{(0)} + N_c^{(1)} - 1$, and so on. Of course, the number of entries in `channels` is equal to the number of dimensions of the histogram you are creating.

The array `mask` is an optional mask which, if present, will be used to select which pixels of the arrays in `images` will contribute to the histogram. `mask` must be an 8-bit array and the same size as the input arrays. Any pixel in `images` corresponding to a nonzero pixel in `mask` will be counted. If you do not wish to use a mask, you can pass `cv::noArray()` instead.

The argument `hist` is the output histogram you would like to fill. The argument `dims` is the number of dimensions that histogram will have. Recall that `dims` is also the number of entries in the `channels` array, indicating how each dimension is to be computed. The argument `histSize` may be a C-style array or an STL style vector of integers and indicates the number of bins that should be allocated in each dimension of `hist`. The number of entries in `histSize` must also be equal to `dims`.

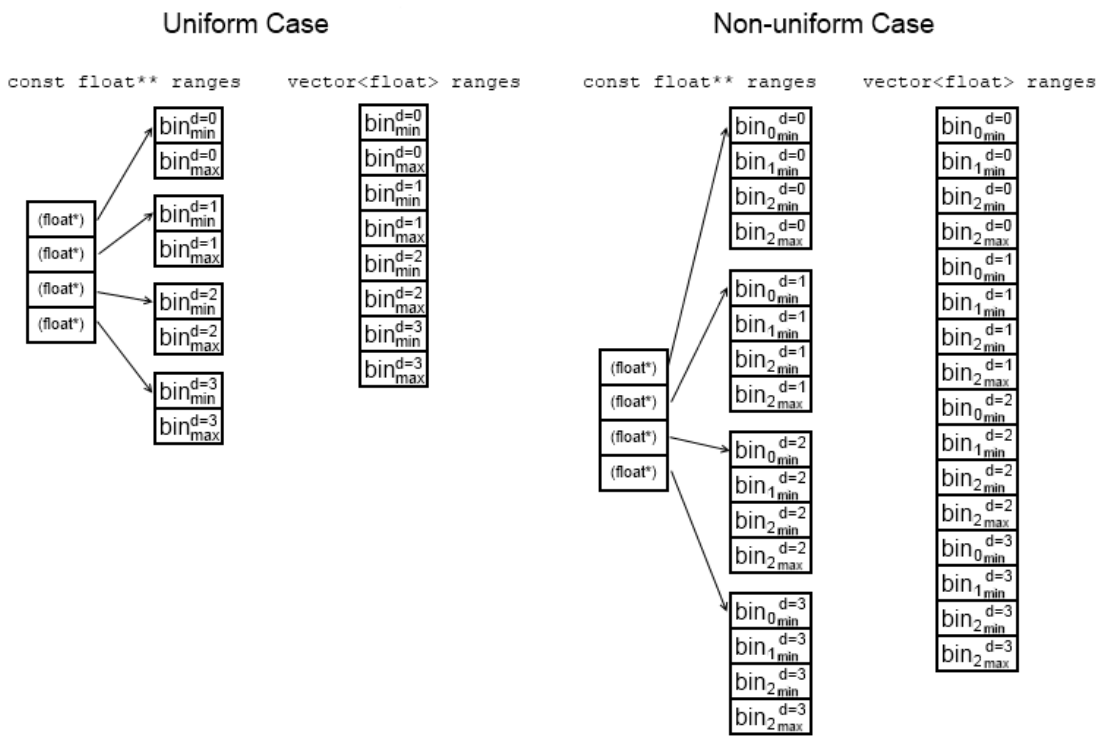


Figure 7-4: The ranges argument may be either a C-style array of arrays, or a single STL style vector of floating point numbers. In the case of a uniform histogram, only the minimum and maximum bin edges must be supplied. For a non-uniform histogram, the lower edge of each bin in each dimension must be supplied, as well as the maximum value.

While `histSize` indicates the number of bins in each dimension, `ranges` indicates the values that correspond to each bin in each dimension. `ranges` also can be a C-style array or an STL vector. In the C-style array case, each entry `ranges[i]` is another array and the length of ranges must be equal to `dims`. In this case, entry `ranges[i]` indicates the bin structure of the corresponding i^{th} dimension. How `ranges[i]` is interpreted depends on the value of the argument `uniform`. If `uniform` is `true`, then all of the bins in the i^{th} dimension are of the same size, and all that is needed is to specify the (inclusive) lower bound of the lowest bin and the (non-inclusive) upper bound of the highest bin (e.g., `ranges[i] = {0,100.0}`). If, on the other hand, `uniform` is `false`, then if there are N_i bins in the i^{th} dimension, there must be $N_i + 1$ entries in `ranges[i]`. Thus, the j^{th} entry will be interpreted as the (inclusive) lower bound of bin j and the (non-inclusive) upper bound of bin $(j - 1)$. In the case in which `ranges` is of type `vector<float>`, the entries having the same meaning as the C-style array values, but here they are "flattened" into one single-level array (i.e., for the uniform case, there are just two entries in ranges per histogram dimension and they are in the order of the dimensions, while for the non-uniform case, there will be $N_i + 1$ entries per dimension, and they are again all in the order of the dimensions).

The final argument `accumulate` is used to tell OpenCV that the array `hist` is not to be deleted, reallocated, or otherwise set to zero before adding new counts from the arrays in `images`.

Basic Manipulations with Histograms

Even though the data structure for the histogram is the same as the data structure used for matrices and image arrays, this particular interpretation of the data structure invites new operations on these arrays that accomplish tasks specific to histograms. In this section, we will touch on some simple operations that are specific to histograms, as well as review how some important histogram manipulations can be performed with array operations we have already discussed in prior chapters.

Histogram Normalization

When dealing with a histogram, we first need to accumulate information into its various bins. Once we have done this, however, it is often desirable to work with the histogram in *normalized form*, so that individual bins will represent the fraction of the total number of events assigned to the entire histogram. In the C++ API, this can be accomplished simply using the array algebra operators and operations:

```
cv::Mat normalized = my_hist / my_hist.Sum();
```

or:

```
my_hist /= my_hist.Sum()
```

Histogram Threshold

It is also common that you wish to threshold a histogram, and (for example) throw away all bins whose entries contain less than some minimum number of counts. Like normalization, this operation can be accomplished without the use of any particular special histogram routine. Instead, you can use the standard array threshold function:

```
cv::threshold(
  my_hist,                          // input histogram
  my_thresholded_hist,              // result with all values<threshold set to zero
  threshold,                        // cutoff value
  0,                                // value does not matter in this case
  cv::THRESH_TOZERO                 // threshold type
);
```

Finding the Most Populated Bin

In some cases, you would like to find the bins that are above some threshold, and throw away the others. In other cases, you would like to simply find the one bin that has the most weight in it. This is particularly common when the histogram is being used to represent a probability distribution. In this case, the function `cv::minMaxLoc()` will give you what you want.

In the case of a two-dimensional array, you can use the `cv::InputArray` form of `cv::minMaxLoc()`:

```
void cv::minMaxLoc(
  cv::InputArray src,                  // Input array
  double*        minVal,               // fill with minimum value (if not NULL)
  double*        maxVal = 0,           // fill with maximum value (if not NULL)
  cv::Point*     minLoc = 0,           // fill with minimum location (if not NULL)
  cv::Point*     maxLoc = 0,           // fill with maximum location (if not NULL)
  cv::InputArray mask   = cv::noArray() // ignore points for which mask is zero
);
```

The arguments `minVal` and `maxVal` are pointers to locations you provide for `cv::minMaxLoc()` to store the minimum and maximum values that have been identified. Similarly, `minLoc` and `maxLoc` are pointers to variables (of type `cv::Point`, in this case) where the actual locations of the minimum and maximum can be stored. If you do not want one or more of these four results to be computed, you can simply pass `NULL` for that (pointer) variable and that information will not be computed:

```
double    max_val;
cv::Point max_pt;

cv::minMaxLoc(
  my_hist,    // input histogram
  NULL,       // don't care about the min value
  &max_val,   // place to put the maximum value
  NULL,       // don't care about the location of the min value
  &max_pt     // place to put the maximum value location (a cv::Point)
);
```

In this example, though, the histogram would need to be two-dimensional.[3] If your histogram is of sparse array type, then there is no problem. Recall that there is an alternate form of `cv::minMaxLoc()` for sparse arrays:

```
void cv::minMaxLoc(
  const cv::SparseMat& src,                    // Input (sparse) array
  double*              minVal,                 // fill minimum value (if not NULL)
  double*              maxVal = 0,             // fill maximum value (if not NULL)
  cv::Point*           minLoc = 0,             // fill minimum location (if not NULL)
  cv::Point*           maxLoc = 0,             // fill maximum location (if not NULL)
  cv::InputArray       mask   = cv::noArray() // ignore points if mask is zero
);
```

It should be noted that this form of `cv::minMaxLoc()` actually differs from the previous form in several ways. In addition to taking a sparse matrix as the source, it also takes type `int*` for the `minIdx` and `maxIdx` variables instead of `cv::Point*` for the analogous `minLoc` and `maxLoc` variables. This is because the sparse matrix form of `cv::minMaxLoc()` supports arrays of any dimensionality. Therefore, you need to allocate the location variables yourself and make sure that they have the correct amount of space available for the *n*-dimensional index associated with a point in the (*n*-dimensional) sparse histogram:

```
double maxval;
int*   max_pt = new int[ my_hist.dims() ];

cv::minMaxLoc(
  my_hist,                 // input sparse histogram
  NULL,                    // don't care about the min value
  &max_val,                // place to put the maximum value
  NULL,                    // don't care about the location of the min value
  &max_pt                  // place to put the maximum value location (a cv::Point)
);
```

It turns out that if you want to find the minimum or maximum of an n-dimensional array that is not sparse, you need to use another function. This function works essentially the same as `cv::minMaxLoc()`, and has a similar name, but is not quite the same creature:

```
void cv::minMaxIdx(
  cv::InputArray src,
  double*        minVal,                // will put minimum value (if not NULL)
  double*        maxVal = 0,            // will put maximum value (if not NULL)
  int*           minLoc = 0,            // will put min location indices (if not NULL)
  int*           maxLoc = 0,            // will put max location indices (if not NULL)
  cv::InputArray mask   = cv::noArray() // ignore points if mask is zero
);
```

In this case, the arguments have the same meanings as the corresponding arguments in the two forms of `cv::minMaxLoc()`. You must allocate `minIdx` and `maxIdx` to C-style arrays of the correct size yourself (as before). One word of warning is in order here, however. If the input array `src` is one dimensional, you should allocate `minIdx` and `maxIdx` to be of dimension two. The reason for this is that `cv::minMaxIdx()` treats a one-dimensional array as a two-dimensional array internally. As a result, if the maximum is found at position k, the return value for `maxIdx` will be `(k,0)` for a single-column matrix and `(0,k)` for a single-row matrix.

Comparing Two Histograms

Yet another indispensable tool for working with histograms, first introduced by Swain and Ballard [Swain91] and further generalized by Schiele and Crowley [Schiele96], is the ability to compare two

[3] If you have a one-dimensional `vector<>` array, you can just use `cv::Mat( vec ).reshape(1)` to make it a *N*-by-1 array in two dimensions.

histograms in terms of some specific criteria for similarity. The function `cv::compareHist()` does just this.

```
double cv::compareHist(
  cv::InputArray       H1,        // First histogram to be compared
  cv::InputArray       H2,        // Second histogram to be compared
  int                  method     // method for the comparison (see options below)
);

double cv::compareHist(
  const cv::SparseMat& H1,        // First histogram to be compared
  const cv::SparseMat& H2,        // Second histogram to be compared
  int                  method     // method for the comparison (see options below)
);
```

The first two arguments are the histograms to be compared, which should be of the same size. The third argument is where we select our desired distance metric. The four available options are as follows.

Correlation (method = cv::COMP_CORREL)

The first comparison is based on statistical correlation; it implements the Pearson Correlation Coefficient, and is typically appropriate when H_1 and H_2 can be interpreted as probability distributions.

$$d_{correl}(H_1, H_2) = \frac{\sum_i H_1'(i) \cdot H_2'(i)}{\sqrt{\sum_i {H_1'}^2(i) \cdot \sum_i {H_2'}^2(i)}}$$

Here, $H_k'(i) = H_k(i) - N^{-1} \sum_j h_k(j)$ and N is equal to the number of bins in the histogram.

For *correlation*, a high score represents a better match than a low score. A perfect match is `1` and a maximal mismatch is `-1`; a value of `0` indicates no correlation (random association).

Chi-square (method = cv::COMP_CHISQR)

For *chi-square*,[4] distance metric is based on the chi-squared test statistic, which is a test as to whether or not two distributions are in fact correlated.

$$d_{chi-square}(H_1, H_2) = \sum_i \frac{\left(H_1(i) - H_2(i)\right)^2}{H_1(i) + H_2(i)}$$

For this test, a low score represents a better match than a high score. A perfect match is 0 and a total mismatch is unbounded (depending on the size of the histogram).

Intersection (method = cv::COMP_INTERSECT)

The *histogram intersection* method is based on a simple intersection of the two histograms. This means that it ask, in effect, what do these two have in common, and sums over all of the bins of the histograms.

$$d_{intersection}(H_1, H_2) = \sum_i \min\left(H_1(i), H_2(i)\right)$$

For this metric, high scores indicate good matches and low scores indicate bad matches. If both histograms are normalized to `1.0`, then a perfect match is `1.0` and a total mismatch is `0.0`.

Bhattacharyya distance (method = cv::COMP_BHATTACHARYYA)

The last option, called the *Bhattacharyya distance* [Bhattacharyya43], is also a measure of overlap of two distributions.

[4] The chi-square test was invented by Karl Pearson [Pearson] who founded the field of mathematical statistics.

$$d_{correl}(H_1, H_2) = \sqrt{1 - \frac{\sum_i H_1(i) \cdot H_2(i)}{\sqrt{\sum_i H_1(i) \sum_i H_2(i)}}}$$

In this case, low scores indicate good matches and high scores indicate bad matches. A perfect match is `0.0` and a total mismatch is a `1.0`.

With `cv::COMP_BHATTACHARYYA`, a special factor in the code is used to normalize the input histograms. In general, however, you should normalize histograms *before* comparing them, because concepts like histogram intersection make little sense (even if allowed) without normalization.

The simple case depicted in Figure 7-5 should help clarify matters using about the simplest case imaginable: a one-dimensional histogram with only two bins. The model histogram has a `1.0` value in the left bin and a `0.0` value in the right bin. The last three rows show the comparison histograms and the values generated for them by the various metrics (the EMD metric will be explained shortly).

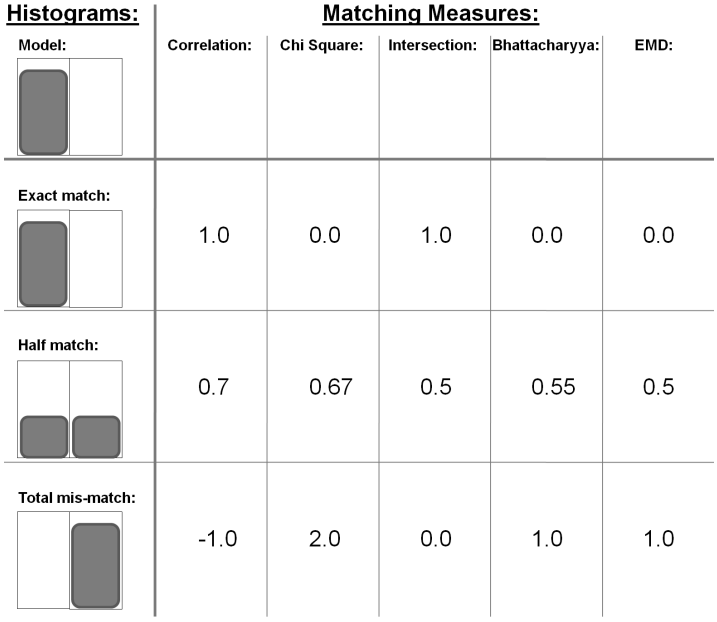


Figure 7-5: Histogram matching measures

Figure 7-5 provides a quick reference for the behavior of different matching types. Close inspection of these matching algorithms in the figure will reveal a disconcerting observation. If histogram bins shift by just one slot, as with the chart's first and third comparison histograms, then all these matching methods (except EMD) yield a maximal mismatch even though these two histograms have a similar "shape." The rightmost column in Figure 7-5 reports values returned by EMD, a type of distance measure. In comparing the third to the model histogram, the EMD measure quantifies the situation precisely: the third histogram has moved to the right by one unit. We shall explore this measure further in the "Earth Mover's Distance" section to follow.

In the authors' experience, intersection works well for quick-and-dirty matching and chi-square or Bhattacharyya work best for slower but more accurate matches. The EMD measure gives the most intuitive matches but is much slower.

Histogram Usage Examples

It's probably time for some helpful examples. The program in Example 7-1 (adapted from the OpenCV code bundle) shows how we can use some of the functions just discussed. This program computes a hue-saturation histogram from an incoming image and then draws that histogram as an illuminated grid.

Example 7-1: Histogram computation and display

```
#include <opencv2/opencv.hpp>
#include <iostream>

using namespace std;

int main( int argc, char** argv ){

  if(argc != 2) {
    cout << "Computer Color Histogram\nUsage: ch7_ex7_1 <imagename>" << endl;
    return -1;
  }
  cv::Mat src = cv::imread(argv[1],1);

  if( src.empty() ) { cout << "Can not load " << argv[1] << endl; return -1; }

  // Compute the HSV image, and decompose it into separate planes.
  //
  cv::Mat hsv;
  cv::cvtColor(src, hsv, cv::BGR2HSV);

  float h_ranges[]      = {0, 180}; // hue is [0, 180]
  float s_ranges[]      = {0, 255};
  const float* ranges[] = {h_ranges, s_ranges};
  int histSize[]        = {20, 2}, ch[] = {0, 1};

  cv::Mat hist;

  // Compute the histogram
  cv::calcHist(&hsv, 1, ch, cv::noArray(), hist, 2, histSize, ranges, true);
  cv::normalize(hist, hist, 0, 255, cv::NORM_MINMAX);

  int scale = 10;
  cv::Mat hist_img(histSize[0]*scale, histSize[1]*scale, CV::U8C3);

  // Draw our histogram.
  for( int h = 0; h < histSize[0]; h++ ) {
    for( int s = 0; s < histSize[1]; s++ ){
      float hval = hist.at<float>(h, s);
      cv::rectangle(
        hist_img,
        cv::Rect(h*scale,s*scale,scale,scale),
        cv::Scalar::all(hval),
        -1
      );
    }
  }

  cv::imshow("image", src);
  cv::imshow("H-S histogram", hist_img);
  cv::waitKey();
  return 0;
}
```

In this example, we have spent a fair amount of time preparing the arguments for `cv::calcHist()`, which is not uncommon.

In many practical applications, it is useful to consider the color histograms associated with human skin tone. By way of example, we have histograms taken from a human hand under various lighting conditions in Figure 7-6. The left column shows images of a hand in an indoor environment, a shaded outdoor environment, and a sunlit outdoor environment. In the middle column are the blue, green, and red (BGR) histograms corresponding to the observed flesh tone of the hand. In the right column are the corresponding

HSV histograms, where the vertical axis is *V* (value), the radius is *S* (saturation) and the angle is *H* (hue). Notice that indoors is the darkest, outdoors in shadow in is a bit brighter, and outdoors in the sun is the brightest. Note also that the colors shift around somewhat as a result of the changing color of the illuminating light.

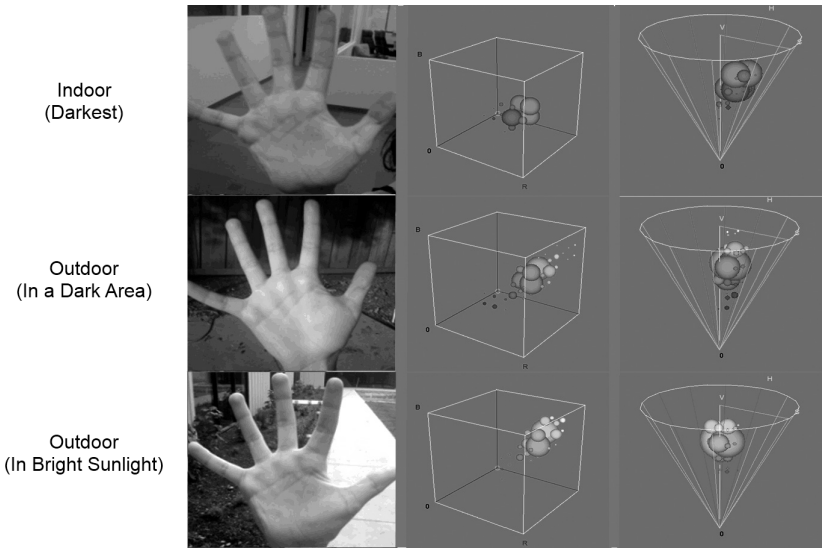


Figure 7-6: Histogram of flesh colors under indoor (upper-left), shadowed outdoor (middle left), and direct sun outdoor (lower-left) lighting conditions; the middle and right-hand columns display the associated BGR and HSV histograms, respectively

As a test of histogram comparison, we could take a portion of one palm (e.g., the top half of the indoor palm), and compare the histogram representation of the colors in that image either with the histogram representation of the colors in the remainder of that image or with the histogram representations of the other two hand images. To make a lower dimensional comparison, we use only hue (*H*) and saturation (*S*) from an HSV color space.

Table 7-1: Histogram comparison, via four matching methods, of palm-flesh colors in upper half of indoor palm with listed variant palm-flesh color. We used 30 bins for hue, and 32 for saturation. For reference, the expected score for a perfect match is provided in the first row.

Comparison	**CORREL**	**CHISQR**	**INTERSECT**	**BHATTACHARYYA**
(Perfect Match)	(1.0)	(0.0)	(1.0)	(0.0)
Indoor lower half	0.96	0.14	0.82	0.2
Outdoor shade	0.09	1.57	0.13	0.8
Outdoor sun	–0.0	1.98	0.01	0.99

To put this experiment into practice (see code later in Example 7-2), we take three images of a hand under different lighting conditions (Figure 7-6). First we construct a histogram from the hand portion of the top image (the dark one), which we will use as our reference. We then compare that histogram to a histogram taken from the hand in the bottom half of that same image, and then to the hands that appear in the next two (whole) images. The first image is an indoor image, the latter two are outdoors. Using 30 bins for hue and 32 for saturation, the matching results are shown in Table 7-1, in which several things are apparent. First, the distance metrics even with the lower half of the same indoor image are hardly perfect. Second, some of the distance metrics return a small number when the distance is small, and a larger number when it is high, while other metrics do the opposite. This is as we should have expected from the simple analysis of the matching measures shown in Figure 7-5. The recognition results in Example 7-1 are not good however because, referencing Figure 7-6, we can see that the distribution in H-S space shifts location, especially in

saturation and so, with fine grain bins, the bins will shift (similar to the last row of Figure 7-5). When we use only 2 bins for saturation and `20` for hue, we get better results as shown in Table 7-2.

Table 7-2: Histogram comparison, same images as Table 7-1, but 2 bins for saturation, `20` for hue.

Comparison	CORREL	CHISQR	INTERSECT	BHATTACHARYYA
(Perfect Match)	(1.0)	(0.0)	(1.0)	(0.0)
Indoor lower half	0.99	0.06	0.93	0.08
Outdoor shade	0.43	19.12	0.41	0.56
Outdoor sun	0.7	147.36	0.98	0.55

Some More Complicated Stuff

Everything we've discussed so far was reasonably basic. Each of the functions provided for a relatively obvious need. Collectively, they form a good foundation for much of what you might want to do with histograms in the context of computer vision (and probably in other contexts as well). At this point we want to look at some more complicated routines available within OpenCV that are extremely useful in certain applications. These routines include a more sophisticated method of comparing two histograms as well as tools for computing and/or visualizing which portions of an image contribute to a given portion of a histogram.

Earth Mover's Distance

We saw earlier that lighting changes can cause significant shifts in color values (see

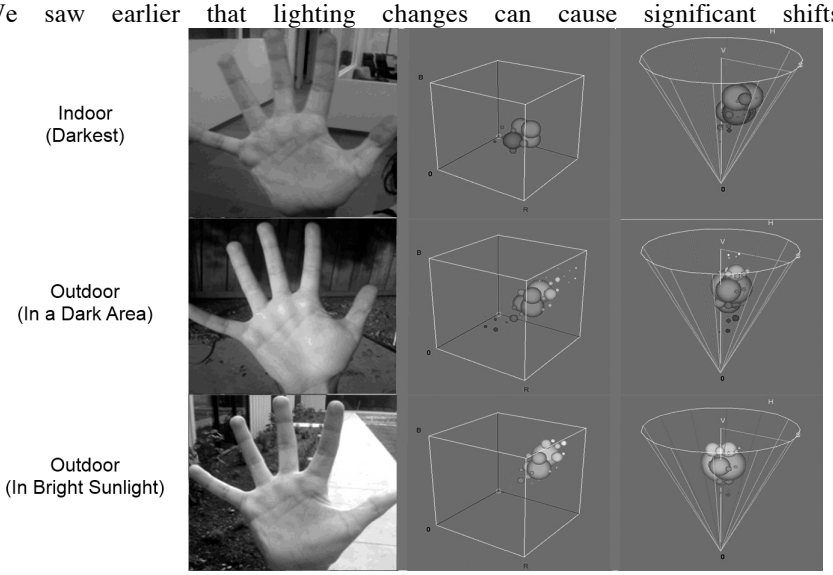


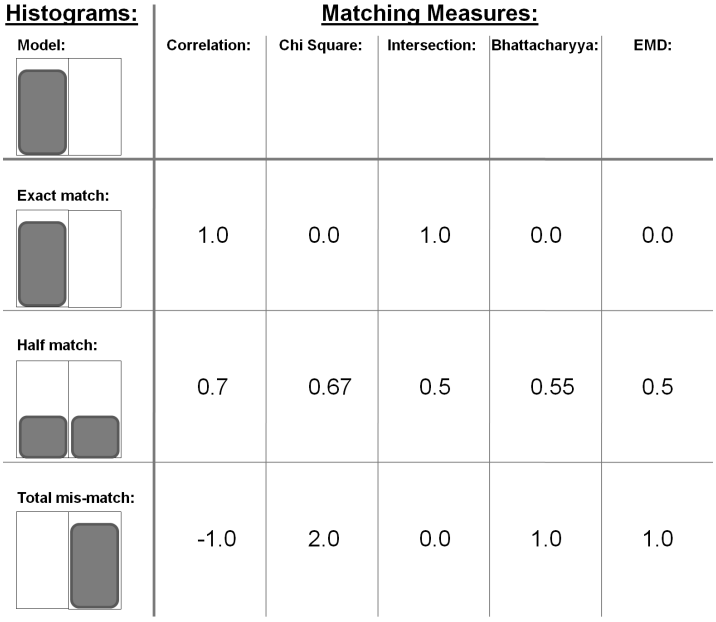
Figure 7-6), although such shifts tend not to change the shape of the histogram of color values, but shift the color value locations and thus cause the histogram-matching schemes we've learned about to fail. The difficulty with histogram *match* measures is that they can return a large difference in the case where two histograms are similarly shaped, but only displaced relative to one another. It is often desirable to have a *distance* measure that performs like a match, but is less sensitive to such displacements. Earth mover's distance (EMD) [Rubner00] is such a metric; it essentially measures how much work it would take to "shovel" one histogram shape into another, including moving part (or all) of the histogram to a new location. It works in any number of dimensions.

Return again to Figure 7-5; we see the "earth shoveling" nature of EMD's distance measure in the right-most column. An exact match is a distance of `0.0`. Half a match is half a "shovel full," the amount it would take to spread half of the left histogram into the next slot. Finally, moving the entire histogram one step to the right would require an entire unit of distance (i.e., to change the model histogram into the "totally mismatched" histogram).

The EMD algorithm itself is quite general; it allows users to set their own distance metric or their own cost-of-moving matrix. One can record where the histogram "material" flowed from one histogram to another, and one can employ nonlinear distance metrics derived from prior information about the data. The EMD function in OpenCV is `cv::EMD()`:

```
float cv::EMD(
  cv::InputArray  signature1,             // size1-by-(dims+1) float array
  cv::InputArray  signature2,             // size2-by-(dims+1) float array
  int             distType,               // distance type (e.g., 'cv::DIST_L1')
  cv::InputArray  cost       = noArray(), // size1-by-size2 array (for cv::DIST_USER)
  float*          lowerBound = 0,         // input/output low bound on sig. distance
  cv::OutputArray flow       = noArray()  // output, size1-by-size2, from sig1 to sig2
);
```

Although we're applying the EMD to histograms, the interface prefers that we talk to it in terms of what the algorithm calls *signatures* for the first two array parameters. These signatures are arrays that are always of type `float` and consist of rows containing the histogram bin count followed by its coordinates. For the one-dimensional histogram of Figure 7-5, the signatures (listed array rows) for the left-hand column of histograms (skipping the model) would be as follows: top, $[[1, 0], [0, 1]]$; middle, $[[0.5, 0], [0.5, 1]]$; bottom, $[[0, 0], [1, 1]]$. If we had a bin in a three-dimensional histogram with a bin count of `537` at (x, y, z) index `(7, 43, 11)`, then the signature row for that bin would be `[537, 7, 43, 11]`. This is how we perform the necessary step of converting histograms into signatures. (We will go through this in a little more detail in Example 7-2.)



The parameter `distType` should be any of: *Manhattan distance* (`cv::DIST_L1`), *Euclidean distance* (`cv::DIST_L2`), *Checkerboard Distance* (`cv::DIST_C`), or a user-defined distance metric (`cv::DIST_USER`). In the case of the user-defined distance metric, the user supplies this information in the form of a (pre-calculated) cost matrix via the `cost` argument. (In this case, cost is an n_1-by-n_2 matrix, with n_1 and n_2 the sizes of `signature1` and `signature2`, respectively.)

The argument `lowerBound` has two functions (one as input, the other as output). As a return value, it is a lower bound on the distance between the centers of mass of the two histograms. In order for this lower

bound to be computed, one of the standard distance metrics must be in use (i.e., not `cv::DIST_USER`), and the total weights of the two signatures must be the same (as would be the case for normalized histograms). If you supply a lower-bound argument, you must also initialize that variable to a meaningful value. This value is used as the lower bound on separations for which the EMD will be computed at all.[5] Of course, if you want the EMD computed no matter what the distance is, you can always initialize `lowerBound` to zero.

The next argument, `flow`, is an optional n_1-by-n_2 matrix that can be used to record the *flow* of mass from the i^{th} point of `signature1` to the j^{th} point of `signature2`. In essence, this tells you how the mass was rearranged to give the computed total EMD.

As an example, suppose we have two histograms, `hist1` and `hist2`, which we want to convert into two signatures, `sig1` and `sig2`. Just to make things more difficult, let's suppose that these are two-dimensional histograms (as in the preceding code examples) of dimension `h_bins` by `s_bins`. Example 7-2 shows how to convert these two histograms into two signatures.

Example 7-2: Creating signatures from histograms for EMD; note that this code is the source of the data in Table 7-1, in which the hand histogram is compared in different lighting conditions

```
#include <opencv2/opencv.hpp>
#include <iostream>

using namespace std;

void help(){
  cout << "\nCall is:\n"
  << "./ch7_ex7_3_expanded modelImage0 testImage1 testImage2 badImage3\n\n"
  << "for example: "
  << "  ./ch7_ex7_3_expanded HandIndoorColor.jpg HandOutdoorColor.jpg "
  << "HandOutdoorSunColor.jpg fruits.jpg\n"
  << "\n";
}

// Compare 3 images' histograms
//
int main( int argc, char** argv ) {

  if( argc != 5 ) { help(); return -1; }

  vector<cv::Mat> src(5);
  cv::Mat         tmp;
  int             i;

  tmp = cv::imread(argv[1], 1);
  if(tmp.empty() ) {
    cerr << "Error on reading image 1," << argv[1] << "\n" << endl;
    help();
    return(-1);
  }

  // Parse the first image into two image halves divided halfway on y
  //
  cv::Size size  = tmp.size();
  int width      = size.width;
  int height     = size.height;
```

[5] This is important because it is typically possible to compute the lower bound for a pair of histograms much more quickly than the actual EMD. As a result, in many practical cases, if the EMD is above some bound, you probably do not care about the actual value of the EMD, only that it is "too big" (i.e., the things you are comparing are "not similar"). In this case, it is quite helpful to have `cv::EMD()` exit once it is known that the EMD value will be big enough that you do not care for the exact value.

```
int halfheight = height >> 1;
cout << "Getting size [[" << tmp.cols << "] [" << tmp.rows << "]]\n" << endl;
cout << "Got size (w,h): (" << size.width << "," << size.height << ")" << endl;

src[0] = cv::Mat(cv::Size(width,halfheight), CV::U8C3);
src[1] = cv::Mat(cv::Size(width,halfheight), CV::U8C3);

// Divide the first image into top and bottom halfs into src[0] and src[1]
//
cv::Mat_<cv::Vec3b>::iterator tmpit = tmp.begin<cv::Vec3b>();

// top half
cv::Mat_<cv::Vec3b>::iterator s0it = src[0].begin<cv::Vec3b>();
for(i = 0; i < width*halfheight; ++i, ++tmpit, ++s0it) *s0it = *tmpit;

// Bottom half
cv::Mat_<cv::Vec3b>::iterator s1it = src[1].begin<cv::Vec3b>();
for(i = 0; i < width*halfheight; ++i, ++tmpit, ++s1it) *s1it = *tmpit;

// Load the other three images
//
for(i = 2; i<5; ++i){
  src[i] = cv::imread(argv[i], 1);
  if(src[i].empty()) {
    cerr << "Error on reading image " << i << ": " << argv[i] << "\n" << endl;
    help();
    return(-1);
  }
}

// Compute the HSV image, and decompose it into separate planes.
//
vector<cv::Mat> hsv(5), hist(5), hist_img(5);
int          h_bins      = 8;
int          s_bins      = 8;
int          hist_size[] = { h_bins, s_bins }, ch[] = {0, 1};
float        h_ranges[]  = { 0, 180 };     // hue range is [0,180]
float        s_ranges[]  = { 0, 255 };
const float* ranges[]    = { h_ranges, s_ranges };
int          scale       = 10;

for(i = 0; i<5; ++i) {
  cv::cvtColor( src[i], hsv[i], cv::BGR2HSV );
  cv::calcHist( &hsv[i], 1, ch, noArray(), hist[i], 2, hist_size, ranges, true );
  cv::normalize( hist[i], hist[i], 0, 255, cv::NORM_MINMAX );
  hist_img[i] = cv::Mat::zeros( hist_size[0]*scale, hist_size[1]*scale, CV::U8C3 );

  // Draw our histogram For the 5 images
  //
  for( int h = 0; h < hist_size[0]; h++ )
    for( int s = 0; s < hist_size[1]; s++ ){
      float hval = hist[i].at<float>(h, s);
      cv::rectangle(
        hist_img[i],
        cv::Rect(h*scale, s*scale, scale, scale),
        cv::Scalar::all(hval),
        -1
      );
    }
}

// Display
cv::namedWindow( "Source0", 1 );cv::imshow( "Source0", src[0] );
```

```
  cv::namedWindow( "HS Histogram0", 1 );cv::imshow( "HS Histogram0", hist_img[0] );

  cv::namedWindow( "Source1", 1 );cv::imshow( "Source1", src[1] );
  cv::namedWindow( "HS Histogram1", 1 ); cv::imshow( "HS Histogram1", hist_img[1] );

  cv::namedWindow( "Source2", 1 ); cv::imshow( "Source2", src[2] );
  cv::namedWindow( "HS Histogram2", 1 ); cv::imshow( "HS Histogram2", hist_img[2] );

  cv::namedWindow( "Source3", 1 ); cv::imshow( "Source3", src[3] );
  cv::namedWindow( "HS Histogram3", 1 ); cv::imshow( "HS Histogram3", hist_img[3] );

  cv::namedWindow( "Source4", 1 ); cv::imshow( "Source4", src[4] );
  cv::namedWindow( "HS Histogram4", 1 ); cv::imshow( "HS Histogram4", hist_img[4] );

  // Compare the histogram src0 vs 1, vs 2, vs 3, vs 4
  //
  cout << "Comparison:\n"
    << "Corr                Chi                 Intersect          Bhat\n"
    << endl;

  for(i=1; i<5; ++i) {  // For each histogram
    cout << "Hist[0] vs Hist[" << i << "]: " << endl;;
    for(int j=0; j<4; ++j) { // For each comparison type
      cout << "method[" << j << "]: " << cv::compareHist(hist[0],hist[i],j) << "  ";
    }
    cout << endl;
  }

  //Do EMD and report
  //
  vector<cv::Mat> sig(5);
  cout << "\nEMD: " << endl;

  // Oi Vey, parse histograms to earth movers signatures
  //
  for( i=0; i<5; ++i) {

    vector<cv::Vec3f> sigv;

    // (re)normalize histogram to make the bin weights sum to 1.
    //
    cv::normalize(hist[i], hist[i], 1, 0, cv::NORM_L1);
    for( int h = 0; h < h_bins; h++ )
      for( int s = 0; s < s_bins; s++ ) {
        float bin_val = hist[i].at<float>(h, s);
        if( bin_val != 0 )
          sigv.push_back( cv::Vec3f(bin_val, (float)h, (float)s));
      }

    // make Nx3 32fC1 matrix, where N is the number of nonzero histogram bins
    //
    sig[i] = cv::Mat(sigv).clone().reshape(1);
    if( i > 0 )
      cout << "Hist[0] vs Hist[" << i << "]: "
           << EMD(sig[0], sig[i], cv::DIST_L2) << endl;
  }
  cv::waitKey(0);
}
```

Back Projection

Back projection is a way of recording how well the pixels fit the distribution of pixels in a histogram model. For example, if we have a histogram of flesh color, then we can use back projection to find flesh color areas in an image. The function for doing this kind of lookup has two variations, one for dense arrays, and one for sparse arrays.

Basic Back Projection: `cv::calcBackProject()`

Back-projection computes a vector from the selected channels of the input images just like `cv::calcHist()`, but instead of accumulating events in the output histogram it reads the output histogram and reports the bin value already present. In the context of statistics, if you think of the input histogram as a (prior) probability distribution for the particular vector (color) on some object, then back projection is computing the probability that any particular part of the image is in fact drawn from that prior distribution (e.g., part of the object).

```
void cv::calcBackProject(
  const cv::Mat*  images,            // C-style array of images, 8U or 32F
  int             nimages,           // number of images in 'images' array
  const int*      channels,          // C-style list of ints identifying channels
  cv::InputArray  hist,              // input histogram array
  cv::OutputArray backProject,       // output single channel array
  const float**   ranges,            // C-style array of 'dims' pairs set bin sizes
  double          scale      = 1,    // Optional scale factor for output
  bool            uniform    = true  // true for uniform binning
);

void cv::calcBackProject(
  const cv::Mat*  images,            // C-style array of images, 8U or 32F
  int             nimages,           // number of images in 'images' array
  const int*      channels,          // C-style list of ints identifying channels
  const cv::SparseMat& hist,         // input (sparse) histogram array
  cv::OutputArray backProject,       // output single channel array
  const float**   ranges,            // C-style array of 'dims' pairs set bin sizes
  double          scale      = 1,    // Optional scale factor for output
  bool            uniform    = true  // true for uniform binning
);

void cv::calcBackProject(
  cv::InputArrayOfArrays images,           // STL-vector of images, 8U or 32F
  const vector<int>&     channels,         // STL-vector, channels indices to use
  cv::InputArray         hist,             // input histogram array
  cv::OutputArray        backProject,      // output single channel array
  const vector<float>&   ranges,           // STL-style vector of range boundaries
  double                 scale      = 1,   // Optional scale factor for output
  bool                   uniform    = true // true for uniform binning
);
```

There are three versions of `cv::calcBackProject()`. The first two use C-style arrays for their inputs. One of these supports dense histograms and one supports sparse histograms. The third version uses the newer template-based inputs rather than C-style pointers[6]. In both cases, the image is provided in the form of a set of individual single or multichannel arrays (the `images` variable), while the histogram is precisely the form of histogram that is produced by `cv::calcHist()` (the `hist` variable). The set of single-channel arrays is exactly the same form as what you would have used when you called `cv::calcHist()` in the first place, only this time, it is the image you want to compare your histogram to. If the argument `images` is a C-style array (type `cv::Mat*`), you will also need to tell `cv::calcBackProject()` how many elements it has; this is the function of the `nimages` argument.

[6] Of these three, the third is the generally preferred form in modern code (i.e., the use of the C-style arrays for input is considered "old-fashioned" in most modern OpenCV code.)

The argument `channels` is a list of the channels that will actually be used in the back projection. Once again, the form of this argument is the same as the form of the corresponding argument used by `cv::calcHist()`. Each integer entry in the array `channels` is related to a channel in the input `arrays` by enumerating the channels in order, starting with the first array (`arrays[0]`), then for the second array (`images[1]`), and so on (e.g., if there were three matrices pointed to by `images`, with three channels each, their channels would correspond to the values `0`, `1`, and `2` for the first array, `3`, `4`, and `5` for the second array, and `6`, `7`, and `8` for the third array). As you can see, though the number of entries in `channels` must be the same as the dimensionality of the histogram `hist`, that number need not be the same as the number of arrays in (or the total number of channels represented by) `images`.

The results of the back projection computation will be placed in the array `backProject`. `backProject` will be the same size and type as `images[0]`, and have a single channel.

Because histogram data is stored in the same matrix structures used for other data, there is no place to record the bin information that was used in the original construction of the histogram. In this sense, to really comprehend a histogram completely, the associated `cv::Mat` (or `cv::SparseMat` or whatever) is needed, as well as the original `ranges` data structure that was used when the histogram was created by `cv::calcHist()`.[7] It is for this reason that this range of information needs to be supplied to `cv::calcBackProject()` in the `ranges` argument.

Finally, there are two optional arguments, `scale` and `uniform`. `scale` is an optional scale factor that is applied to the return values placed in `backProject`. (This is particularly useful if you want to visualize the results.) `uniform` is used to indicate whether or not the input histogram is a uniform histogram (in the sense of `cv::calcHist()`). Because `uniform` defaults to true, this argument is only needed for non-uniform histograms.

Example 7-1 showed how to convert an image into single-channel planes and then make an array of them. As described above, the values in `backProject` are set to the values in the associated bin in `hist`. If the histogram is normalized, then this value can be associated with a conditional probability value (i.e., the

[7] An alternative approach would have been to define another data-type for histograms which inherited from cv::Mat, but which also contained this bin information. The authors of the library chose not to take this route in the 2.0 (and later) version of the library in favor of simplifying the library.

probability that a pixel in `image` is a member of the type characterized by the histogram in `hist`).[8] In

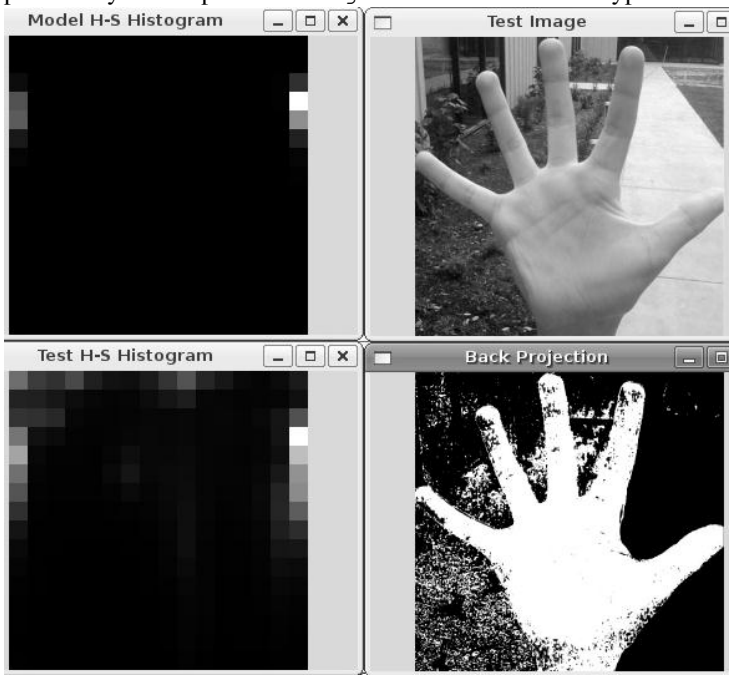


Figure 7-7, we use a flesh-color histogram to derive a probability of flesh image.

[8] Specifically, in the case of our flesh-tone H-S histogram, if C is the color of the pixel and F is the probability that a pixel is flesh, then this probability map gives us $p(C|F)$, the probability of drawing that color if the pixel actually is flesh. This is not quite the same as $p(F|C)$, the probability that the pixel is flesh given its color. However, these two probabilities are related by Bayes' theorem [Bayes1763] and so, if we know the overall probability of encountering a flesh-colored object in a scene as well as the total probability of encountering of the range of flesh colors, then we can compute $p(F|C)$ from $p(C|F)$. Specifically, Bayes' theorem establishes the following relation:

$$p(F|C) = \frac{p(F)}{p(C)} p(C|F)$$

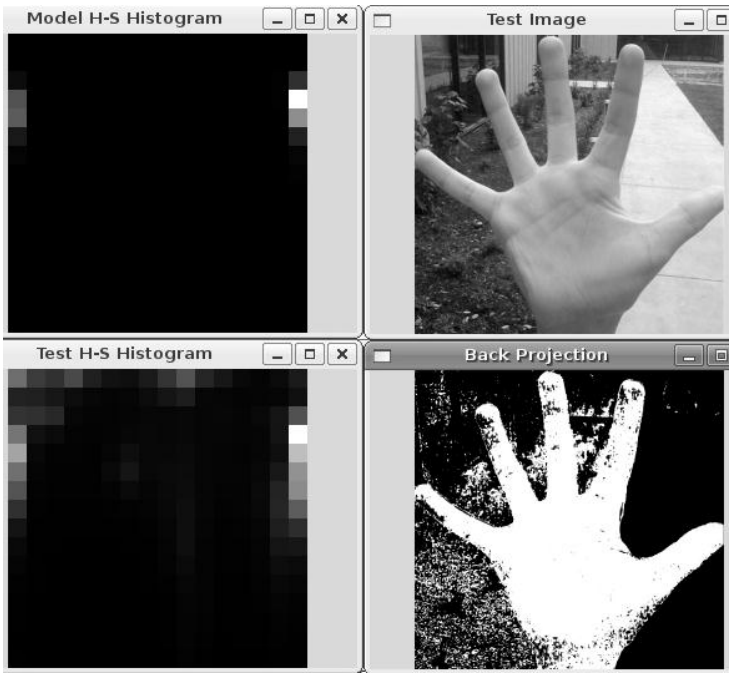


Figure 7-7: Back projection of histogram values onto each pixel based on its color: the HS (Hue and Saturation planes of an HSV representation of the image) flesh-color histogram (upper-left) is used to convert the hand image (upper-right) into the flesh-color probability image (lower-right); the lower-left panel is the histogram of the hand image

When `backProject` is a byte image rather than a float image, you should either not normalize the histogram or else scale it up before use. The reason is that the highest possible value in a normalized histogram is `1`, so anything less than that will be rounded down to `0` in the 8-bit image. You might also need to scale `backProject` in order to see the values with your eyes, depending on how high the values are in your histogram.

Template Matching

Template matching via `cv::matchTemplate()` is not based on histograms; rather, the function matches an actual image patch against an input image by "sliding" the patch over the input image using one of the matching methods described in this section.[9]

If, as in Figure 7-8, we have an image patch containing a face, then we can slide that face over an input image looking for strong matches that would indicate another face is present.

```
void cv::matchTemplate
  cv::InputArray  image,        // Input image to be searched, 8U or 32F, size W-by-H
  cv::InputArray  templ,        // Template to use, same type as 'image', size w-by-h
  cv::OutputArray result,       // Result image, type 32F, size (W-w+1)-by(H-h+1)
  int             method        // Comparison method to use
);
```

The input to `cv::matchTemplate()` starts with a single 8-bit or floating-point plane or color `image`. The matching model in `templ` is just a patch from another (presumably similar) image containing the object for which you are searching. The computed output will be put in the `result` image, which should

[9] In prior versions of the library, there was a function called: `cvCalcBackProjectPatch()`. This function was not ported to the 2.x library as it was considered to be very slow, and the similar results can be achieved through use of `cv::matchTemplate()`.

be a single-channel byte or floating-point image of size (`image.width - templ.width + 1, image.height - templ.height + 1`). The matching `method` is chosen from one of the options listed below (we use I to denote the input image, T the template, and R the result image in the definitions). For each of these, there is also a normalized version[10]:

Figure 7-8: `cv::matchTemplate()` sweeps a template image patch across another image looking for matches

Square difference matching method (method = cv::TM_SQDIFF)

These methods match the squared difference, so a perfect match will be 0 and bad matches will be large:

$$R_{sq_diff} = \sum_{x',y'} [T(x',y') - I(x+x',y+y')]^2$$

Normalized square difference matching method (method = cv::TM_SQDIFF_NORMED)

$$R_{sq_diff_normed} = \frac{\sum_{x',y'} [T(x',y') - I(x+x',y+y')]^2}{\sqrt{\sum_{x',y'} T(x',y')^2 \cdot \sum_{x',y'} I(x+x',y+y')^2}}$$

Correlation matching methods (method = cv::TM_CCORR)

These methods multiplicatively match the template against the image, so a perfect match will be large and bad matches will be small or zero.

$$R_{ccorr} = \sum_{x',y'} T(x',y') \cdot I(x+x',y+y')$$

Normalized cross-correlation matching method (method = cv::TM_SQDIFF_NORMED)

$$R_{ccorr_normed} = \frac{\sum_{x',y'} T(x',y') \cdot I(x+x',y+y')}{\sqrt{\sum_{x',y'} T(x',y')^2 \cdot \sum_{x',y'} I(x+x',y+y')^2}}$$

[10] The normalized versions were first developed by Galton [Galton] as described by Rodgers [Rodgers88]. The normalized methods are useful, as they can help reduce the effects of lighting differences between the template and the image. In each case, the normalization coefficient is the same.

Correlation coefficient matching methods (method = cv::TM_CCOEFF)

These methods match a template relative to its mean against the image relative to its mean, so a perfect match will be `1.0` and a perfect mismatch will be `-1.0`; a value of `0.0` simply means that there is no correlation (random alignments).

$$R_{ccoeff} = \sum_{x',y'} T'^{(x',y')} \cdot I'^{(x+x',y+y')}$$

$$T'(x',y') = T(x',y') - \frac{\sum_{x'',y''} T(x'',y'')}{(w-h)}$$

$$I'(x+x',y+y') = I(x+x',y+y') - \frac{\sum_{x'',y''} I(x'',y'')}{(w-h)}$$

Normalized correlation coefficient matching method (method = cv::TM_CCOEFF_NORMED)

$$R_{ccoeff_normed} = \frac{\sum_{x',y'} T'(x',y') \cdot I'(x+x',y+y')}{\sqrt{\sum_{x',y'} T'(x',y')^2 \cdot \sum_{x',y'} I'(x+x',y+y')^2}}$$

Here T' and I' are as defined for `cv::TM_CCOEFF`.

As usual, we obtain more accurate matches (at the cost of more computations) as we move from simpler measures (square difference) to the more sophisticated ones (correlation coefficient). It's best to do some test trials of all these settings and then choose the one that best trades off accuracy for speed in your application.

> Be careful when interpreting your results. The square-difference methods show best matches with a minimum, whereas the correlation and correlation-coefficient methods show best matches at maximum points.

Once we use `cv::matchTemplate()` to obtain a matching `result` image, we can then use `cv::minMaxLoc()` or `cv::minMaxIdx()` to find the location of the best match. Again, we want to ensure there's an area of good match around that point in order to avoid random template alignments that just happen to work well. A good match should have good matches nearby, because slight misalignments of the template shouldn't vary the results too much for real matches. Looking for the best matching "hill" can be done by slightly smoothing the result image before seeking the maximum (for correlation or correlation-coefficient) or minimum (for square-difference matching methods). The morphological operators (for example) can be helpful in this context.

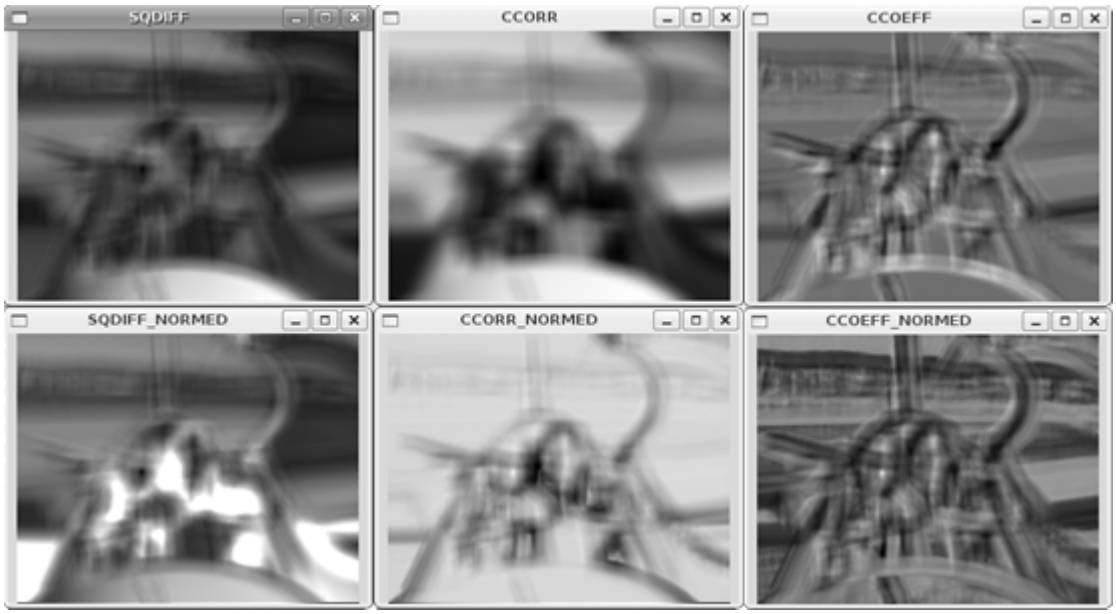


Figure 7-9: Match results of six matching methods for the template search depicted in Figure 7-8: the best match for square difference is zero and for the other methods it's the maximum point; thus, matches are indicated by dark areas in the left column and by bright spots in the other two columns

Example 7-3 should give you a good idea of how the different template matching techniques behave. This program first reads in a template and image to be matched and then performs the matching via the methods we've discussed here.

Example 7-3: Template matching

```
#include <opencv2/opencv.hpp>
#include <iostream>

using namespace std;

void help(){

  cout << "\n"
    "Example of using matchTemplate(). The call is:\n"
    "\n"
    "ch7_ex7_5 template image_to_be_searched\n"
    "\n"
    "   This routine will search using all methods:\n"
    "         cv::TM_SQDIFF        0\n"
    "         cv::TM_SQDIFF_NORMED 1\n"
    "         cv::TM_CCORR         2\n"
    "         cv::TM_CCORR_NORMED  3\n"
    "         cv::TM_CCOEFF        4\n"
    "         cv::TM_CCOEFF_NORMED 5\n"
    "\n";
}


// Display the results of the matches
//
int main( int argc, char** argv ) {

  if( argc != 3) {
    help();
```

```
    return -1;
  }

  cv::Mat src, templ, ftmp[6]; // ftmp is what to display on

  // Read in the template to be used for matching:
  //
  if((templ=cv::imread(argv[1], 1)).empty()) {
    cout << "Error on reading template " << argv[1] << endl;
    help(); return -1;
  }

  // Read in the source image to be searched:
  //
  if((src=cv::imread(argv[2], 1)).empty()) {
    cout << "Error on reading src image " << argv[2] << endl;
    help(); return -1;
  }

  // Do the matching of the template with the image
  for(int i=0; i<6; ++i){
    cv::matchTemplate( src, templ, ftmp[i], i);
    cv::normalize(ftmp[i],ftmp[i],1,0,cv::MINMAX);
  }

  // Display
  //
  cv::imshow( "Template", templ );
  cv::imshow( "Image", src );
  cv::imshow( "SQDIFF", ftmp[0] );
  cv::imshow( "SQDIFF_NORMED", ftmp[1] );
  cv::imshow( "CCORR", ftmp[2] );
  cv::imshow( "CCORR_NORMED", ftmp[3] );
  cv::imshow( "CCOEFF", ftmp[4] );
  cv::imshow( "CCOEFF_NORMED", ftmp[5] );

  // Let user view results:
  //
  cv::waitKey(0);
}
```

Note the use of `cv::normalize()` in this code, which allows us to display the results in a consistent way. (Recall that some of the matching methods can return negative-valued results.) We use the `cv::MINMAX` flag when normalizing; this tells the function to shift and scale the floating-point images so that all returned values are between `0.0` and `1.0`. Figure 7-9 shows the results of sweeping the face template over the source image (shown in Figure 7-9) using each of `cv::matchTemplate()`'s available matching methods. In outdoor imagery especially, it's almost always better to use one of the normalized methods. Among those, correlation coefficient gives the most clearly delineated match—but, as expected, at a greater computational cost. For a specific application, such as automatic parts inspection or tracking features in a video, you should try all the methods and find the speed and accuracy trade-off that best serves your needs.

Summary

In this chapter, we learned how OpenCV represents histograms at dense or sparse matrix objects. In practice, such histograms are typically used to represent probability density functions, which associate probability amplitude to every element of an array of some number of dimensions. We learned how to do basic operations on arrays, which are useful when interpreting arrays as probability distributions—such as normalization and comparison with other distributions.

Exercises

1. Generate 1,000 random numbers r_i between 0.0 and 1.0. Decide on a bin size and then take a histogram of $1/r_i$.
   a) Are there similar numbers of entries (i.e., within a factor of ±10) in each histogram bin?
   b) Propose a way of dealing with distributions that are highly nonlinear so that each bin has, within a factor of 10, the same amount of data.
2. Take three images of a hand in each of the three lighting conditions discussed in the text. Use `cv::calcHist()` to make an RGB histogram of the flesh color of one of the hands photographed indoors.
   a) Try using just a few large bins (e.g., 2 per dimension), a medium number of bins (16 per dimension) and many bins (256 per dimension). Then run a matching routine (using all histogram matching methods) against the other indoor lighting images of hands. Describe what you find.
   b) Now add 8 and then 32 bins per dimension and try matching across lighting conditions (train on indoor, test on outdoor). Describe the results.
3. As in exercise 2, gather RGB histograms of hand flesh color. Take one of the indoor histogram samples as your model and measure EMD (earth mover's distance) against the second indoor histogram and against the first outdoor shaded and first outdoor sunlit histograms. Use these measurements to set a distance threshold.
   a) Using this EMD threshold, see how well you detect the flesh histogram of the third indoor histogram, the second outdoor shaded, and the second outdoor sunlit histograms. Report your results.
   b) Take histograms of randomly chosen nonflesh background patches to see how well your EMD discriminates. Can it reject the background while matching the true flesh histograms?
4. Using your collection of hand images, design a histogram that can determine under which of the three lighting conditions a given image was captured. Toward this end, you should create features—perhaps sampling from parts of the whole scene, sampling brightness values, and/or sampling relative brightness (e.g., from top to bottom patches in the frame) or gradients from center to edges.
5. Assemble three histograms of flesh models from each of our three lighting conditions.
   a) Use the first histograms from indoor, outdoor shaded, and outdoor sunlit as your models. Test each one of these against the second images in each respective class to see how well the flesh-matching score works. Report matches.
   b) Use the "scene detector" you devised in part a, to create a "switching histogram" model. First use the scene detector to determine which histogram model to use: indoor, outdoor shaded, or outdoor sunlit. Then use the corresponding flesh model to accept or reject the second flesh patch under all three conditions. How well does this switching model work?
6. Create a flesh-region interest (or "attention") detector.
   a) Just indoors for now, use several samples of hand and face flesh to create an RGB histogram.
   b) Use `cv::calcBackProject()` to find areas of flesh.
   c) Use `cv::erode()` from Chapter 5 to clean up noise and then `cv::floodFill()` (from the same chapter) to find large areas of flesh in an image. These are your "attention" regions.
7. Try some hand-gesture recognition. Photograph a hand about two feet from the camera; create some (nonmoving) hand gestures: thumb up, thumb left, and thumb right.
   a) Using your attention detector from exercise 6, take image gradients in the area of detected flesh around the hand and create a histogram model for each of the three gestures. Also create a histogram of the face (if there's a face in the image) so that you'll have a (nongesture) model of that large flesh region. You might also take histograms of some similar but nongesture hand positions, just so they won't be confused with the actual gestures.

b) Test for recognition using a webcam: use the flesh interest regions to find "potential hands"; take gradients in each flesh region; use histogram matching above a threshold to detect the gesture. If two models are above threshold, take the better match as the winner.

c) Move your hand one to two feet further back and see if the gradient histogram can still recognize the gestures. Report.

8. Repeat exercise 7 but with EMD for the matching. What happens to EMD as you move your hand back?

9. With the same images as before but with captured image patches instead of histograms of the flesh around the hand, use `cv::matchTemplate()` instead of histogram matching. What happens to template matching when you move your hand backwards so that its size is smaller in the image?

10. With your hands facing a camera, take the gradient direction of several pictures of your open hand, a closed fist and a "thumbs up" gesture. Collect histograms of the gradient direction in a window around your hands. This becomes your trained "model". Now run live and use the various histogram matching techniques to see how well they can recognize your gestures.

8

Contours

Although algorithms like the Canny edge detector can be used to find the edge pixels that separate different segments in an image, they do not tell you anything about those edges as entities in themselves. The next step is to be able to assemble those edge pixels into contours. By now you have probably come to expect that there is a convenient function in OpenCV that will do exactly this for you, and indeed there is: `cv::findContours()`. We will start out this chapter with some basics that we will need in order to use this function.. With those concepts in hand, we will get into contour finding in some detail. Thereafter, we will move on to the many things we can do with contours after they've been computed.

Contour Finding

A contour is a list of points that represent, in one way or another, a curve in an image. This representation can be different depending on the circumstance at hand. There are many ways to represent a curve. Contours are represented in OpenCV by STL style `vector<>` template objects in which every entry in the vector encodes information about the location of the next point on the curve. It should be noted that though a sequence of 2d points (`vector<cv::Point>` or `vector<cv::Point2f>`) is the most common representation, there are other ways to represent contours as well. One example of such a construct is the Freeman Chain, in which each point is represented as a particular "step" in a given direction from the prior point. We will get into such variations in more detail as we encounter them. For now, the important thing to know is that contours are almost always STL vectors, but are not necessarily limited to the obvious vectors of `cv::Point` objects.

The function `cv::findContours()` computes contours from binary images. It can take images created by `cv::Canny()`, which have edge pixels in them, or images created by functions like `cv::threshold()` or `cv::adaptiveThreshold()`, in which the edges are implicit as boundaries between positive and negative regions.[1]

Contour Hierarchies

Before getting down to exactly how to extract contours, it is worth taking a moment to understand exactly what a contour is, and how groups of contours can be related to one another. Of particular interest is the concept of a contour tree, which is important for understanding one of the most useful ways

[1] There are some subtle differences between passing edge images and binary images to `cvFindContours();` we will discuss those shortly.

`cv::findContours()` (retrieval methods derive from Suzuki [Suzuki85]) can communicate its results to us.

Take a moment to look at

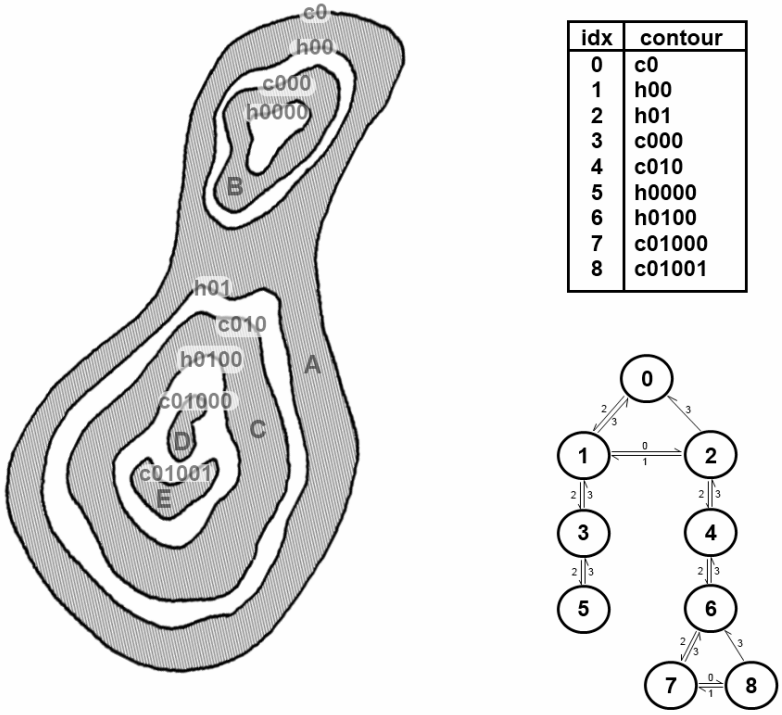


Figure 8-1, which depicts the functionality of `cv::findContours()`. The upper part of the figure shows a test image containing a number of "colored" (here, gray) regions (labeled A through E) on a light background. The lower portion of the figure depicts the same image along with the contours that will be located by `cv::findContours()`. Those contours are labeled cX or hX, where "c" stands for "contour," "h" stands for "hole," and "X" is some number. OpenCV and `cv::findContours()` distinguishes between the exterior boundaries of non-zero regions which are labeled *contours* and the *interior boundaries* which are labeled *holes*.

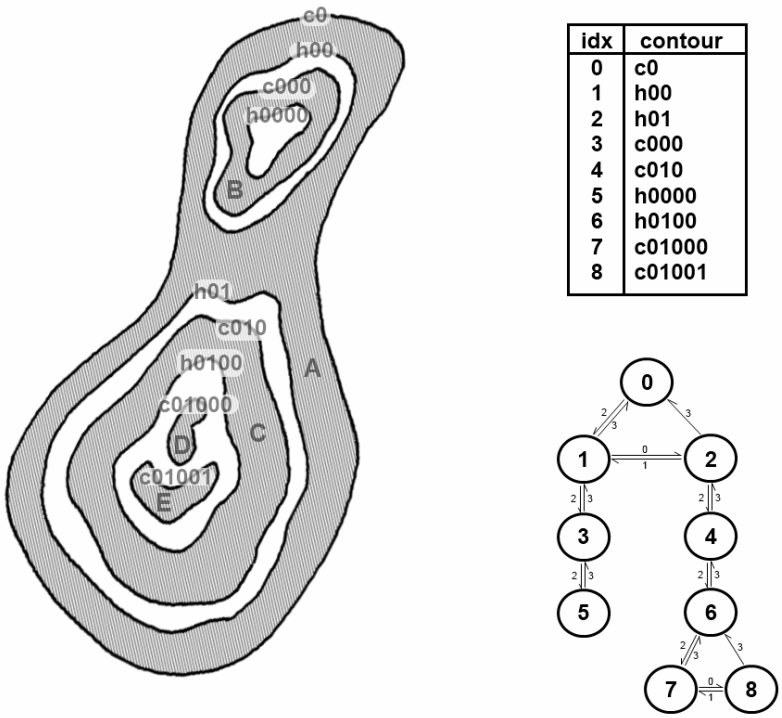


Figure 8-1: A test image (left side) passed to `cv::findContours()`*. There are five colored regions (labeled A, B, C, D, and E), but contours are formed from both the exterior and interior edges of each colored region. The result is nine contours in total. Each contour is identified and appears in an output list (the* `contours` *argument—upper-right). Optionally, a hierarchical representation can also be generated (the* `hierarchy` *argument—lower-right). In the graph shown (corresponding to the constructed contour tree), each node is a contour, the links in the graph are labeled with the index in the four-element data structure associated with each node in the* `hierarchy` *array.*

The concept of containment here is important in many applications. For this reason, OpenCV can be asked to assemble the found contours into a *contour tree*[2] that encodes the containment relationships in its structure. A contour tree corresponding to this test image would have the contour called c0 at the root node, with the holes h00 and h01 as its children. Those would in turn have as children the contours that they directly contain, and so on.

There are many possible ways to represent such a tree. OpenCV represents such trees with arrays (typically of vectors) in which each entry in the array represents one particular contour. In that array, each entry or *node* contains a set of four integers (typically represented as an element of type `cv::Vec4i`, just like an entry in a four-channel array). Each of the four components indicates another node in the hierarchy with a particular relationship to the current node. Where a particular relationship does not exist, that element of the data structure is set to `-1` (e.g., element 3, the parent id for the root node would have value `-1` because it has no parent).

Table 8-1: Meaning of each component in the four-element vector representation of each node in a contour hierarchy list

Index	Meaning

[2] Contour trees first appeared in Reeb [Reeb46] and were further developed by [Bajaj97], [Kreveld97], [Pascucci02], and [Carr04].

0	Next Contour (same level)
1	Previous Contour (same level)
2	First Child (next level down)
3	Parent (next level up)

By way of example, consider the contours in

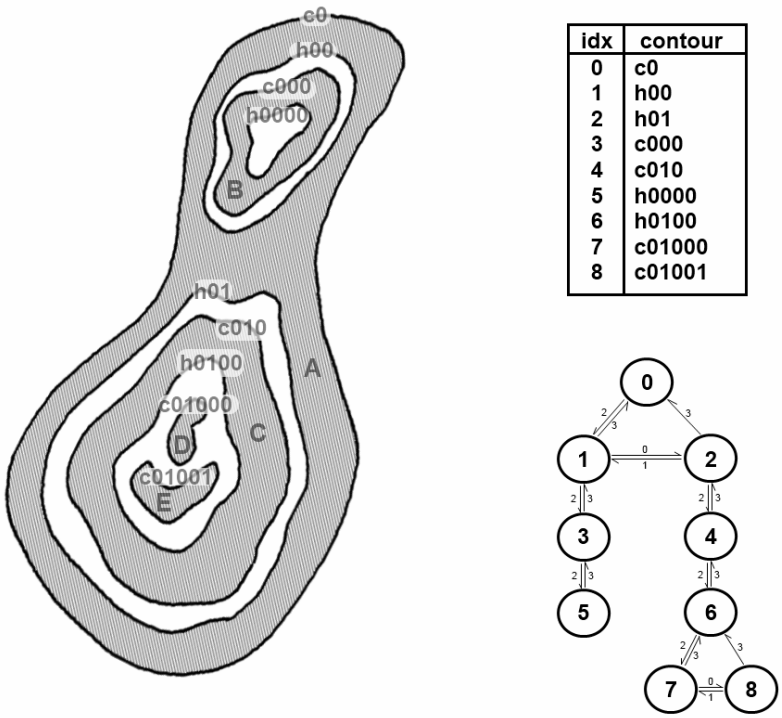
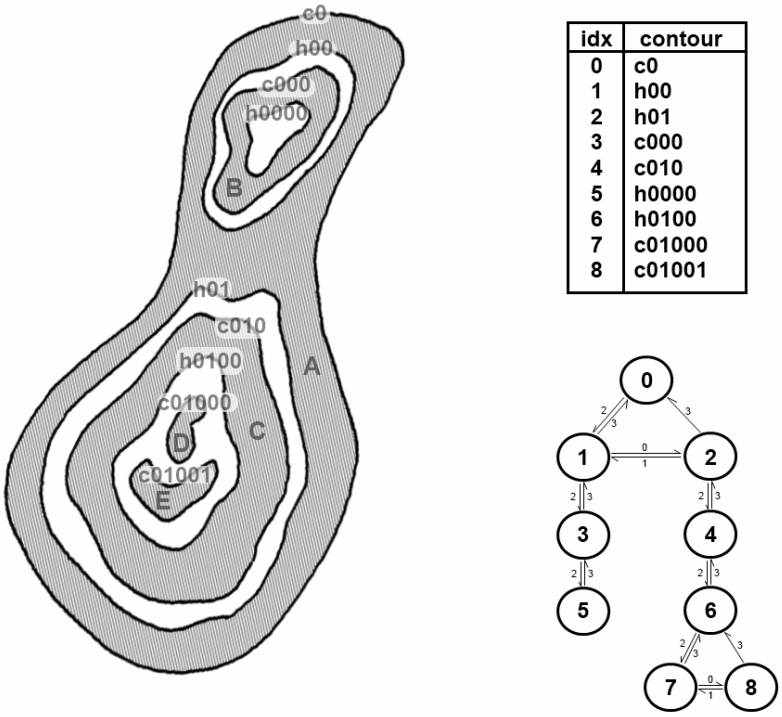


Figure 8-1. The five colored regions result in a total of nine total contours (counting both the exterior and the interior edges of each region). If a contour tree is constructed from these nine contours, each node will

have as children those contours that are contained within it. The resulting tree is visualized in the lower-



right of

Figure 8-1, For each node, those links that are valid are also visualized, and the links are labeled with the index associated with that link in the four-element data structure for that node (**Error! Reference source not found.**).

It is interesting to note the consequences of using `cv::findContours()` on an image generated by `cv::canny()` or a similar edge detector relative to what happens with a binary image such as the test image shown in Figure 8-1. Deep down, `cv::findContours()` does not really know anything about edge images. This means that, to `cv::findContours()`, an "edge" is just a very thin "on" area. As a result, for every exterior contour, there will be a hole contour that almost exactly coincides with it. This hole is actually just inside of the exterior boundary. You can think of it as the on-to-off transition that marks the interior edge of the edge.

Finding Contours with `cv::findContours()`

With this concept of contour trees in hand, we can look at the `cv::findContours()` function itself and see exactly how we tell it what we want and how we interpret its response:

```
void cv::findContours(
  cv::InputOutputArray    image,                // Input "binary" 8-bit single channel
  cv::OutputArrayOfArrays contours,             // Vector of vectors or points
  cv::OutputArray         hierarchy,            // (optional) topology information
  int                     mode,                 // Contour retrieval mode (Figure 8-2)
  int                     method,               // Approximation method
  cv::Point               offset = cv::Point() // (optional) Offset every point
);

void cv::findContours(
  cv::InputOutputArray    image,                // Input "binary" 8-bit single channel
  cv::OutputArrayOfArrays contours,             // Vector of vectors or points
```

```
  int                    mode,                // Contour retrieval mode (Figure 8-2)
  int                    method,              // Approximation method
  cv::Point              offset = cv::Point() // (optional) Offset every point
);
```

The first argument is the input image; this image should be an 8-bit single-channel image and will be interpreted as binary (i.e., as if all nonzero pixels were equivalent to one another). When it runs, `cv::findContours()` will actually use this image as scratch space for computation, so if you need that image for anything later, you should make a copy and pass that to `cv::findContours()`. The second argument is an array of arrays, which in most practical cases will mean an STL vector of STL vectors. This will be filled with the list of contours found (i.e., it will be a vector of contours, where `contours[i]` will be a specific contour and thus `contours[i][j]` would refer to a specific vertex in `contour[i]`).

The next argument `hierarchy` can be either supplied or not supplied (by using one of the two forms of the function shown above). If supplied, `hierarchy` is the output that describes the tree structure of the contours. The output `hierarchy` will be an array (again typically an STL vector) with one entry for each contour in `contours`. Each such entry will contain an array of four elements, each indicating the node to which a particular link from the current node is connected (**Error! Reference source not found.**).

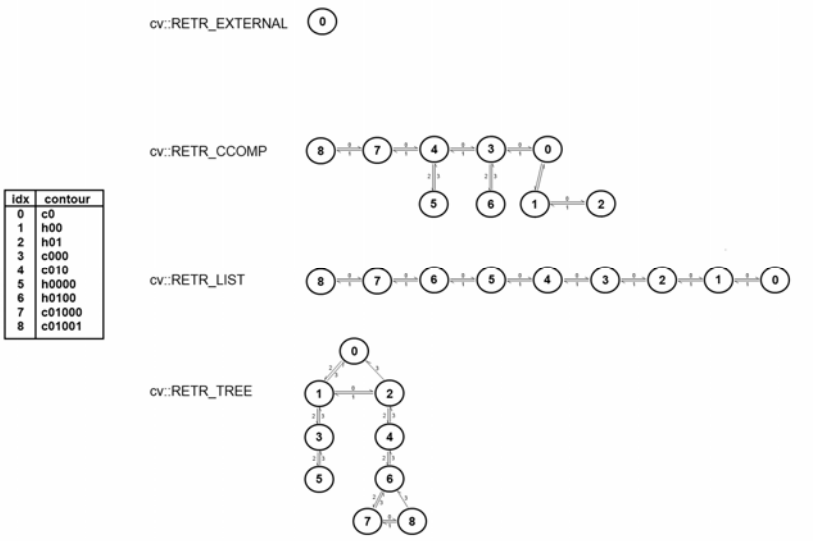


Figure 8-2: The way in which the tree node variables are used to "hook up" all of the contours located by `cv::findContours()`. *The contour nodes are the same as in*

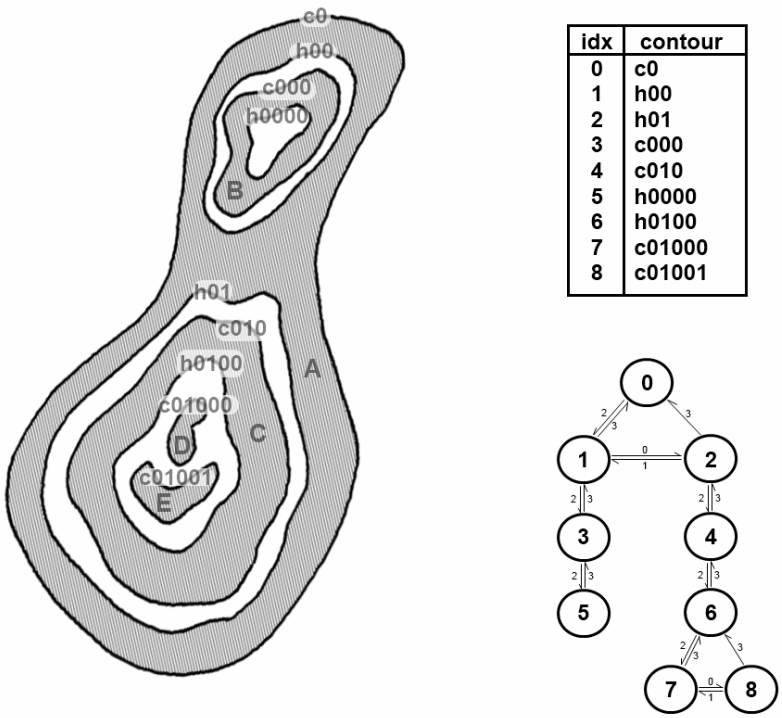


Figure 8-1.

The `mode` argument tells OpenCV how you would like the contours extracted. There are four possible values for `mode`: `cv::RETR_EXTERNAL`, `cv::RETR_LIST`, `cv::RETR_CCOMP`, and `cv::RETR_TREE`. Each mode is described below:

cv::RETR_EXTERNAL

Retrieves only the extreme outer contours. In

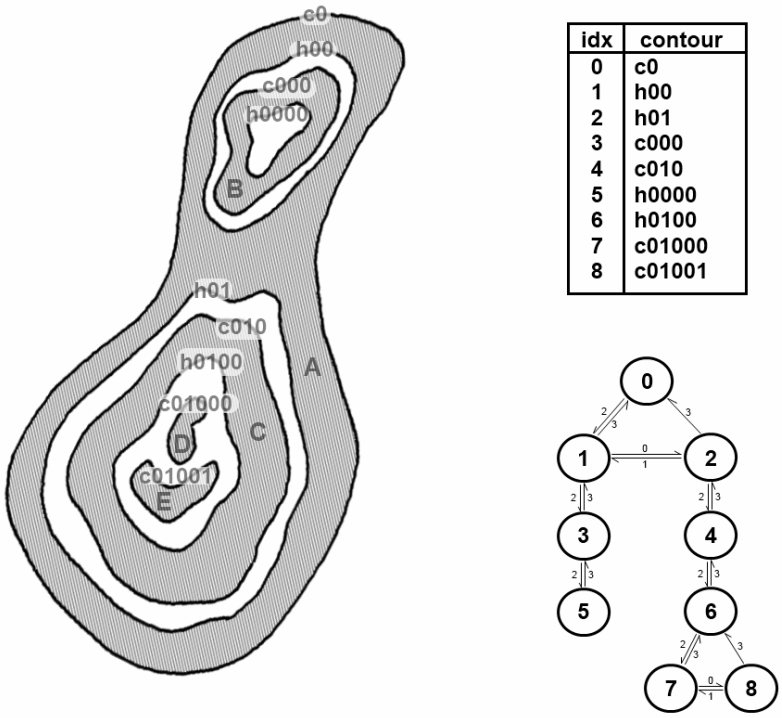


Figure 8-1, there is only one exterior contour, so

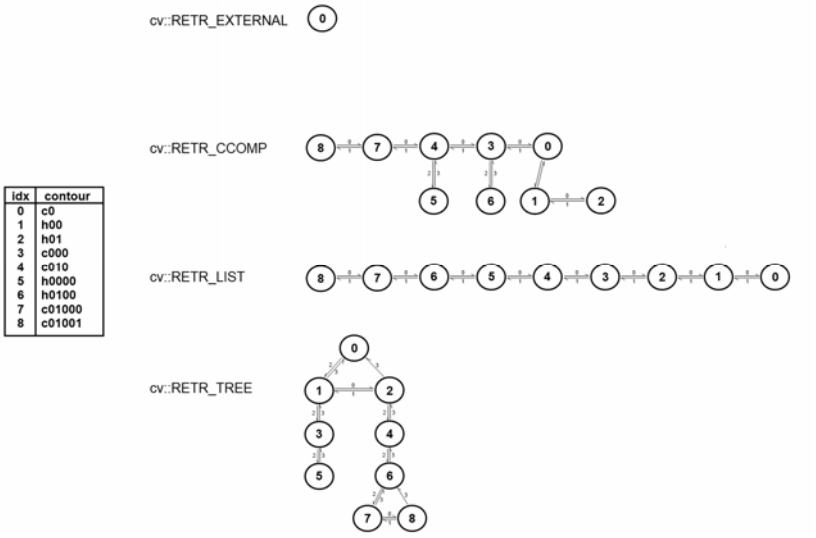


Figure 8-2 indicates that the first contour points to that outermost sequence and that there are no further connections.

`cv::RETR_LIST`

Retrieves all the contours and puts them in the list.

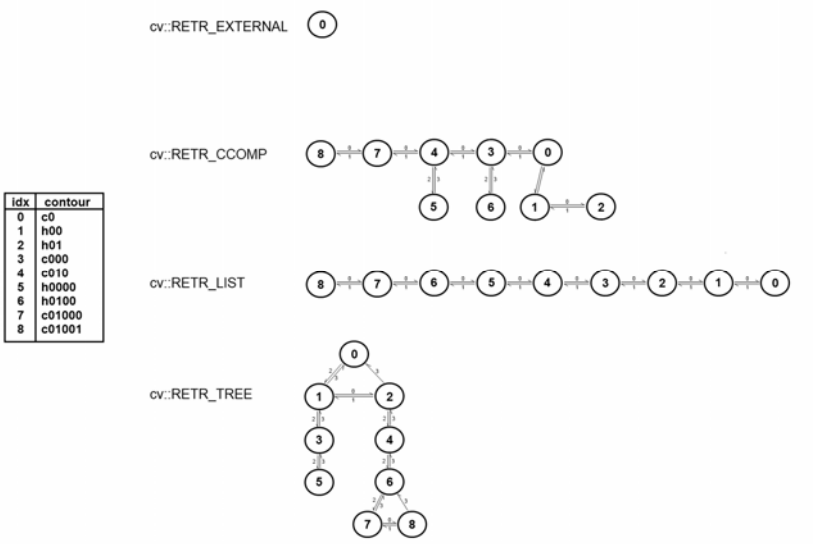


Figure 8-2 depicts the "hierarchy" resulting from the test image in

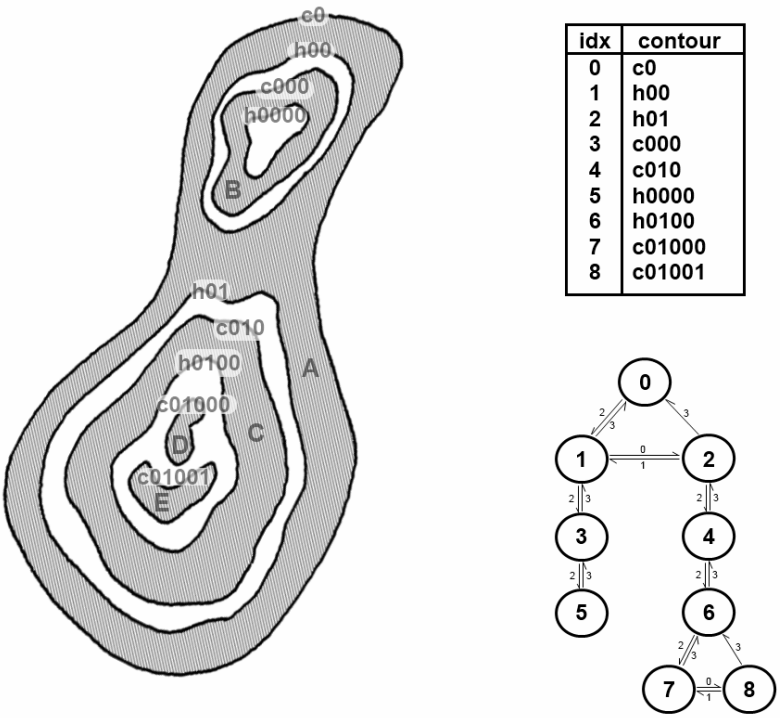
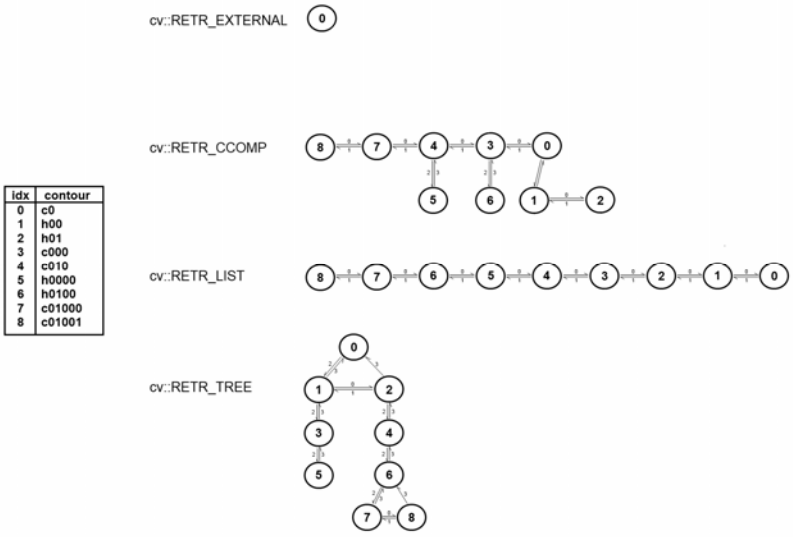


Figure 8-1. In this case, nine contours are found and they are all connected to one another by `hierarchy[i][0]` and `hierarchy[i][1]` (`hierarchy[i][2]` and `hierarchy[i][3]` are not used here).[3]

`cv::RETR_CCOMP`

Retrieves all the contours and organizes them into a two-level hierarchy, where the top-level boundaries are external boundaries of the components and the second-level boundaries are boundaries of the holes.



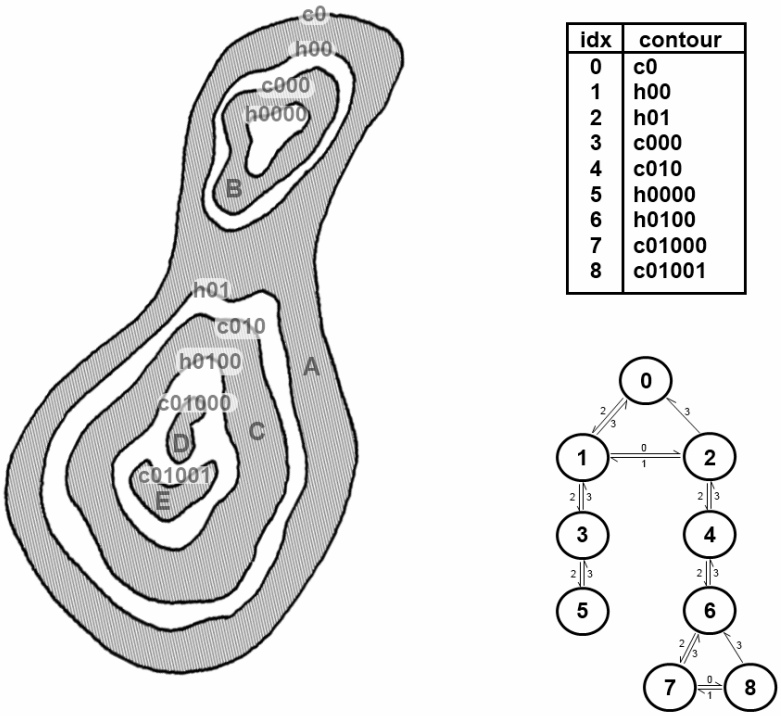
Referring to

Figure 8-2, we can see that there are five exterior boundaries, of which three contain holes. The holes are connected to their corresponding exterior boundaries by `hierarchy[i][2]` and `hierarchy[i][3]`. The outermost boundary c0 contains two holes. Because `hierarchy[i][2]` can contain only one value, the node can only have one child. All of the holes inside of c0 are connected to one another by the `hierarchy[i][0]` and `hierarchy[i][1]` pointers.

[3] You are not very likely to use this option. This organization primarily made sense in previous versions of the OpenCV library in which the contours return value was not automatically organized into a list as the `vector<>` type now implies.

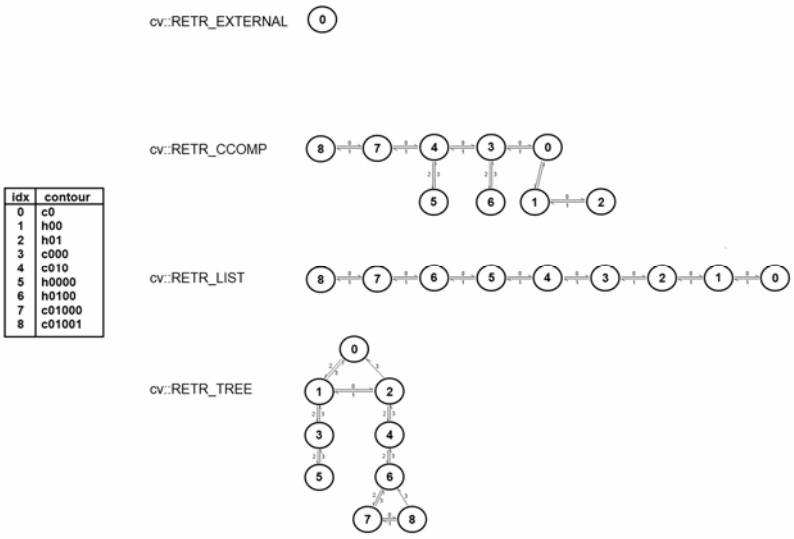
`cv::RETR_TREE`

Retrieves all the contours and reconstructs the full hierarchy of nested contours. In our example



idx	contour
0	c0
1	h00
2	h01
3	c000
4	c010
5	h0000
6	h0100
7	c01000
8	c01001

(



idx	contour
0	c0
1	h00
2	h01
3	c000
4	c010
5	h0000
6	h0100
7	c01000
8	c01001

Figure 8-1 and

Figure 8-2), this means that the root node is the outermost contour c0. Below c0 is the hole h00, which is connected to the other hole h01 at the same level. Each of those holes in turn has children (the contours c000 and c010, respectively), which are connected to their parents by vertical links. This continues down to the most-interior contours in the image, which become the leaf nodes in the tree.

The next five values pertain to the `method` (i.e., how the contours are represented):

`cv::CHAIN_APPROX_NONE`

Translates all the points from the contour code into points. This operation will produce a large number of points, as each point will be one of the eight neighbors of the previous point. No attempt is made to reduce the number of vertices returned.

`cv::CHAIN_APPROX_SIMPLE`

Compresses horizontal, vertical, and diagonal segments, leaving only their ending points. For many special cases, this can result in a substantial reduction of the number of points returned. The extreme example would be a rectangle (of any size) that is oriented along the x-y axes. In this case, only four points would be returned.

`cv::CHAIN_APPROX_TC89_L1` or `cv::CHAIN_APPROX_TC89_KCOS`

Applies one of the flavors of the Teh-Chin chain approximation algorithm.[4] The Teh-Chin algorithm is a more sophisticated (and more compute-intensive) method for reducing the number of points returned. The T-C algorithm requires no additional parameters to run.

The final argument to `cv::findContours()` is `offset`. The `offset` argument is optional. If present, every point in the returned contour will be shifted by this amount. This is particularly useful when either the contours are extracted from a region of interest, but you would like them represented in the parent image's coordinate system, or the reverse case, where you are extracting the contours in the coordinates of a larger image, but would like to express them relative to some subregion of the image.

Drawing Contours

One of the most straightforward things you might want to do with a list of contours, once you have it, is to draw the contours on the screen. For this we have `cv::drawContours()`:

```
void  cv::drawContours(
  cv::InputOutputArray   image,                // Will draw on input image
  cv::InputArrayOfArrays contours,             // Vector of vectors or points
  int                    contourIdx,           // Contour to draw (-1 is "all")
  const cv::Scalar&      color,                // Color for contours
  int                    thickness = 1,        // Thickness for contour lines
  int                    lineType  = 8,        // Connectedness ('4' or '8')
  cv::InputArray         hierarchy = noArray(), // optional (from findContours)
  int                    maxLevel  = INT_MAX,  // Max level in hierarchy to descend
  cv::Point              offset = cv::Point()  // (optional) Offset every point
)
```

The first argument `image` is simple: it is the image on which to draw the contours. The next argument, `contour`, is the list of contours to be drawn. This is in the same form as the `contour` output of `cv::findContours()`; it is a list of lists of points. The `contourIdx` argument can be used to select either a single contour to draw or to tell `cv::drawContours()` to draw all of the contours on the list provided in the `contours` argument. If `contourIdx` is a positive number, all contours will be drawn. If `contourIdx` is negative (usually this is just set to `-1`), all contours are drawn.

The `color`, `thickness`, and `lineType` arguments are similar to the corresponding arguments in other draw functions such as `cv::Line()`. As usual, the color argument is a four-component `cv::Scalar`, the thickness is an integer indicating the thickness of the lines to be drawn in pixels, and the `lineType` may be either 4 or 8 indicating whether the line is to be drawn as a 4-connected (ugly), 8-connected (not too ugly), or `cv::AA` (pretty) line.

[4] If you are interested in the details of how this algorithm works, you can consult Teh, C.H. and Chin, R.T., *On the Detection of Dominant Points on Digital Curve*. PAMI 11 8, pp. 859–872 (1989). Because the algorithm requires no tuning parameters, however, you can get quite far without knowing the deeper details of the algorithm.

The `hierarchy` argument corresponds to the hierarchy output from `cv::findContours()`. The `hierarchy` works with the `maxLevel` argument. The latter limits the depth in the hierarchy to which contours will be drawn in your image. Setting `maxLevel` to zero indicates that only "level 0" (the highest level) in the hierarchy should be drawn; higher numbers indicate that number of layers down from the highest level which should be included. Looking at

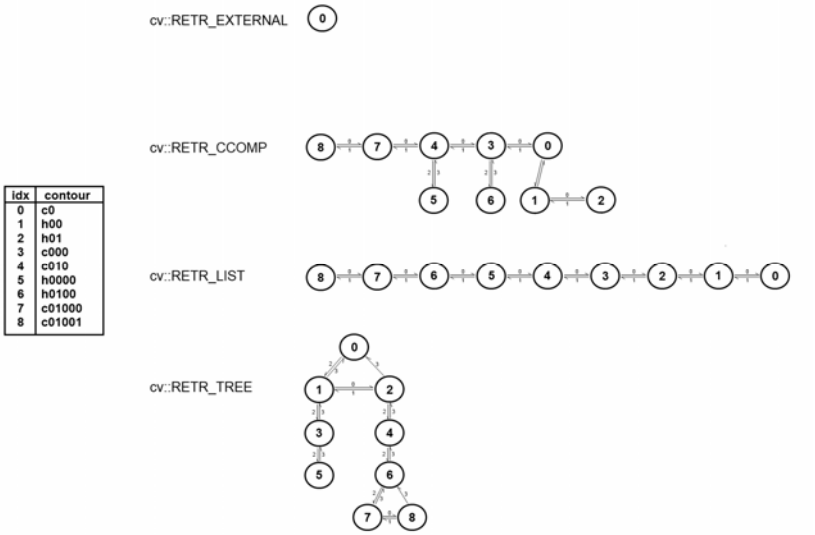


Figure 8-2, you can see that this is useful for contour trees; it is also potentially useful for connected components (`cv::RETR_CCOMP`) in case you would like only to visualize exterior contours (but not "holes"—interior contours).

Finally, we can give an `offset` to the draw routine so that the contour will be drawn elsewhere than at the absolute coordinates by which it was defined. This feature is particularly useful when the contour has already been converted to center-of-mass or other local coordinates. `offset` is particularly helpful in the case in which you have used `cv::findContours()` one or more times in different image sub-regions (ROIs) but now want to display all the results within the original large image. Conversely, we could use `offset` if we'd extracted a contour from a large image and then wanted to form a small mask for this contour.

A Contour Example

Example 8-1 is drawn from the OpenCV package. Here we create a window with an image in it. A trackbar sets a simple threshold, and the contours in the thresholded image are drawn. The image is updated whenever the trackbar is adjusted.

Example 8-1: Finding contours based on a trackbar's location; the contours are updated whenever the trackbar is moved

```
#include <opencv2/opencv.hpp>
#include <iostream>

using namespace std;

cv::Mat g_gray, g_binary;
int g_thresh = 100;

void on_trackbar(int, void*) {
  cv::threshold( g_gray, g_binary, g_thresh, 255, cv::THRESH_BINARY );
  vector< vector< cv::Point> > contours;
  cv::findContours(
    g_binary,
```

```
    contours,
    cv::noArray(),
    cv::RETR_LIST,
    cv::CHAIN_APPROX_SIMPLE
  );
  g_binary = cv::Scalar::all(0);

  cv::drawContours( g_binary, contours, -1, cv::Scalar::all(255));
  cv::imshow( "Contours", g_binary );
}

int main( int argc, char** argv )
{
  if( argc != 2 || ( g_gray = cv::imread(argv[1], 0)).empty() ) {
    cout << "Find threshold dependent contours\nUsage: ./ch8_ex8_1 fruits.jpg"
      << endl;
    return -1;
  }
  cv::namedWindow( "Contours", 1 );
  cv::createTrackbar(
    "Threshold",
    "Contours",
    &g_thresh,
    255,
    on_trackbar
  );
  on_trackbar(0, 0);
  cv::waitKey();
  return 0;
}
```

Here, everything of interest to us is happening inside of the function `on_trackbar()`. The image `g_gray` is thresholded such that only those pixels brighter than `g_thresh` remain nonzero. The `cv::findContours()` function is then called on this thresholded image. `cvDrawContours()` is then called and the contours are drawn (in white) onto the grayscale image.

Another Contour Example

In this example, we find contours on an input image and then proceed to draw them one by one. This is a good example to tinker with on your own to explore the effects of changing either the contour finding mode (`cv::RETR_LIST` in the code) or the `max_depth` that is used to draw the contours (`0` in the code). If you set `max_depth` to a larger number, notice that the example code steps through the contours returned by `cv::findContours()` by means of `hierarchy[i][1]`. Thus, for some topologies (`cv::RETR_TREE, cv::RETR_CCOMP`, etc.), you may see the same contour more than once as you step through. See Example 8-2.

Example 8-2: Finding and drawing contours on an input image

```
#include <opencv2/opencv.hpp>
#include <algorithm>
#include <iostream>

using namespace std;

struct AreaCmp
{
    AreaCmp(const vector<float>& _areas) : areas(&_areas) {}
    bool operator()(int a, int b) const { return (*areas)[a] > (*areas)[b]; }
    const vector<float>* areas;
};
```

```
int main(int argc, char* argv[]) {

  cv::Mat img, img_edge, img_color;

  // load image or show help if no image was provided
  if( argc != 2 || (img = cv::imread( argv[1], cv::LOAD_IMAGE_GRAYSCALE )).empty() ){
      cout << "\nExample 8_2 Drawing Contours\nCall is:\n./ch8_ex8_2 image\n\n";
      return -1;
  }

  cv::threshold(img, img_edge, 128, 255, cv::THRESH_BINARY);
  cv::imshow("Image after threshold", img_edge);
  vector< vector< cv::Point > > contours;
  vector< cv::Vec4i > hierarchy;

  cv::findContours(
    img_edge,
    contours,
    hierarchy,
    cv::RETR_LIST,
    cv::CHAIN_APPROX_SIMPLE
  );
  cout << "\n\nHit any key to draw the next contour, ESC to quit\n\n";
  cout << "Total Contours Detected: " << contours.size() << endl;
  vector<int> sortIdx(contours.size());
  vector<float> areas(contours.size());
  for( int n = 0; n < (int)contours.size(); n++ ) {
      sortIdx[n] = n;
      areas[n] = contourArea(contours[n], false);
  }
  // sort contours so that the largest contours go first
  std::sort( sortIdx.begin(), sortIdx.end(), AreaCmp(areas ));

  for( int n = 0; n < (int)sortIdx.size(); n++ ) {
     int idx = sortIdx[n];
     cv::cvtColor( img, img_color, cv::GRAY2BGR );
     cv::drawContours(img_color, contours, idx,
                  cv::Scalar(0,0,255), 2, 8, hierarchy,
                  0 // Try different values of max_level, and see what happens
                  );
     cout << "Contour #" << idx << ": area=" << areas[idx] <<
        ", nvertices=" << contours[idx].size() << endl;
     cv::imshow(argv[0], img_color);
     int k;
     if((k = cv::waitKey()&255) == 27)
       break;
  }
  cout << "Finished all contours\n";
  return 0;
}
```

More to Do with Contours

When analyzing an image, there are many different things we might want to do with contours. After all, most contours are—or are candidates to be—things that we are interested in identifying or manipulating. The various relevant tasks include characterizing the contours in various ways, simplifying or approximating them, matching them to templates, and so on.

In this section, we will examine some of these common tasks and visit the various functions built into OpenCV that will either do these things for us or at least make it easier for us to perform our own tasks.

Polygon Approximations

If we are drawing a contour or are engaged in shape analysis, it is common to approximate a contour representing a polygon with another contour having fewer vertices. There are many different ways to do this; OpenCV offers implementations of two of them.

Polygon Approximation with `cv::approxPolyDP()`

The routine `cv::approxPolyDP()` is an implementation of one of these two algorithms[5]:

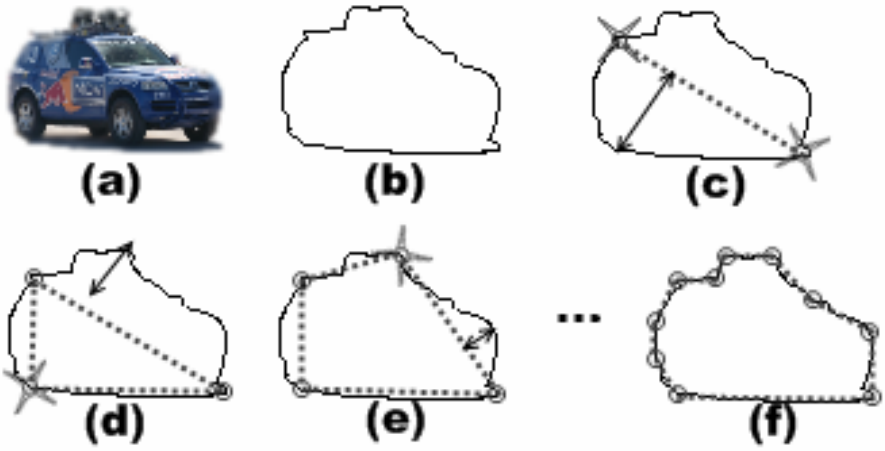
```
void cv::approxPolyDP(
  cv::InputArray  curve,       // Array or vector of 2-dimensional points
  cv::OutputArray approxCurve, // Result, type is same as 'curve'
  double          epsilon,     // Max distance from original 'curve' to 'approxCurve'
  bool            closed       // If true, assume link from last to first vertex
);
```

The `cv::approxPolyDP()` function acts on one polygon at a time, which is given in the input `curve`. The output of `cv:: approxPolyDP()` will be placed in the `approxCurve` output array. As usual, these polygons can be represented as either STL vectors of `cv::Point` objects or as OpenCV `cv::Mat` arrays of size N-by-1 (but having two channels). Whichever representation you choose, the input and output arrays used for curve and `approxCurve` should be of the same type.

The parameter `epsilon` is the accuracy of approximation you require. The meaning of the epsilon parameter is that this is the largest deviation you will allow between the original polygon and the final approximated polygon. `closed`, the last argument, indicates whether or not the sequence of points indicated by `curve` should be considered a closed polygon. If set to `true`, the curve will be assumed to be closed (i.e., that the last point is to be considered connected to the first point).

The Douglas-Peucker Algorithm Explained

In order to help understand how to set the `epsilon` parameter, as well as to better understand the output of `cv::approxPolyDP()`, it is worth taking a moment to understand exactly how the algorithm works.



In

[5] For aficionados, the method we are discussing here is the Douglas-Peucker (DP) approximation [Douglas73]. Other popular methods are the Rosenfeld-Johnson [Rosenfeld73] and Teh-Chin [Teh89] algorithms. Of those two, the Teh-Chin algorithm is not available in OpenCV as a reduction method, but is available at the time of the extraction of the polygon (see "Finding Contours with `cv::findContours()`").

Figure 8-3, starting with a contour (panel b), the algorithm begins by picking two extremal points and connecting them with a line (panel c). Then the original polygon is searched to find the point farthest from the line just drawn, and that point is added to the approximation.

The process is iterated (panel d), adding the next most distant point to the accumulated approximation, until all of the points are less than the distance indicated by the precision parameter (panel f). This means that good candidates for the parameter are some fraction of the contour's length, or of the length of its bounding box, or a similar measure of the contour's overall size.

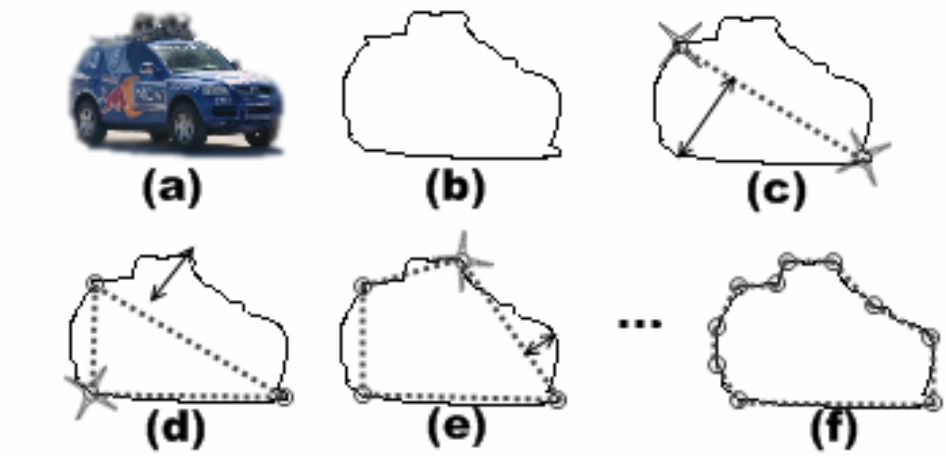


Figure 8-3: Visualization of the DP algorithm used by cv::`approxPolyDP()`: the original image (a) is approximated by a contour (b) and then, starting from the first two maximally separated vertices (c), the additional vertices are iteratively selected from that contour (d–f)

Geometry and Summary Characteristics

Another task that one often faces with contours is computing their various *summary characteristics*. These might include length or some other form of size measure of the overall contour. Other useful characteristics are the *contour moments*, which can be used to summarize the gross shape characteristics of a contour discussed in the next section. Some of the methods we will discuss work equally well for any collection of points (i.e., even those that do not imply a piecewise curve between those points). We will mention along the way which methods make sense only for curves (such as computing arc length) and which make sense for any general set of points (such as bounding boxes).

Length using cv::arcLength()

The subroutine `cv::arcLength()` will take a contour and return its length.

```
double  cv::arcLength(
  cv::InputArray  points,      // Array or vector of 2-dimensional points
  bool            closed       // If true, assume link from last to first vertex
);
```

The first argument of `cv::arcLength()` is the contour itself, whose form may be any of the usual representations of a curve (i.e., STL vector of points or array of two-channel elements). The second `closed` argument indicates whether the contour should be treated as closed. If the contour is considered closed, the distance from the last point in `points` to the first contributes to the overall arc length.

`cv::arcLength()` is an example of a case where the points argument is implicitly assumed to represent a curve, and so is not particularly meaningful for a general set of points.

Upright Bounding Box with `cv::boundingRect()`

Of course, the length and area are simple characterizations of a contour. One of the simplest ways to characterize a contour is to report a bounding box for that contour. The simplest version of that would be to simply compute the upright bounding rectangle. This is what `cv::boundingRect()` does for us:

```
cv::Rect cv::boundingRect(     // Return upright rectangle bounding the points
  cv::InputArray  points,      // Array or vector of 2-dimensional points
);
```

The `cv::boundingRect()` function just takes one argument, which is the curve whose bounding box you would like computed. The function returns a value of type `cv::Rect`, which is the bounding box you are looking for.

The bounding box computation is meaningful for any set of points, whether or not those points represent a curve or are just some arbitrary constellation of points.

A Minimum Area Rectangle with `cv::minAreaRect()`

One problem with the bounding rectangle from `cv::boundingRect()` is that it returns a `cv::Rect` and so can only represent a rectangle whose sides are oriented horizontally and vertically. In contrast, the routine `cv::minAreaRect()` returns the minimal rectangle that will bound your contour, and this rectangle may be inclined relative to the vertical; see

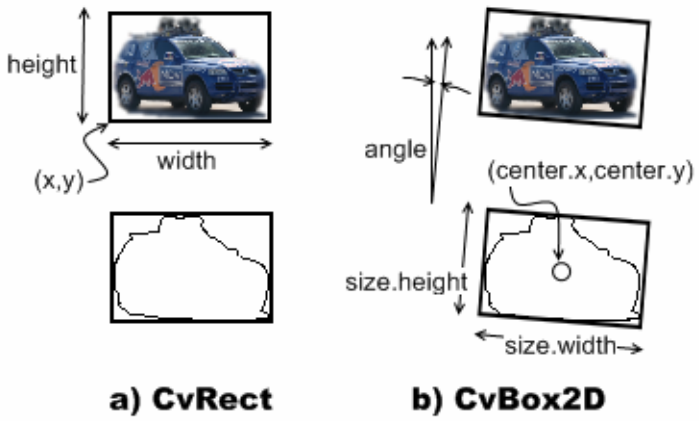


Figure 8-4. The arguments are otherwise similar to `cv::boundingRect()`. The OpenCV data type `cv::RotatedRect` is just what is needed to represent such a rectangle. Recall that it has the following definition:

```
class cv::RotatedRect {
  cv::Point2f center;      // Exact center point (around which to rotate)
  cv::Size2f  size;        // Size of rectangle (centered on 'center')
  float       angle;       // degrees
};
```

So, in order to get a little tighter fit, you can call `cv::minAreaRect()`:

```
cv::RotatedRect cv::minAreaRect( // Return rectangle bounding the points
  cv::InputArray  points,        // Array or vector of 2-dimensional points
);
```

As usual, `points` can be any of the standard representations for a sequence of points, and is equally meaningful for curves as well as arbitrary point sets.

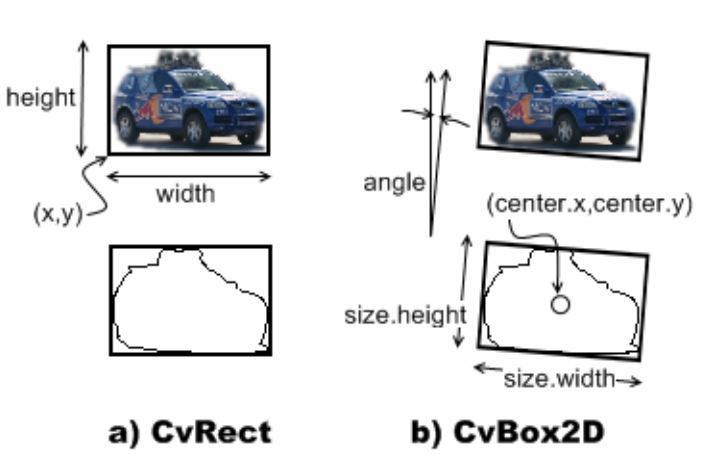


Figure 8-4: `cv::Rect` can represent only upright rectangles, but `cv::Box2D` can handle rectangles of any inclination

A Minimal Enclosing Circle Using `cv::minEnclosingCircle()`

Next, we have `cv::minEnclosingCircle()`.[6] This routine works pretty much the same way as the bounding box routines, with the exception that there is no convenient data type for the return value. As a result, you must pass in references to variables you would like set by `cv::minEnclosingCircle()`:

```
void  cv::minEnclosingCircle(
   cv::InputArray points,        // Array or vector of 2-dimensional points
   cv::Point2f&   center,        // Result location of circle center
   float&         radius         // Result radius of circle
);
```

The input curve is just the usual sequence of points representation. The center and radius variables are variables you will have to allocate and which will be set for you by `cv::minEnclosingCircle()`.

The `cv::minEnclosingCircle` function is equally meaningful for curves as for general point sets.

Fitting an Ellipse with `cv::fitEllipse()`

As with the minimal enclosing circle, OpenCV also provides a method for fitting an ellipse to a set of points:

```
cv::RotatedRect cv::fitEllipse(  // Return rectangle bounding the ellipse (Figure 8-5)
  cv::InputArray  points         // Array or vector of 2-dimensional points
);
```

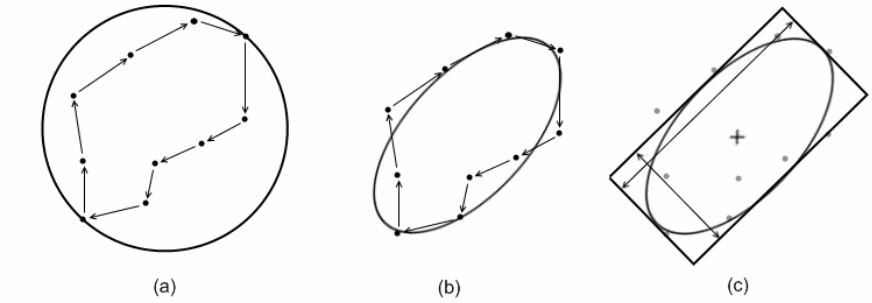
`cv::fitEllipse()` takes just a points array as argument.

At first glance, it might appear that `cv::fitEllipse()` is just the elliptical analog of `cv::minEnclosingCircle()`. There is, however, a subtle difference between `cv::minEnclosingCircle()` and `cv::fitEllipse()`, which is that the former simply computes the smallest circle that completely encloses the given points, whereas the latter uses a fitting function and returns the ellipse that is the best approximation to the point set. This means that not all points in the

[6] For more information on the inner workings of these fitting techniques, see Fitzgibbon and Fisher [Fitzgibbon95] and Zhang [Zhang96].

contour will even be enclosed in the ellipse returned by `cv::fitEllipse()`.[7] The fitting is done using a least-squares fitness function.

The results of the fit are returned in a `cv::RotatedRect` structure. The indicated box exactly encloses



the ellipse (

Figure 8-5).

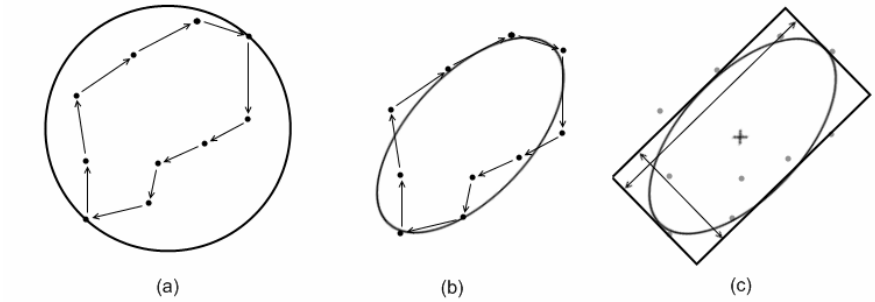


Figure 8-5: Ten-point contour with the minimal enclosing circle superimposed (a) and with the best fitting ellipsoid (b). A "rotated rectangle" is used by OpenCV to represent that ellipsoid (c).

Finding the Best Line Fit to Your Contour with `cv::fitLine()`

In many cases, your "contour" will actually be a set of points which you believe is approximately a straight line—or, more accurately, which you believe to be a noisy sample whose underlying origin is a straight line. In such a situation, the problem is to determine what line would be the best explanation for the points you observe. In fact, there are many reasons why one might want to find a best line fit, and as a result, there are many variations of how to actually do that fitting.

This fitting is done by minimizing a cost function, which is defined to be:

$$\text{cost}(\theta) \quad \sum_{\text{points } i} \rho(r_i), \text{ where } r_i \quad r(\theta, x_i).$$

Here θ is the set of parameters that define the line, x_i is the i[th] point in the contour, and r_i is the distance between that point and the line defined by θ. Thus, it is the function $\rho(r_i)$ that fundamentally distinguishes

[7] Of course, if the number of points is sufficiently small (or certain other degenerate cases—including all of the points being collinear), it is possible for all of the points to lie on the ellipse. In general, however, some points will be inside, some will be outside, and few if any will actually lie on the ellipse itself.

the different available fitting methods. In the case of $\rho(r_i) = \frac{1}{2}r_i^2$, the cost function will become the familiar least-squares fitting procedure that is probably familiar to most readers from elementary statistics. The more complex distance functions are useful when more robust fitting methods are needed (i.e., fitting methods that handle outlier data points more gracefully). Table 8-2 shows the available forms for $\rho(r)$ and the associated OpenCV enum values used by `cv::fitLine()`.

Table 8-2: Available distance metrics for the `distType` parameter in `cv::fitLine()`

distType	Distance Metric	
cv::DIST_L2	$\rho(r_i) = \frac{1}{2}r_i^2$	Least-squares method
cv::DIST_L1	$\rho(r_i) = r_i$	
cv::DIST_L12	$\rho(r_i) = 2 \cdot \left(\sqrt{1 + \frac{1}{2}r^2} - 1\right)$	
cv::DIST_FAIR	$\rho(r_i) = C^2 \cdot \left(\frac{r}{C} - \log\left(1 + \frac{r}{C}\right)\right)$	$C = 1.3998$
cv::DIST_WELSCH	$\rho(r_i) = \frac{C^2}{2} \cdot \left(1 - \exp\left(-\left(\frac{r}{C}\right)^2\right)\right)$	$C = 2.9846$
cv::DIST_HUBER	$\rho(r_i) = \begin{cases} \frac{1}{2}r_i^2 & r < C \\ C \cdot \left(r - \frac{C}{2}\right) & r \geq C \end{cases}$	$C = 1.345$

The OpenCV function `cv::fitLine()` has the following function prototype:

```
void cv::fitLine(
  cv::InputArray  points,        // Array or vector of 2-dimensional points
  cv::OutputArray line,          // Vector of Vec4f (2d), or Vec6f (3d)
  int             distType,      // Distance type (Table 8-2)
  double          param,         // Parameter for distance metric (Table 8-2)
  double          reps,          // Radius accuracy parameter
  double          aeps           // Angle accuracy parameter
);
```

The argument `points` is mostly what you have come to expect, a representation of a set of points either as a `cv::Mat` array or an STL vector. One very important difference, however, between `cv::fitLine()` and many of the other functions we are looking at in this section is that `cv::fitLine()` accepts both two- and three-dimensional points. The output `line` is a little strange. That entry should be of type `cv::Vec4f` (for a two-dimensional line), or `cv::Vec6f` (for a three-dimensional line) where the first half of the values gives the line direction and the second half a point on the line. The third argument, `distType`, allows us to select the distance metric we would like to use. The possible values for `distType` are shown in Table 8-2. The argument `param` is used to supply a value for the parameters used by some of the distance metrics (these parameters appear as the variables C in Table 8-2). This parameter can be set to `0`, in which case `cv::fitLine()` will automatically select the optimal value for the selected distance metric. The parameters `reps` and `aeps` represent your required accuracy for the origin of the fitted line (the x, y, z parts) and for the angle of the line (the v_x, v_y, v_z parts). Typical values for these parameters are `1e-2` for both of them.

Finding the Convex Hull of a Contour Using `cv::convexHull()`

There are many situations in which we need to simplify a polygon by finding its *convex hull*. The convex hull of a polygon or contour is the polygon which completely contains the original, is made only of points from the original, and is everywhere convex (i.e., that the internal angle between any three sequential points is less than 180 degrees). An example of a convex hull can be seen in Figure 8-6. There are many reasons to compute convex hulls. One particularly common reason is that testing if a point is inside of a convex polygon can be very fast, and it is often worthwhile to test first if a point is inside of the convex hull of a complicated polygon before even bothering to test if it is in the true polygon.

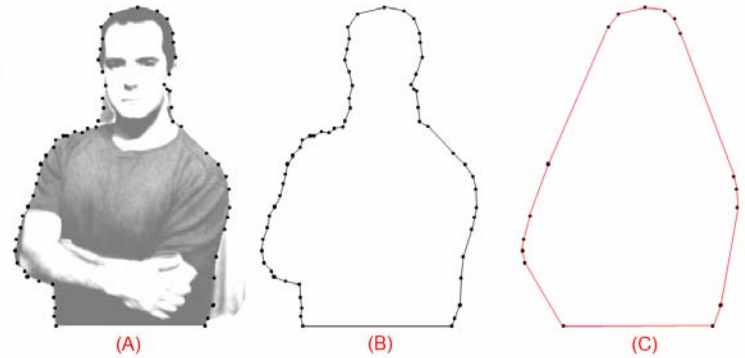


Figure 8-6: An image (a) is converted to a contour (b). The Convex hull of that contour (c) has far fewer points and is a much simpler piece of geometry.

In order to compute the convex hull of a contour, OpenCV provides the function `cv::convexHull()`:

```
void cv::convexHull(
  cv::InputArray  points,               // Array or vector of 2-dimensional points
  cv::OutputArray hull,                 // Can be array of points or of (int) indices
  bool            clockwise    = false, // if true, output points will be clocwise
  bool            returnPoints = true   // true for points in 'hull', else indices
);
```

The `points` input to `cv::convexHull()` can be any of the usual representations of a contour. The argument `hull` is where the resulting convex hull will appear. For this argument, you have two options: if you like, you can provide the usual contour type structure and `cv::convexHull()` will fill that up with the points in the resulting convex hull. The other option is to provide not an array of points but an array of integers. In this case, `cv::convexHull()` will associate an index with each point that is to appear in the hull and place those indices in `hull`. In this case, the indexes will begin at zero and the index value `i` will refer to the point `points[i]`.

The `clockwise` argument indicates how you would like to have `cv::convexHull()` express the hull it computes. If `clockwise` is set to `true`, then hull will be in a clockwise order, otherwise it will be in a counter-clockwise order. The final argument, `returnPoints`, is associated with the option to return point indices rather than point values. If `points` is an STL vector object, this argument is ignored, because the type of the vector template (`int` vs. `cv::Point`) can be used to infer what you want. If, however, `points` is a `cv::Mat` type array, `returnPoints` must be set to `true` if you are expecting point coordinates, and false if you are expecting indices.

Geometrical Tests

When dealing with bounding boxes and other summary representations of polygon contours, it is often desirable to perform such simple geometrical checks as polygon overlap or a fast overlap check between bounding boxes. OpenCV provides a small but handy set of routines for this sort of geometrical checking.

Many important tests that apply to rectangles are supported through interfaces provided by those rectangle types. For example, the `contains()` method of type `cv::Rect` can be passed a point and it will determine if that point is inside the rectangle.

Similarly, the minimal rectangle containing two rectangles can be computed with the logical "or" operator (e.g., `rect1 | rect2`), while the intersection of two rectangles can be computed with the logical "and" operator (e.g., `rect1 & rect2`). For `cv::RotatedRect`, we have no such functions since the result is mostly not a rectangle.

For operations on general curves, however, there are library functions that can be used.

Testing If a Contour Is Convex with `cv::isContourConvex()`

A common thing to want to know about a contour is whether or not it is convex. There are lots of reasons to do this, but one of the most common reasons is that there are a lot of algorithms for working with polygons that either only work on convex polygons or that can be simplified dramatically for the case of convex polygons. To test if a polygon is convex, simply call `cv::isContourConvex()` and pass it your contour in any of the usual representations. The contour passed will always be assumed to be a closed polygon (i.e., a link between the last and first points in the contour is presumed to be implied):

```
bool cv::isContourConvex(      // Return true if contour is convex
  cv::InputArray  contour      // Array or vector of 2-dimensional points
);
```

Because of the nature of the implementation, `cv::isContourConvex()` requires that the contour passed to it be a *simple polygon*. This means that the contour must not have any self-intersections.

Testing If a Point Is Inside a Polygon with `cv::pointPolygonTest()`

```
double cv::pointPolygonTest(   // Return distance to polygon boundary (or just side)
  cv::InputArray contour,      // Array or vector of 2-dimensional points
  cv::Point2f    pt,           // Test point
  bool           measureDist   // If true, return distance, otherwise, {0,+1,-1} only
);
```

This first geometry toolkit function is `cv::pointPolygonTest()`, which allows you to test whether a point is inside a polygon (indicated by the array `contour`). In particular, if the argument `measureDist` is set to `true`, then the function returns the distance to the nearest contour edge; that distance is 0 if the point is inside the contour and positive if the point is outside. If the `measure_dist` argument is `false` then the return values are simply + 1, – 1, or 0 depending on whether the point is inside, outside, or on an edge (or vertex), respectively. As always, the contour itself can be either an STL vector or an *n*-by-1 two-channel array of points.

Matching Contours and Images

Now that we have a pretty good idea of what a contour is and of how to work with contours as objects in OpenCV, we would like to move to the topic of how to use them for some practical purposes. The most common task associated with contours is matching them in some way with one another. We may have two computed contours that we'd like to compare or a computed contour and some abstract template with which we'd like to compare our contour. We will discuss both of these cases.

Moments

One of the simplest ways to compare two contours is to compute what are called *contour moments*. Contour moments represent certain high-level characteristics of a contour, an image, or a set of points. (The entire discussion that follows will apply equally well to contours, images, or point sets, so for convenience we will just refer to these options collectively as *objects*.) Numerically, the moments are defined by the following formula:

$$m_{p,q} \equiv \sum_{i=1}^{N} I(x_i, y_i) x_i^p y_i^q.$$

In this expression, the moment $m_{p,q}$ is defined as a sum over all of the pixels in the object, in which the value of the pixel at point *x, y* is multiplied by the factor $x^p y^q$. For example, if the image is a binary image (i.e., one in which every pixel is either zero of one), then m_{00} is just the area of the nonzero pixels in the image. In the case of a contour, the result is the length of the contour,[8] and in the case of a point set it is just the number of points. After a little thought, you should be able to convince yourself that for the same binary image, the m_{10} and m_{01} moments, divided by the m_{00} moment, are the average *x* and *y* values across the object. The term "moments" relates to the how this term is used in statistics, and the higher order moments can be related to what are called the moments of a statistical distribution (i.e., area, average, variance, etc.). In this sense, you can think of the moments of a non-binary image as being the moments of a binary image in which individual pixels are individually weighted.

Computing Moments with `cv::moments()`

The function that computes these moments for us is

```
cv::Moments cv::moments(             // Return structure contains moments
  cv::InputArray points,             // 2-dimensional points or an "image"
  bool           binaryImage = false // If false, interpret image values as "mass"
)
```

The first argument, `points`, is the contour we are interested in, and the second, `binaryImage`, tells OpenCV if the input image should be interpreted as a binary image. The `points` argument can be either a two-dimensional array (in which case it will be understood to be an image) or a set of points represented as an N-by-1 or 1-by-N array (with two channels) or an STL vector of `cv::Point` objects. In the latter cases (the sets of points), `cv::moments` will interpret these points not as a discrete set of points, but as a contour with those points as vertices.[9] The meaning of this second argument is that if `true`, all nonzero pixels will be treated as having value one (1), rather than whatever actual value is stored there. This is particularly useful when the image is the output of a threshold operation, but which might, for example, have 255 as its nonzero values. The `cv::moments()` function returns an instance of the `cv::Moments` object. That object is defined as follows:

```
class Moments {
public:
```

---

[8] Mathematical purists might object that $m_{0,1}$ should be not the contour's length but rather its area. But because we are looking here at a contour and not a filled polygon, the length and the area are actually the same in a discrete pixel space (at least for the relevant distance measure in our pixel space). There are also functions for computing moments of `IplImage` images; in that case, $m_{0,1}$ would actually be the area of nonzero pixels. Indeed the distinction is not entirely academic, however, a contour is actually represented as a set of vertex points, the formula used to compute the length will not give precisely the same area as would be computed by first rasterizing the contour (i.e., using `cv::drawContours()`) and then computing the area of that rasterization—though the two should converge to the same value in the limit of infinite resolution.

[9] In such case as you may need to handle a set of points, rather than a contour, it is most convenient to simply create an image containing those points.

```
    double m00;                   //   zero order moment              (x1)
    double m10, m01;              //  first order moments             (x2)
    double m20, m11, m02;         // second order moments             (x3)
    double m30, m21, m12, m03;    //  third order moments             (x4)
    double mu20, mu11, mu02;      // second order central moments     (x3)
    double mu30, mu21, mu12, mu03; //  third order central moments     (x4)
    double nu20, nu11, nu02;      // second order Hu invariant moments (x3)
    double nu30, nu21, nu12, nu03; //  third order Hu invariant moments (x4)
  Moments();
  Moments(
    double m00,
    double m10, double m01,
    double m20, double m11, double m02,
    double m30, double m21, double m12, double m03
  );
  Moments( const CvMoments& moments );  // convert v1.x struct to C++ object
  operator CvMoments() const;           // convert C++ object to v1.x struct
}
```

A single call to `cv::moments()` will compute all of the moments up to third order (i.e., moments for which $p + q \leq 3$). It will also compute what are called *central moments* and *normalized central moments*. We will discuss those next.

More About Moments

The moment computation just described gives some rudimentary characteristics of a contour that can be used to compare two contours. However, the moments resulting from that computation are not the best parameters for such comparisons in most practical cases. In general, the moments we have discussed so far will not be the same for two otherwise identical contours which are displaced relative to one another, of different size, or rotated relative to one another.

Central Moments Are Invariant Under Translation

Given a particular contour or image, the m_{00} moment of that contour will clearly be the same no matter where that contour appears in an image. The higher order moments, however, clearly will not be. Consider the m_{10} moment, which we identified earlier with the average x-position of a pixel in the object. Clearly given two otherwise identical objects in different places, the average x-position is different. It may be less obvious on casual inspection, but the second order moments, which tell us something about the spread of the object, are also not invariant under translation[10]. This is not particularly convenient, as we would certainly like (in most cases) to be able to use these moments to compare an object that might appear anywhere in an image to a reference object that appeared somewhere (probably somewhere else) in some reference image.

The solution to this is to compute what are called central moments, which are usually denoted $\mu_{p,q}$ and defined by the following relation:

$$\mu_{p,q} \quad \sum_{i=1}^{N} I(x_i, y_i)(x \quad \bar{x})^p (y \quad \bar{y})^q,$$

where:

[10] For those who are not into this sort of math jargon, the phrase "invariant under translation" means that some quantity computed for some object is unchanged if that entire object is moved (i.e., "translated") from one place to another in the image. The phrase "invariant under rotation" similarly means that the quantity being computed is unchanged of the object is rotated in the image.

$$\bar{x} = \frac{m_{10}}{m_{00}}, \text{ and } \bar{y} = \frac{m_{01}}{m_{00}}.$$

Of course, it should be immediately clear that $\mu_{00}=m_{00}$ (because the terms involving p and q vanish anyhow), and that the μ_{10} and μ_{01} central moments are both equal to zero as well. The higher order moments are thus the same as the non-central moments but measured with respect to the "center of mass" of (or in the coordinates of the center of mass of) the object as a whole. Because these measurements are relative to this center, they do not change if the object appears in any arbitrary location in the image.

> You will notice that there are no elements `mu00`, `mu10`, or `mu01` in the object cv::Moments. This is simply because these values are "trivial" (i.e., `mu00=m00`, and `mu10=mu01=0`). The same is true for the normalized central moments (except that `nu00=1`, while `nu10` and `nu01` are both zero). For this reason they are not included in the structure, as they would just waste memory storing redundant information.

Normalized Central Moments Are Also Invariant Under Scaling

Just as the central moments allow for us to compare two different objects that are in different locations in our image, it is also often important to be able to compare two different objects that are the same except for being different sizes. (This sometimes happens because we are looking for an object of a type that appears in nature of different sizes—e.g., bears—but more often it is simply because we do not necessarily know how far the object will be from the imager that generated our image in the first place.)

Just as the central moments are defined based on the original moments by subtracting out the average to achieve translational invariance, the *normalized central moments* achieve scale invariance by factoring out the overall size of the object. The formula for the normalized central moments is the following:

$$\nu_{p,q} = \frac{\mu_{p,q}}{m_{00}^{\left(\frac{p+q}{2}+1\right)}}.$$

This marginally intimidating formula just says that the normalized central moments are equal to the central moments up to a normalization factor that is itself just some power of the area of the object (with that power being greater for higher order moments).

There is no specific function for computing normalized moments in OpenCV, as they are computed automatically by `cv::Moments()` when the standard and central moments are computed.

Hu Invariant Moments are Invariant Under Rotation

Finally, the *Hu invariant moments* are linear combinations of the normalized central moments. The idea here is that, by combining the different normalized central moments, it is possible to create functions representing different aspects of the image in a way that is invariant to scale, rotation, and (for all but the one called h_1) reflection.

For the sake of completeness, we show here the actual definitions of the Hu moments:

$$h_1 = \nu_{20} + \nu_{02}$$

$$h_2 = (\nu_{20} + \nu_{02})^2 + 4\nu_{11}^2$$

$$h_3 = (\nu_{30} - 3\nu_{12})^2 + (3\nu_{21} - \nu_{03})^2$$

$$h_4 = (\nu_{30} + \nu_{12})^2 + (\nu_{21} + \nu_{03})^2$$

$$h_5 = (\nu_{30} - 3\nu_{12})(\nu_{30} + \nu_{12})\left[(\nu_{30} + \nu_{12})^2 - 3(\nu_{21} + \nu_{03})^2\right] + (3\nu_{21} - \nu_{03})(\nu_{21} + \nu_{03})\left[3(\nu_{30} + \nu_{12})^2 - (\nu_{21} + \nu_{03})^2\right]$$

$$h_6 = (\nu_{20} - \nu_{02})\left[(\nu_{30} + \nu_{12})^2 - (\nu_{21} + \nu_{03})^2\right] + 4\nu_{11}(\nu_{30} + \nu_{12})(\nu_{21} + \nu_{03})$$

$$h_7 = (3\nu_{21} - \nu_{03})(\nu_{30} + \nu_{12})\left[(\nu_{30} + \nu_{12})^2 - 3(\nu_{21} + \nu_{03})^2\right] - (\nu_{30} - 3\nu_{12})(\nu_{21} + \nu_{03})\left[3(\nu_{30} + \nu_{12})^2 - (\nu_{21} + \nu_{03})^2\right]$$

Looking at Figure 8-7 and Table 8-3, we can gain a sense of how the Hu moments behave. Observe first that the moments tend to be smaller as we move to higher orders. This should be no surprise in that, by their definition, higher Hu moments have more powers of various normalized factors. Since each of those factors is less than 1, the products of more and more of them will tend to be smaller numbers.

Figure 8-7: Images of five simple characters; looking at their Hu moments yields some intuition concerning their behavior.

Table 8-3: Values of the Hu moments for the five simple characters shown in

Figure 8-7.

	h_1	h_2	h_3	h_4	h_5	h_6	h_7
A	2.837e–1	1.961e–3	1.484e–2	2.265e–4	–4.152e–7	1.003e–5	–7.941e–9
I	4.578e–1	1.820e–1	0.000	0.000	0.000	0.000	0.000
O	3.791e–1	2.623e–4	4.501e–7	5.858e–7	1.529e–13	7.775e–9	–2.591e–13
M	2.465e–1	4.775e–4	7.263e–5	2.617e–6	–3.607e–11	–5.718e–8	–7.218e–24
F	3.186e–1	2.914e–2	9.397e–3	8.221e–4	3.872e–8	2.019e–5	2.285e–6

Other factors of particular interest are that the "I", which is symmetric under 180 degree rotations and reflection, has a value of exactly 0 for h_3 through h_7 and that the "O," which has similar symmetries, has all nonzero moments. We leave it to the reader to look at the figures, compare the various moments, and build a basic intuition for what those moments represent.

Computing Hu Invariant Moments with `cv::HuMoments()`

While the other moments were all computed with the same function `cv::moments()`, the Hu invariant moments are computed with a second function that takes the `cv::Moments` object you got from `cv::moments()` and returns a list of numbers for the seven invariant moments:

```
void cv::HuMoments(
  const cv::Moments& moments, // Input is result from cv::moments() function
  double*            hu       // Return is C-style array of 7 Hu moments
);
```

The function `cv::HuMoments()` expects a `cv::Moments` object and a pointer to a C-style array you should have already allocated with room for the seven invariant moments.

Matching and Hu Moments

```
double  cv::MatchShapes(
  cv::InputArray object1,      // First array of 2-d points or cv:U8C1 image
  cv::InputArray object2,      // Second array of 2-d points or cv:U8C1 image
  int            method,       // Comparison method (Table 8-4)
  double         parameter = 0 // Method-specific parameter (for later extensions)
);
```

Naturally, with Hu moments we would like to compare two objects and determine whether they are similar. Of course, there are many possible definitions of "similar." To make this process somewhat easier, the OpenCV function `cv::matchShapes()` allows us to simply provide two objects and have their moments computed and compared according to a criterion that we provide.

These objects can be either grayscale images or contours. In either case, `cv::matchShapes()` will compute the moments for you before proceeding with the comparison. The `method` used in `cv::matchShapes()` is one of the three listed in Table 8-4.

Table 8-4: Matching methods used by `cv::matchShapes()`

Value of method	`cv::matchShapes()` return value
`cv::CONTOURS_MATCH_I1`	$A_1 \quad \sum_{i=1\ldots7} \left\lvert \frac{1}{m_i^A} \quad \frac{1}{m_i^B} \right\rvert$
`cv::CONTOURS_MATCH_I2`	$A_2 \quad \sum_{i=1\ldots7} \lvert m_i^A \quad m_i^B \rvert$
`cv::CONTOURS_MATCH_I3`	$A_1 \quad \sum_{i=1\ldots7} \left\lvert \frac{m_i^A \quad m_i^B}{m_i^A} \right\rvert$

In the table, m_i^A and m_i^B are defined as:

$$m_i^A = \text{sign}(h_i^A) \cdot \log(h_i^A),$$

$$m_i^B = \text{sign}(h_i^B) \cdot \log(h_i^B).$$

In these expressions, h_i^A and h_i^B are the Hu invariant moments of images *A* and *B*, respectively.

Each of the three values defined in Table 8-4 has a different meaning in terms of how the comparison metric is computed. This metric determines the value ultimately returned by `cv::matchShapes()`. The final `parameter` argument is not currently used, so we can safely leave it at the default value of 0 (it is there for future comparison metrics that may require an additional user provided parameter).

Summary

In this chapter we learned about contours, sequences of points in two dimensions. These sequences could be represented as STL vectors of two-dimensional point objects (e.g., `cv::Vec2f`), as N-by-1 dual channel arrays, or as N-by-2 single channel arrays. Such sequences can be used to represent contours in in image plane, and there are many features built into the library to help us construct and manipulate these contours.

Contours are generally useful for representing spatial partitions of an image. In this context, the OpenCV library provides us with tools for comparing such partitions to one another, as well as for testing properties of these partitions, such as convexity, moments, or the relationship of an arbitrary point with such a contour.

Exercises

1. Finding the extremal points (i.e., the two points that are farthest apart) in a closed contour of *N* points can be accomplished by comparing the distance of each point to every other point.
   a) What is the complexity of such an algorithm?
   b) Explain how you can do this faster. What is the maximal closed contour length that could fit into a 4-by-4 image? What is its contour area?
2. Using PowerPoint or a similar program, draw a white circle of radius 20 on a black background (the circle's circumference will thus be $2 \pi 20 \approx 125.7$. Save your drawing as an image.
   a) Read the image in, turn it into grayscale, threshold, and find the contour. What is the contour length? Is it the same (within rounding) or different from the calculated length?
   b) Using 125.7 as a base length of the contour, run `cv::approxPolyDP()` using as parameters the following fractions of the base length: 90%, 66%, 33%, 10%. Find the contour length and draw the results.
3. Suppose we are building a bottle detector and wish to create a "bottle" feature. We have many images of bottles that are easy to segment and find the contours of, but the bottles are rotated and come in various sizes. We can draw the contours and then find the Hu moments to yield an invariant bottle-feature vector. So far, so good—but should we draw filled-in contours or just line contours? Explain your answer.
4. When using `cv::moments()` to extract bottle contour moments in exercise 3, how should we set `isBinary`? Explain your answer.
5. Take the letter shapes used in the discussion of Hu moments. Produce variant images of the shapes by rotating to several different angles, scaling larger and smaller, and combining these transformations. Describe which Hu features respond to rotation, which to scale, and which to both.
6. Make a shape in PowerPoint (or another drawing program) and save it as an image. Make a scaled, a rotated, and a rotated and scaled version of the object and then store these as images. Compare them

using `cv::matchShapes(). How do the match scores compare for Hu moments verses ordinary moments?`

7. Use a depth sensor such as the kinect camera to segment your hands. Use moments to attempt to recognize various gestures.

9

Background Subtraction

Overview of Background Subtraction

Because of its simplicity and because camera locations are fixed in many contexts, *background subtraction* (aka *background differencing*) is a fundamental image processing operation for video security applications. Toyama, Krumm, Brumitt, and Meyers give a good overview and comparison with many techniques [Toyama99]. In order to perform background subtraction, we first must "learn" a model of the background.

Once learned, this *background model* is compared against the current image and then the known background parts are subtracted away. The objects left after subtraction are presumably new foreground objects.

Of course, "background" is an ill-defined concept that varies by application. For example, if you are watching a highway, perhaps average traffic flow should be considered background. Normally, background is considered to be any static or periodically moving parts of a scene that remain static or periodic over the period of interest. The whole ensemble may have time-varying components, such as trees waving in morning and evening wind but standing still at noon. Two common but substantially distinct environment categories that are likely to be encountered are indoor and outdoor scenes. We are interested in tools that will help us in both of these environments. First we will discuss the weaknesses of typical background models and then will move on to discuss higher-level scene models. In that context, we present a quick method that is mostly good for indoor static background scenes whose lighting doesn't change much. We then follow this by a "codebook" method that is slightly slower but can work in both outdoor and indoor scenes; it allows for periodic movements (such as the trees waving in the wind) and for lighting to change slowly or periodically. This method is also tolerant to learning the background even when there are occasional foreground objects moving by. We'll top this off by another discussion of connected components (first seen in Chapter 5) in the context of cleaning up foreground object detection. We will then compare the quick background method against the codebook background method. This chapter will conclude with a discussion of the implementations available in the OpenCV library of two modern algorithms for background subtraction. These algorithms use the principles discussed in the chapter, but also include both extensions and implementation details which make them more suitable for real-world application.

Weaknesses of Background Subtraction

Although the background modeling methods mentioned here work fairly well for simple scenes, they suffer from an assumption that is often violated: namely that the behavior of all of the pixels in the image is statistically independent from the behavior of all of the others. Notably, the methods we describe here learn

a model for the variations a pixel experiences without considering any of its neighboring pixels. In order to take surrounding pixels into account, we could learn a multipart model, a simple example of which would be an extension of our basic independent pixel model to include a rudimentary sense of the brightness of neighboring pixels. In this case, we use the brightness of neighboring pixels to distinguish when neighboring pixel values are relatively bright or dim. We then learn effectively two models for the individual pixel: one for when the surrounding pixels are bright and one for when the surrounding pixels are dim. In this way, we have a model that takes into account the surrounding *context*. But this comes at the cost of twice as much memory use and more computation, since we now need different values for when the surrounding pixels are bright or dim. We also need twice as much data to fill out this two-state model. We can generalize the idea of "high" and "low" contexts to a multidimensional histogram of single and surrounding pixel intensities as well as make it even more complex by doing all this over a few time steps. Of course, this richer model over space and time would require still more memory, more collected data samples, and more computational resources.[1]

Because of these extra costs, the more complex models are usually avoided. We can often more efficiently invest our resources in cleaning up the *false positive* pixels that result when the independent pixel assumption is violated. This cleanup usually takes the form of image processing operations (`cv::erode(), cv::dilate()`, and `cv::floodFill()`, mostly) that eliminate stray patches of pixels. We've discussed these routines previously (Chapter 5) in the context of finding large and compact[2] *connected components* within noisy data. We will employ connected components again in this chapter and so, for now, will restrict our discussion to approaches that assume pixels vary independently.

Scene Modeling

How do we define background and foreground? If we're watching a parking lot and a car comes in to park, then this car is a new foreground object. But should it stay foreground forever? How about a trash can that was moved? It will show up as foreground in two places: the place it was moved to and the "hole" it was moved from. How do we tell the difference? And again, how long should the trash can (and its hole) remain foreground? If we are modeling a dark room and suddenly someone turns on a light, should the whole room become foreground? To answer these questions, we need a higher-level "scene" model, in which we define multiple levels between foreground and background states, and a timing-based method of slowly relegating unmoving foreground patches to background patches. We will also have to detect and create a new model when there is a global change in a scene.

In general, a scene model might contain multiple layers, from "new foreground" to older foreground on down to background. There might also be some motion detection so that, when an object is moved, we can identify both its "positive" aspect (its new location) and its "negative" aspect (its old location, the "hole").

In this way, a new foreground object would be put in the "new foreground" object level and marked as a positive object or a hole. In areas where there was no foreground object, we could continue updating our background model. If a foreground object does not move for a given time, it is demoted to "older foreground," where its pixel statistics are provisionally learned until its learned model joins the learned background model.

For global change detection such as turning on a light in a room, we might use global frame differencing. For example, if many pixels change at once, then we could classify it as a global rather than local change and then switch to using a different model for the new situation.

---

[1] In cases in which a computer is expected to "learn" something from data, it is often the case that the primary practical obstacle to success turns out to be having enough data. The more complex your model becomes, the easier it is to get yourself into a situation in which the expressive power of your model vastly exceeds your capability to generate training data for that model. We will re-encounter this issue in more detail in the later chapter on machine learning.

[2] Here we are using mathematician's definition of "compact," which has nothing to do with size.

A Slice of Pixels

Before we go on to modeling pixel changes, let's get an idea of what pixels in an image can look like over time. Consider a camera looking out a window to a scene of a tree blowing in the wind. **Error! Reference source not found.** shows what the pixels in a given line segment of the image look like over 60 frames. We wish to model these kinds of fluctuations. Before doing so, however, we make a small digression to discuss how we sampled this line because it's a generally useful trick for creating features and for debugging.

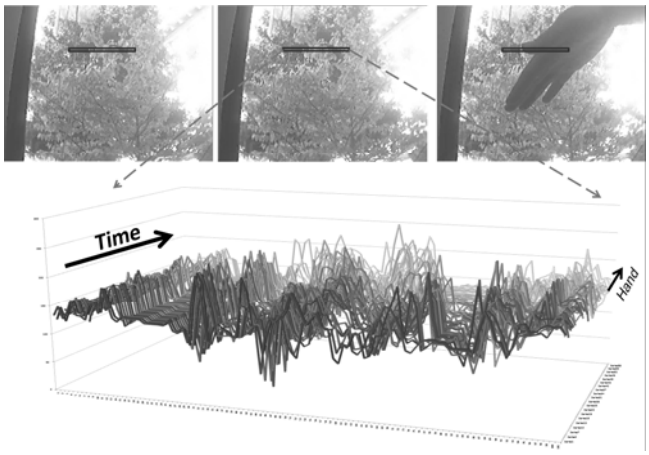


Figure 9-1: Fluctuations of a line of pixels in a scene of a tree moving in the wind over 60 frames: some dark areas (upper-left) are quite stable, whereas moving branches (upper-center) can vary widely

Because this comes up quite often in various contexts, OpenCV makes it easy to sample an arbitrary line of pixels. This is done with the object called the line iterator, which we encountered way back in Chapter 3. The line iterator, `cv::LineIterator`, is an object which, once instantiated, can be queried to give us information about all of the points along a line in sequence.

The first thing we need to do is to instantiate a line iterator object. We do this with the `cv::LineIterator` constructor:

```
cv::LineIterator::LineIterator(
  const cv::Mat& image,                // Image to iterate over
  cv::Point      pt1,                  // Start point for iterator
  cv::Point      pt2,                  // End point for iterator
  int            connectivity  = 8,    // Connectivity, either 4 or 8
  int            left_to_right = 0     // Iteration direction, 0 (false) or 1 (true)
);
```

Here, the input `image` may be of any type or number of channels. The points `pt1` and `pt2` are the ends of the line segment. The `connectivity` can be `4` (the line can step right, left, up, or down) or `8` (the line can additionally step along the diagonals). Finally, if `left_to_right` is set to `0` (false), then `line_iterator` scans from `pt1` to `pt2`; otherwise, it will go from the leftmost to the rightmost point.[3]

The iterator can then just be incremented through, pointing to each of the pixels along the line between the given endpoints points. We increment the iterator with the usual `cv::LineIterator ::operator++()`. All the channels are available at once. If, for example, our line iterator is called `line_iterator`, then we can access the current point by dereferencing the iterator (e.g., `*line_iterator`). One word of warning is in order here, however, which is that the return type of `cv::LineIterator::operator*()` is not a pointer to a built-in OpenCV vector type (i.e., `cv::Vec<>` or some instantiation of it), but rather it is a `uchar*` pointer. This means that you will

[3] The `left_to_right` flag was introduced because a discrete line drawn from `pt1` to `pt2` does not always match the line from `pt2` to `pt1`. Therefore, setting this flag gives the user a consistent rasterization regardless of the `pt1, pt2` order.

typically want to cast this value yourself to something like `cv::Vec3f*` (or whatever is appropriate for the array `image`).[4]

With this convenient tool in hand, we can take look at how it can be used to extract data from a scene. The program in Example 9-1 reads a movie file and generates the sort of data seen in Figure 9-1.

Example 9-1: Reads out the RGB values of all pixels in one row of a video and accumulates those values into three separate files

```
#include <opencv2/opencv.hpp>
#include <iostream>
#include <fstream>

using namespace std;

void help() {
  cout << "\n"
    << "Read out RGB pixel values and store them to disk\nCall:\n"
    << "./ch9_ex9_1 avi_file\n"
    << "\n This will store to files blines.csv, glines.csv and rlines.csv\n\n"
    << endl;
}

int main( int argc, char** argv  ) {

  if(argc != 2) { help(); return -1; }
  cv::namedWindow( "Example9_1", CV_WINDOW_AUTOSIZE );

  cv::VideoCapture cap;
  if((argc < 2)|| !cap.open(argv[1]))
  {
    cerr << "Couldn't open video file" << endl;
    help();
    cap.open(0);
    return -1;
  }

  cv::Point pt1(10,10), pt2(30,30);
  int max_buffer;
  cv::Mat rawImage;
  ofstream b,g,r;
  b.open("blines.csv");
  g.open("glines.csv");
  r.open("rlines.csv");

  // MAIN PROCESSING LOOP:
  //
  for(;;) {
    cap >> rawImage;
    if( !rawImage.data ) break;

    cv::LineIterator it( rawImage, pt1, pt2, 8);
    for( int j=0; j<it.count; ++j,++it ) {
      b << (int)(*it)[0] << ", ";
```

[4] In some cases, you can get away with being a little sloppy here. Specifically, when the image is already of unsigned character type, you can just access the elements directly with constructions like `(*line_iterator)[0]`, or `(*line_iterator)[1]`, and so on. On close inspection, these are actually dereferencing the iterator to get a character pointer, then using the built-in C offset dereference bracket operator, rather than casting the dereferenced iterator to an OpenCV vector type like Vec3f and accessing the channel through the overloaded dereferencing operator of that class. In the end, for the special case of Vec3b (or any number of channels), it happens to all come to the same thing.

```
      g << (int)(*it)[1] << ", ";
      r << (int)(*it)[2] << ", ";
      (*it)[2] = 255;              // Mark this sample in red
    }
    cv::imshow( "Example9_1", rawImage );
    int c = cv::waitKey(10);
    b << "\n"; g << "\n"; r << "\n";
  }

  // CLEAN UP:
  //
  b << endl; g << endl; r << endl;
  b.close(); g.close(); r.close();
  cout << "\n"
    << "Data stored to files: blines.csv, glines.csv and rlines.csv\n\n"
    << endl;
}
```

In Example 9-1, we stepped through the points, one at a time, and processed each one. Another common and useful way to approach the problem is to create a buffer (of the appropriate type) and copy the entire line into that buffer before processing the buffer. In that case, the buffer copy would have looked something like the following:

```
LineIterator it(rawImage, pt1, pt2, 8);
vector<Vec3b> buf(it.count);
for( int i=0; i < it.count; i++, ++it ) buf[i] = (const Vec3b)*it;
```

The primary advantage of this approach is that if the image `rawImage` were not of an unsigned character type, this method is a cleaner way of casting to the appropriate type.

We are now ready to move on to some methods for modeling the kinds of pixel fluctuations seen in Figure 9-1**Error! Reference source not found.**. As we move from simple to increasingly complex models, we shall restrict our attention to those models that will run in real time and within reasonable memory constraints.

Frame Differencing

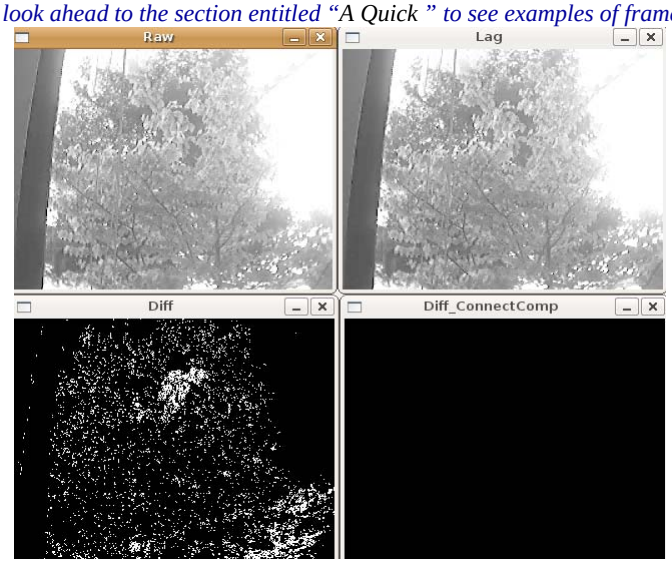
The very simplest background subtraction method is to subtract one frame from another (possibly several frames later) and then label any difference that is "big enough" the foreground. This process tends to catch the edges of moving objects. For simplicity, let's say we have three single-channel images: `frameTime1`, `frameTime2`, and `frameForeground`. The image `frameTime1` is filled with an older grayscale image, and `frameTime2` is filled with the current grayscale image. We could then use the following code to detect the magnitude (absolute value) of foreground differences in `frameForeground`:

```
cv::absdiff(
  frameTime1,                  // First input array
  frameTime2,                  // Second input array
  frameForeground              // Result array
);
```

Because pixel values always exhibit noise and fluctuations, we should ignore (set to 0) small differences (say, less than 15), and mark the rest as big differences (set to 255):

```
cv::threshold(
  frameForeground,              // Input Image
  frameForeground,              // Result image
  15,                           // Threshold value
  255,                          // Max value for upward operations
  cv::THRESH_BINARY             // Threshold type to use
);
```

The image `frameForeground` *then marks candidate foreground objects as 255 and background pixels as 0. We need to clean up small noise areas as discussed earlier; we might do this with* `cv::erode()` *followed by* `cv::dilate()` *or by using connected components. For color images, we could use the same code for each color channel and then combine the channels with the* `cv::max()` *function. This method is much too simple for most applications other than merely indicating regions of motion. For a more effective background model we need to keep some statistics about the means and average differences of pixels in the scene. You can look ahead to the section entitled "A Quick " to see examples of frame differencing in*



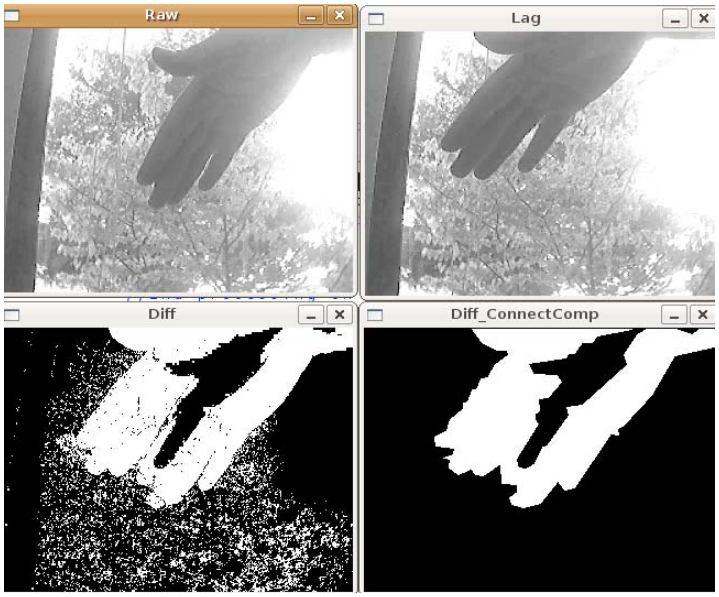


Figure 9-6 and Figure 9-7.

Averaging Background Method

The averaging method basically learns the average and standard deviation (or similarly, but computationally faster, the average difference) of each pixel as its model of the background.

Consider the pixel line from Figure 9-1. Instead of plotting one sequence of values for each frame (as we did in that figure), we can represent the variations of each pixel throughout the video in terms of an average value and a pixel's associated average differences (Figure 9-2). In the same video, a foreground object

(which is, in fact, a hand) passes in front of the camera. The resulting change in statistics of an associated pixel is shown in the figure.

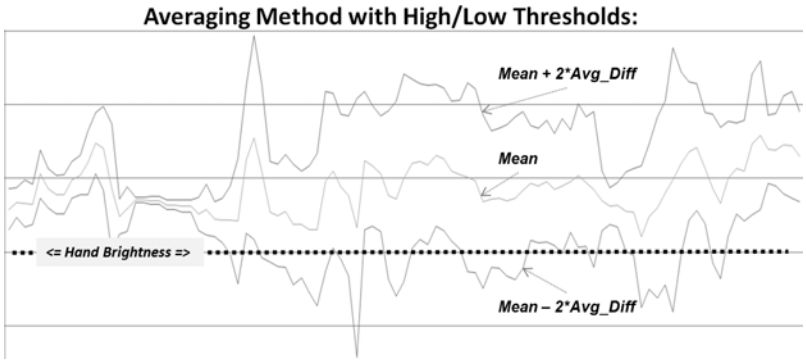


Figure 9-2: Data from Figure 9-1 presented in terms of average differences: an object (a hand) passes in front of the camera causing the statistics of a pixel's value to change.

The averaging method makes use of four OpenCV routines: `cv::Mat::operator+=()`, to accumulate images over time; `cv::absdiff()`, to accumulate frame-to-frame image differences over time; `cv::inRange()`, to segment the image (once a background model has been learned) into foreground and background regions; and `cv::max()`, to compile segmentations from different color channels into a single mask image. Because this is a rather long code example, we will break it into pieces and discuss each piece in turn.

First, we create pointers for the various scratch and statistics-keeping images we will need along the way. It will prove helpful to sort these pointers according to the type of images they will later hold.

```
#include <opencv2/opencv.hpp>
#include <iostream>
#include <fstream>

using namespace std;

// Global storage
//
// Float, 3-channel images
//
cv::Mat IavgF, IdiffF, IprevF, IhiF, IlowF;
cv::Mat tmp, tmp2;

// Float, 1-channel images
//
vector<cv::Mat> Igray(3);
vector<cv::Mat> Ilow(3);
vector<cv::Mat> Ihi(3);

// Byte, 1-channel image
//
cv::Mat Imaskt;

// Counts number of images learned for averaging later
//
float Icount;
```

Next, we create a single call to allocate all the necessary intermediate images. For convenience, we pass in a single image (from our video) that can be used as a reference for sizing the intermediate images:

```
// I is just a sample image for allocation purposes
// (passed in for sizing)
//
```

```
void AllocateImages( IplImage* I ){

  CvSize sz = cvGetSize( I );

  IavgF     = cv::Mat::zeros( sz, CV::F32C3 );
  IdiffF    = cv::Mat::zeros( sz, CV::F32C3 );
  IprevF    = cv::Mat::zeros( sz, CV::F32C3 );
  IhiF      = cv::Mat::zeros( sz, CV::F32C3 );
  IlowF     = cv::Mat::zeros( sz, CV::F32C3 );
  Icount    = 0.00001; // Protect against divide by zero

  tmp       = cv::Mat::zeros( sz, CV::F32C3 );
  tmp2      = cv::Mat::zeros( sz, CV::F32C3 );
  Imaskt    = cv::Mat( sz, CV::F32C1 );
}
```

In the next piece of code, we learn the accumulated background image and the accumulated absolute value of frame-to-frame image differences (a computationally quicker proxy[5] for learning the standard deviation of the image pixels). This is typically called for 30 to 1,000 frames, sometimes taking just a few frames from each second or sometimes taking all available frames. The routine will be called with a three-color-channel image of depth 8 bits:

```
// Learn the background statistics for one more frame
// I is a color sample of the background, 3-channel, 8u
//
void accumulateBackground( cv::Mat& I ){

  static int first = 1;                 // nb. Not thread safe
  I.convertTo( tmp, CV::F32 );          // convert to float
  if( !first ){
    IavgF += tmp;
    cv::absdiff( tmp, IprevF, tmp2 );
    IdiffF += tmp2;
    Icount += 1.0;
  }
  first = 0;
  IprevF = tmp;
}
```

We first use `cv::Mat::convertTo()` to turn the raw background 8-bit-per-channel, three-color-channel image into a floating-point three-channel image. We then accumulate the raw floating-point images into `IavgF`. Next, we calculate the frame-to-frame absolute difference image using `cv::absdiff()` and accumulate that into image `IdiffF`. Each time we accumulate these images, we increment the image count `Icount`, a global, to use for averaging later.

Once we have accumulated enough frames, we convert them into a statistical model of the background. That is, we compute the means and deviation measures (the average absolute differences) of each pixel:

```
void createModelsfromStats() {

    IavgF  *= (1.0/Icount);
    IdiffF *= (1.0/Icount);

    // Make sure diff is always something
    //
    IdiffF += cv::Scalar( 1.0, 1.0, 1.0 );
```

[5] Notice our use of the word "proxy." Average difference is not mathematically equivalent to standard deviation, but in this context it is close enough to yield results of similar quality. The advantage of average difference is that it is faster to compute than the standard deviation. With only a tiny modification of the code example you can use standard deviations instead and compare the quality of the final results for yourself; we'll discuss this more explicitly later in this section.

```
    setHighThreshold( 7.0 );
    setLowThreshold( 6.0 );
}
```

In this section, we use `cv::Mat::operator*=()` to calculate the average raw and absolute difference images by dividing by the number of input images accumulated. As a precaution, we ensure that the average difference image is at least 1; we'll need to scale this factor when calculating a foreground-background threshold and would like to avoid the degenerate case in which these two thresholds could become equal.

The next two routines, `setHighThreshold()` and `setLowThreshold()`, are utility functions that set a threshold based on the frame-to-frame average absolute differences (FFAAD). The FFAAD can be thought of as the basic metric against which we compare observed changes in order to determine if they are significant. The call `setHighThreshold(7.0)`, for example, fixes a threshold such that any value that is 7 times the FFAAD above average for that pixel is considered foreground; likewise, `setLowThreshold(6.0)` sets a threshold bound that is 6 times the FFAAD below the average for that pixel. Within this range around the pixel's average value, objects are considered to be background. These threshold functions are:

```
void setHighThreshold( float scale ) {
  IhiF = IavgF + (IdiffF * scale);
  cv::split( IhiF, Ihi );
}
void setLowThreshold( float scale ) {
  IlowF = IavgF - (IdiffF * scale);
  cv::split( IlowF, Ilow );
}
```

In `setLowThreshold()` and `setHighThreshold()`, we first scale the difference image (the FFAAD) prior to adding or subtracting these ranges relative to `IavgF`. This action sets the `IhiF` and `IlowF` range for each channel in the image via `cv::split()`.

Once we have our background model, complete with high and low thresholds, we use it to segment the image into foreground (things not "explained" by the background image) and the background (anything that fits within the high and low thresholds of our background model). Segmentation is done by calling:

```
// Create a binary: 0,255 mask where 255 means foreground pixel
// I      Input image, 3-channel, 8u
// Imask  Mask image to be created, 1-channel 8u
//
void backgroundDiff(
  cv::Mat& I,
  cv::Mat& Imask
) {
  I.convertTo( tmp, CV::F32 );    // To float
  cv::split( tmp, Igray );

  // Channel 1
  //
  cv::inRange( Igray[0], Ilow[0], Ihi[0], Imask );

  // Channel 2
  //
  cv::inRange( Igray[1], Ilow[1], Ihi[1], Imaskt );
  Imask = cv::min( Imask, Imaskt );

  // Channel 3
  //
  cv::inRange( Igray[2], Ilow[2], Ihi[2], Imaskt );
  Imask = cv::min( Imask, Imaskt );

  // Finally, invert the results
  //
```

```
  Imask = 255 – Imask;
}
```

This function first converts the input image `I` (the image to be segmented) into a floating-point image by calling `cv::Mat::convertTo()`. We then convert the three-channel image into separate one-channel image planes using `cv::split()`. These color channel planes are then checked to see if they are within the high and low range of the average background pixel via the `cv::inRange()` function, which sets the grayscale 8-bit depth image `Imaskt` to max (255) when it's in range and to 0 otherwise. For each color channel, we logically AND[6] the segmentation results into a mask image `Imask`, since strong differences in any color channel are considered evidence of a foreground pixel here. Finally, we invert `Imask` using `cv::operator-()`, because foreground should be the values out of range, not in range. The mask image is the output result.

By way of putting it all together, we can define a function `main()`, which reads in a video and builds a background model. For our example, we run the video in a training mode until the user hits the space bar, after which the video runs in a mode in which any foreground objects detected are highlighted in red:

```
void help() {
  cout << "\n"
    << "Train a background model on incoming video, then run the model\n"
    << "./ch9_ex9_2 avi_file\n"
    << endl;
}

int main( int argc, char** argv  ) {

  if(argc != 2) { help(); return -1; }
  cv::namedWindow( "Example9_2", cv::WINDOW_AUTOSIZE );

  cv::VideoCapture cap;
  if((argc < 2)|| !cap.open(argv[1])) {
    cerr << "Couldn't open video file" << endl;
    help();
    cap.open(0);
    return -1;
  }

  // FIRST PROCESSING LOOP (TRAINING):
  //
  while(1) {
    cap >> image;
    if( !image.data ) exit(0);

    accumulateBackground( image );

    cv::imshow( "Example9_2", rawImage );
    if( cv::waitKey(7) == 0x20 ) break;
  }

  // We have all of our data, so create the models
  //
  createModelsfromStats();

  // SECOND PROCESSING LOOP (TESTING):
  //
  cv::Mat mask;
  while(1) {
```

[6] In this circumstance, you could have used the bitwise OR operator as well, because the images being OR'ed are unsigned character images and only the values `0x00` and `0xff` are relevant. In general, however, the `cv::max()` operation is a good way to get a "fuzzy" OR which responds sensibly to a range of values.

```
    cap >> image;
    if( !image.data ) exit(0);

    backgroundDiff( image, mask );

    // A simple visualization is to write to the red channel
    //
    cv::split( image, Igray );
    Igray[2] = cv::max( mask, Igray[2] );
    cv::merge( Igray, image );

    cv::imshow( "Example9_2", image );
    if( cv::waitKey(7) == 0x20 ) break;
  }

  exit(0);
}
```

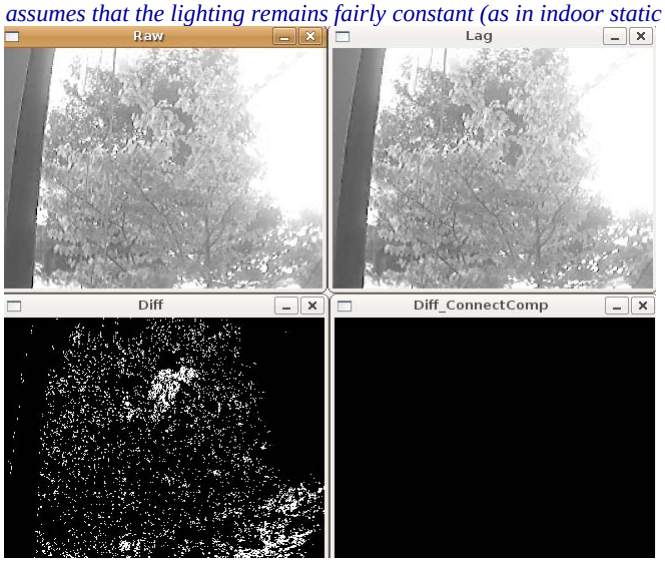
We've just seen a simple method of learning background scenes and segmenting foreground objects. It will work well only with scenes that do not contain moving background components (it would fail with a waving curtain or waving trees). It also assumes that the lighting remains fairly constant (as in indoor static scenes). You can look ahead to 

Figure 9-6 to check the performance of this averaging method.

Accumulating Means, Variances, and Covariances

The averaging background method just described made use the accumulation operator, `cv::Mat::operator+=()` to do what was essentially the simplest possible thing: to sum up a bunch of data that we could then normalize into an average. The average is a convenient statistical quantity for a lot of reasons, of course, but one often overlooked advantage of the average is the fact that it can be computed incrementally in this way.[7] This means that we can do processing *on line* without needing to accumulate all of the data before analyzing. We will now consider a slightly more sophisticated model, which can also be computed on line in this way.

Our next model will represent the intensity (or color) variation within a pixel by computing a *Gaussian model* for that variation. A one-dimensional Gaussian model is characterized by a single mean (or average)

[7] For purists, our implementation above is not exactly a purely incremental computation, as we divide by the number of samples at the end. There does, however, exist a purely incremental method for updating the average when a new data point is introduced, but the "nearly incremental" version used is substantially more computationally efficient. We will continue throughout the chapter to refer to methods as "incremental" if they can be computed from purely cumulative functions of the data combined with factors associated with overall normalization.

and a single *variance* (which tells us something about the expected spread of measured values about the mean). In the case of a d-dimensional model (e.g., a three-color model), there will be a d-dimensional vector for the mean, and a d^2-element matrix that represents not only the individual variances of the d-dimensions, but also the covariances, which represent correlations between each of the individual dimensions.

As promised, each of these quantities—the means, the variances, and the covariances—can be computed in an incremental manner. Given a stream of incoming images, we can define three functions that will accumulate the necessary data, and three functions that will actually convert those accumulations into the model parameters.

The code which follows assumes the existence of a few global variables:

```
cv::Mat sum;
cv::Mat sqsum;
int     image_count = 0;
```

Computing the Mean with cv::Mat::operator+=()

As we saw in our previous example, the best method to compute the pixel means is to add them all up using `cv::Mat::operator+=()` and then divide by the total number of images to obtain the mean:

```
void accumulateMean(
  cv::Mat& I
) {
  if( sum.empty ) {
    sum = cv::Mat::zeros( I.size(), CV::F32C(I.channels()) );
  }
  I.convertTo( scratch, sum.type() );
  sum += scratch;
  image_count++;
}
```

The function above, `accumulateMean()`, is then called on each incoming image. Once all of the images that are going to be used for the background model have been computed, you can call the next function, `computeMean()` to get a single "image" that contains the averages for every pixel across your entire input set:

```
cv::Mat& computeMean(
  cv::Mat& mean
) {
  mean = sum / image_count;
}
```

Computing the mean with `cv::accumulate()`

OpenCV provides another function, which is essentially similar to just using the `cv::Mat::operator+=()` operator, but with two important distinctions. The first is that it will automatically handle the `cv::Mat::convertTo()` functionality (and thus remove the need for a scratch image), and the second is that it allows the use of an image mask. This function is `cv::accumulate()`. The ability to use an image mask when computing a background model is a very useful thing, as one often has some other information that some part of the image should not be included in the background model. For example, one might be building a background model of a highway or other uniformly colored area, and be able to immediately determine from color that some objects are not part of the background. This sort of thing can be very helpful in a real-world situation in which there is little or no opportunity to get access to the scene in the complete absence of foreground objects.

The accumulate function has the following prototype:

```
void accumulate(

  cv::InputArray       src,                // Input, 1 or 3 channels, U8 or F32
  cv::InputOutputArray dst,                // Result image, F32 or F64
```

```
  cv::InputArray       mask = cv::noArray() // Use src pixel if mask pixel != 0
);
```

Here the array `dst` is the array in which the accumulation is happening, and `src` is the new image which will be added. `cv::accumulate()` admits an optional mask. If present, only the pixels in `dst` which correspond to nonzero elements in mask will be updated.

With `cv::accumulate()`, the function `accumulateMean()` above can be simplified to:

```
void accumulateMean(
  cv::Mat& I
) {
  if( sum.empty ) {
    sum = cv::Mat::zeros( I.size(), CV::F32C(I.channels()) );
  }
  cv::accumulate( I, sum );
  image_count++;
}
```

Variation: Computing the mean with `cv::accumulateWeighted()`

Another alternative that is often useful is to use a *running average*. The running average is given by the following formula:

$$acc(x,y) = (1-\alpha)\cdot acc(x,y) + \alpha \cdot image(x,y)$$

For a constant value of α, running averages are not equivalent to the result of summing with `cv::Mat::operator+=()`, or `cv::accumulate()`. To see this, simply consider adding three numbers (2, 3, and 4) with α set to 0.5. If we were to accumulate them with `cv::accumulate()`, then the sum would be 9 and the average 3. If we were to accumulate them with `cv::accumulateWeighted()`, the first sum would give $0.5\cdot 2 + 0.5\cdot 3 = 2.5$, and then adding the third term would give $0.5\cdot 2.5 + 0.5\cdot 4 = 3.25$. The reason the second number is larger is that the most recent contributions are given more weight than those from farther in the past. Such a running average is thus also called a *tracker*. The parameter α can be thought of as setting the amount of time necessary for the influence of a previous frame to fade.

To accumulate running averages across entire images, we use the OpenCV function `cv::accumulateWeighted()`:

```
void accumulateWeighted(
  cv::InputArray       src,                // Input, 1 or 3 channels, U8 or F32
  cv::InputOutputArray dst,                // Result image, F32 or F64
  double               alpha,              // Weight factor applied to src
  cv::InputArray       mask = cv::noArray() // Use src pixel if mask pixel != 0
);
```

Here the array `dst` is the array in which the accumulation is happening, and `src` is the new image that will be added. The value alpha is the weighting parameter. Like `cv::accumulate()`,

`cv::accumulateWeighted()` admits an optional mask. If present, only the pixels in `dst` that correspond to nonzero elements in mask will be updated.

Finding the variance with the help of `cv::accumulateSquare()`

We can also accumulate squared images, which will allow us to compute quickly the variance of individual pixels. You may recall from your last class in statistics that the variance of a finite population is defined by the formula:

$$\sigma^2 = \frac{1}{N}\sum_{i=0}^{N-1}(x_i - \bar{x})^2$$

where $\bar{x}$ is the mean of x for all N samples. The problem with this formula is that it entails making one pass

through the images to compute $\bar{x}$ and then a second pass to compute σ^2. A little algebra should allow you to convince yourself that the following formula will work just as well:

$$\sigma^2 = \left(\frac{1}{N}\sum_{i=0}^{N-1} x_i^2\right) - \left(\frac{1}{N}\sum_{i=0}^{N-1} x_i\right)^2$$

Using this form, we can accumulate both the pixel values and their squares in a single pass. Then, the variance of a single pixel is just the average of the square minus the square of the average. With this in mind, we can define an accumulation function and a computation function as we did with the mean. As with the mean, one option would be to first do an element by element squaring of the incoming image, and then to accumulate that with something like `sqsum += I.mul(I)`. This, however, has several disadvantages, the most significant of which is that `I.mul(I)` does not do any kind of implicit type conversion (as we saw that += did not do either). As a result, elements of (for example) an 8-bit array, when squared, will almost inevitably cause overflows. As with `cv::accumulate()`, however, OpenCV provides us with a convenient function that does what we need all in a single convenient package; it is called `cv::accumulateSquare()`:

```
void accumulateSquare(
  cv::InputArray       src,                // Input, 1 or 3 channels, U8 or F32
  cv::InputOutputArray dst,                // Result image, F32 or F64
  cv::InputArray       mask = cv::noArray() // Use src pixel if mask pixel != 0
);
```

With the help of `cv::accumulateSquare()`, we can write a function to accumulate the information we need for our variance computation:

```
void accumulateVariance(
  cv::Mat& I
) {
  if( sum.empty ) {
    sum   = cv::Mat::zeros( I.size(), CV::F32C(I.channels()) );
    sqsum = cv::Mat::zeros( I.size(), CV::F32C(I.channels()) );
  }
  cv::accumulate( I, sum );
  cv::accumulateSquare( I, sqsum );
  image_count++;
}
```

The associated computation function would then be:

```
// note that 'variance' is sigma^2
//
void computeVariance(
```

```
  cv::Mat& variance
) {
  double one_by_N = 1.0 / image_count;
  variance = one_by_N  * sqsum – (one_by_N * one_by_N) * sum.mul(sum);
}
```

Finding the Covariance with `cv::accumulateWeighted()`

The variance of the individual channels in a multichannel image captures some important information about how similar we *expect* background pixels in future images to be to our observed average. This, however, is still a very simplistic model for both the background and our "expectations." One important additional concept which we could introduce is the concept of covariance. Covariance captures inter-relations between the variations in individual channels. For example, our background might be an ocean scene in which we expect very little variation in the red channel, but quite a bit in the green and blue channels. If we follow our intuition that the ocean is just one color really, and that the variation we see is primarily a result of lighting effects, we might conclude that if there were a gain or loss of intensity in the green channel, there should be a *corresponding* gain or loss of intensity in the blue channel. The corollary of this is that if there were a substantial gain in the blue channel without an accompanying gain in the green channel, we might not want to consider this background. This intuition is captured by the concept of covariance.

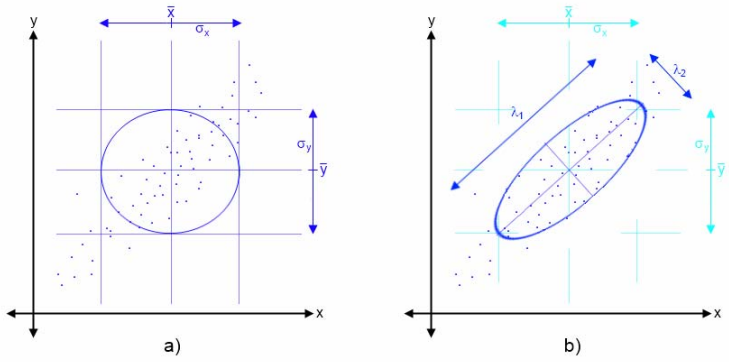


Figure 9-3: The same data set is visualized on the left and the right. On the left (a) the (square root of the) variance of the data in the x and y dimensions is shown, and the resulting model for the data is visualized. On the right (b) the covariance of the data is captured in the visualized model. The model has become an ellipsoid which is narrower in one dimension and wider in the other than the more simplistic model on the left.

In Figure 9-3, we visualize what might be the blue and green channel data for a particular pixel in our ocean background example. On the left, the variance only has been computed. On the right, the covariance between the two channels has also been computed, and the resulting model is able to fit the data much more tightly.

Mathematically, the covariance between any two different observables is given by the formula:

$$Cov(x,y) = \left(\frac{1}{N}\sum_{i=0}^{N-1}(x_i \cdot y_i)\right) - \left(\frac{1}{N}\sum_{i=0}^{N-1}x_i\right)\left(\frac{1}{N}\sum_{i=0}^{N-1}y_i\right)$$

As you can see, the covariance between any observable x and itself: $Cov(x,y)$ is equal to the variance of that same observable σ_x^2. In a d-dimensional space (such as the RGB values for a pixel, for which d=3), it

is convenient to talk about the *covariance matrix* Σ_{xy}, whose components include all of the covariances between the variables as well as the variances of the variables individually. As can be seen from the formula above, the covariance matrix is symmetric, i.e., $\Sigma_{xy} = \Sigma_{yx}$.

In a much earlier chapter, we encountered a function which could be used when dealing with individual vectors of data (as opposed to whole arrays of individual vectors). That function was

`cv::calcCovarMatrix()`, which will allow us to provide N vectors of dimension d and it will spit out

the d-by-d covariance matrix. Our problem now, however, is that we would like to compute such a matrix

for every point in an array (or at least, in the case of a 3-dimensional RGB image, we would like to compute the 6 unique entries in that matrix).

In practice, the best way to do this is simply to compute the variances using the code we already developed, and to compute the three new objects (the off-diagonal elements of Σ_{xy}) separately. Looking at the formula for the covariance, we see that `cv::accumulateSquare()` will not quite work here, as we need to accumulate the $(x_i \cdot y_i)$ terms (i.e., the product of two different channel values from a particular pixel in each image).

The function which does this for us in OpenCV is `cv::accumulateProduct()`.

```
void accumulateProduct(
  cv::InputArray       src1,               // Input, 1 or 3 channels, U8 or F32
  cv::InputArray       src2,               // Input, 1 or 3 channels, U8 or F32
  cv::InputOutputArray dst,                // Result image, F32 or F64
  cv::InputArray       mask = cv::noArray() // Use src pixel if mask pixel != 0
);
```

This function works exactly like `cv::accumulateSquare()`, except that rather than squaring the individual elements of `src`, it multiplies the corresponding elements of `src1` and `src2`. What it does not do (unfortunately) is allow us to pluck individual channels out of those incoming arrays. In the case of multichannel arrays in `src1` and `src2`, the computed result is done on a per-channel basis.

For our current need to compute the off-diagonal elements of a covariance model, this is not really what we want. What we want are different channels of the *same* image. To do this, we will have to split our incoming image apart using `cv::split()`.

```
vector<cv::Mat> planes(3);
vector<cv::Mat> sums(3);
vector<cv::Mat> xysums(6);

int image_count = 0;

void accumulateCovariance(
  cv::Mat& I
) {
  int i, j, n;
  if( sum.empty ) {
    for( i=0; i<3; i++ ) {    // the r, g, and b sums
      sums[i]   = cv::Mat::zeros( I.size(), CV::F32C1 );
    }
    for( n=0; n<6; n++ ) {    // the rr, rg, rb, gg, gb, and bb elements
      xysums[n] = cv::Mat::zeros( I.size(), CV::F32C1 ) );
    }
  }
```

```
  cv::split( I, rgb );
  for( i=0; i<3; i++ ) {
    cv::accumulate( rgb[i], sums[i] );
  }
  n = 0;
  for( i=0; i<3; i++ ) {      // "row" of Sigma
    for( j=i; j<3; j++ ) {    // "column" of Sigma
      n++;
      cv::accumulateProduct( rgb[i], rgb[j], xysums[n] );
    }
  }
  image_count++;
}
```

The corresponding compute function is also just a slight extension of the compute function for the variances we saw earlier.

```
// note that 'variance' is sigma^2
//
void computeVariance(
  cv::Mat& covariance      // a six-channel array, channels are the
                           // rr, rg, rb, gg, gb, and bb elements of Sigma_xy
) {
  double one_by_N = 1.0 / image_count;

  // reuse the xysum arrays as storage for individual entries
  //
  int n = 0;
  for( int i=0; i<3; i++ ) {     // "row" of Sigma
    for( int j=i; j<3; j++ ) {   // "column" of Sigma
      n++;
      xysums[n] = one_by_N  * xysums[n]
        - (one_by_N * one_by_N) * sums[i].mul(sums[j]);
    }
  }

  // reassemble the six individual elements into a six-channel array
  //
  cv::merge( xysums, covariance );
}
```

A Brief Note on Model Testing and `cv::Mahalanobis()`

In this section, we introduced some slightly more complicated models, but did not discuss how to test if a particular pixel in a new image is in the predicted domain of variation for the background model. In the case of the variance-only model (Gaussian models on all channels with an implicit assumption of statistical independence between the channels) the problem is made more complicated by the fact that the variances for the individual dimensions will not necessarily be equal. In this case, however, it is common to compute what is called a z-score for each dimension separately (the z-score is the distance from the mean divided by the standard deviation: $(x - \bar{x})/\sigma_x$). The z-score tells us something about the probability of the individual pixel originating from the distribution in question. The z-scores for multiple dimensions are then summarized as the square root of sum of squares (e.g., $\sqrt{z_{red}^2 + z_{green}^2 + z_{blue}^2}$).

In the case of the full covariance matrix, the analog of the z-score is called the *Mahalanobis distance*. This distance is essentially the distance from the mean to the point in question measured in constant-probability contours such as that shown in Figure 9-3. Looking back at Figure 9-3, a point up and to the left of the mean in the model (a) will appear to have a low Mahalanobis distance by that model. The same point would have a much higher Mahalanobis distance by the model in (b). It is worth noting that the z-score formula for the simplified model in the previous paragraph is precisely the Mahalanobis distance under the model in Figure 9-3 (a), as one would expect.

OpenCV provides a function for computing Mahalanobis distances:

```
double cv::Mahalanobis(                    // Return distance as F64
  cv::InputArray vec1,                     // First vector (1-dimensional, length n)
  cv::InputArray vec2,                     // Second vector (1-dimensional, length n)
  cv::InputArray icovar                    // Inverse covariance matrix, n-by-n
);
```

The `cv::Mahalanobis()` function expects vector objects for `vec1` and `vec2` of dimension d and a d-by-d matrix for the *inverse* covariance `icovar`. (The inverse covariance is used because inverting this matrix is costly, and in most cases you have many vectors you would like to compare with the same covariance—so the assumption is that you will invert it once and pass the inverse covariance to `cv::Mahalanobis()` many times for each such inversion.)

> In our context of background subtraction, this is not entirely convenient, as `cv::Mahalanobis()` wants to be called on a per-element basis. Unfortunately, there is no array-sized version of this capability in OpenCV. As a result, you will have to loop through each pixel, create the covariance matrix from the individual elements, invert that matrix, and store the inverse somewhere. Then, when you want to make a comparison, you will need to loop through the pixels in your image, retrieve the inverse covariance you need, and call `cv::Mahalanobis()` for each pixel.

A More Advanced Background Subtraction Method

Many background scenes contain complicated moving objects such as trees waving in the wind, fans turning, curtains fluttering, and so on. Often, such scenes also contain varying lighting, such as clouds passing by or doors and windows letting in different light.

A nice method to deal with this would be to fit a time-series model to each pixel or group of pixels. This kind of model deals with the temporal fluctuations well, but its disadvantage is the need for a great deal of memory [Toyama99]. If we use 2 seconds of previous input at 30 Hz, this means we need 60 samples for each pixel. The resulting model for each pixel would then encode what it had learned in the form of 60 different adapted *weights*. Often we'd need to gather background statistics for much longer than 2 seconds, which means that such methods are typically impractical on present-day hardware.

To get fairly close to the performance of adaptive filtering, we take inspiration from the techniques of video compression and attempt to form a *codebook*[8] to represent significant states in the background.[9] The simplest way to do this would be to compare a new value observed for a pixel with prior observed values. If the value is close to a prior value, then it is modeled as a perturbation on that color. If it is not close, then it can seed a new group of colors to be associated with that pixel. The result could be envisioned as a bunch

[8] The method OpenCV implements is derived from Kim, Chalidabhongse, Harwood, and Davis [Kim05], but rather than learning oriented cylinders in RGB space, for speed, the authors use axis-aligned boxes in YUV space. Fast methods for cleaning up the resulting background image can be found in Martins [Martins99].

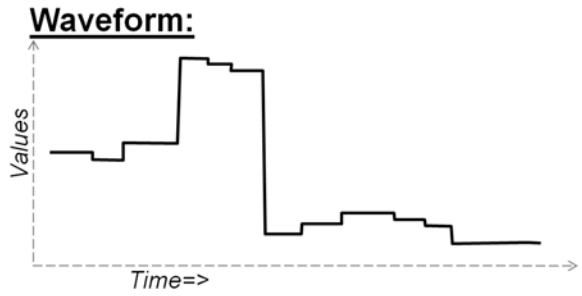
[9] There is a large literature for background modeling and segmentation. OpenCV's implementation is intended to be fast and robust enough that you can use it to collect foreground objects mainly for the purposes of collecting data sets to train classifiers on. Recent work in background subtraction allows arbitrary camera motion [Farin04; Colombari07] and dynamic background models using the mean-shift algorithm [Liu07].

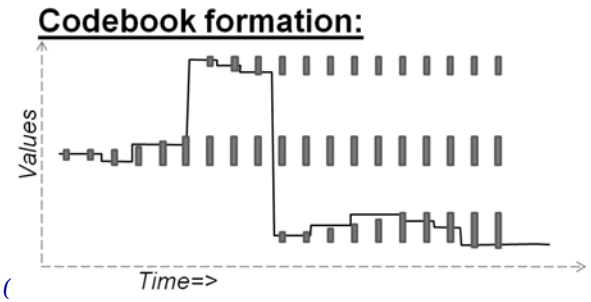
of blobs floating in RGB space, each blob representing a separate volume considered likely to be background.

In practice, the choice of RGB is not particularly optimal. It is almost always better to use a color space whose axis is aligned with brightness, such as the YUV color space. (YUV is the most common choice, but spaces such as HSV, where V is essentially brightness, would work as well.) The reason for this is that, empirically, most of the natural variation in the background tends to be along the brightness axis, not the color axis.

The next detail is how to model these "blobs." We have essentially the same choices as before with our simpler model. We could, for example, choose to model the blobs as Gaussian clusters with a mean and a covariance. It turns out that the simplest case, in which the "blobs" are simply boxes with a learned extent in each of the three axes of our color space, works out quite well. It is the simplest in terms of memory required and in terms of the computational cost of determining whether a newly observed pixel is inside any of the learned boxes.

Let's explain what a codebook is by using a simple example





(

Figure 9-4). A codebook is made up of boxes that grow to cover the common values seen over time. The upper panel of

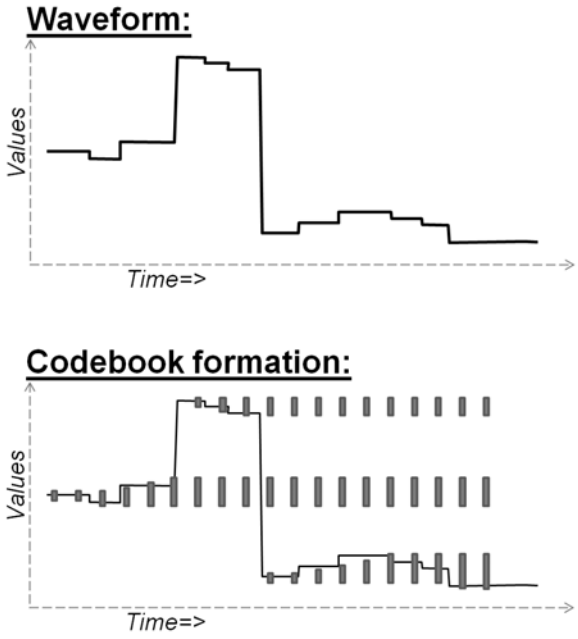


Figure 9-4 shows a waveform over time; you could think of this as the brightness of an individual pixel. In the lower panel, boxes form to cover a new value and then slowly grow to cover nearby values. If a value is too far away, then a new box forms to cover it and likewise grows slowly toward new values.

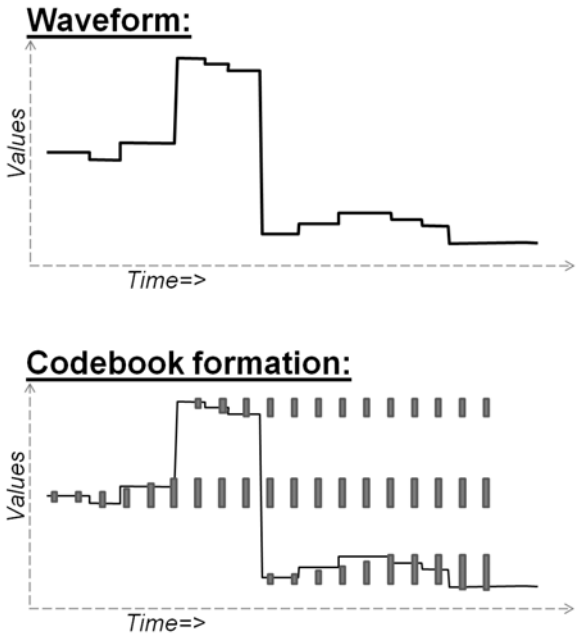


Figure 9-4: Codebooks are just "boxes" delimiting intensity values: a box is formed to cover a new value and slowly grows to cover nearby values; if values are too far away then a new box is formed (see text)

In the case of our background model, we will learn a codebook of boxes that cover three dimensions: the three channels that make up our image at each pixel.

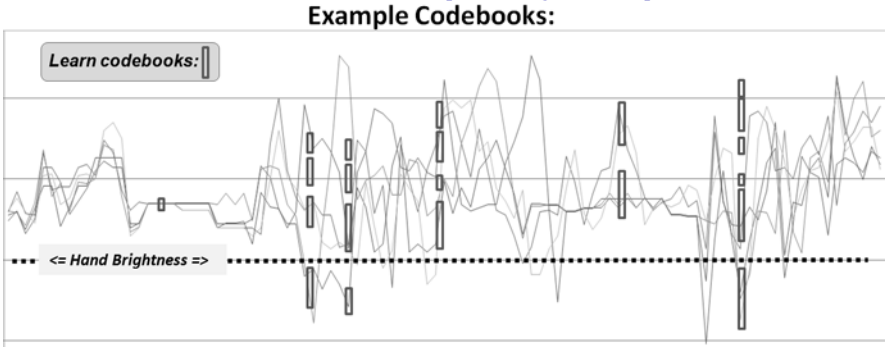


Figure 9-5 visualizes the (intensity dimension of the) codebooks for six different pixels learned from the data in Figure 9-1.[10] *This codebook method can deal with pixels that change levels dramatically (e.g., pixels in a windblown tree, which might alternately be one of many colors of leaves, or the blue sky beyond that tree). With this more precise method of modeling, we can detect a foreground object that has values between the pixel values. Compare this with Figure 9-2, where the averaging method cannot distinguish the hand value (shown as a dotted line) from the pixel fluctuations. Peeking ahead to the next section, we see the better performance of the codebook method versus the averaging method shown later in*

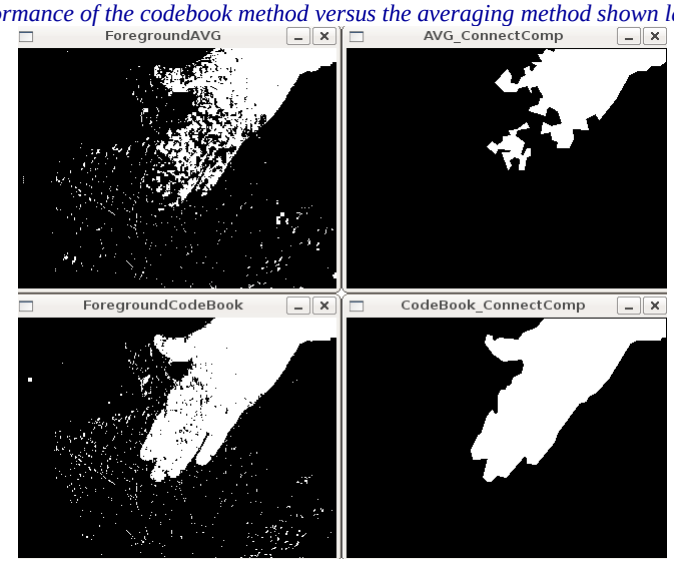
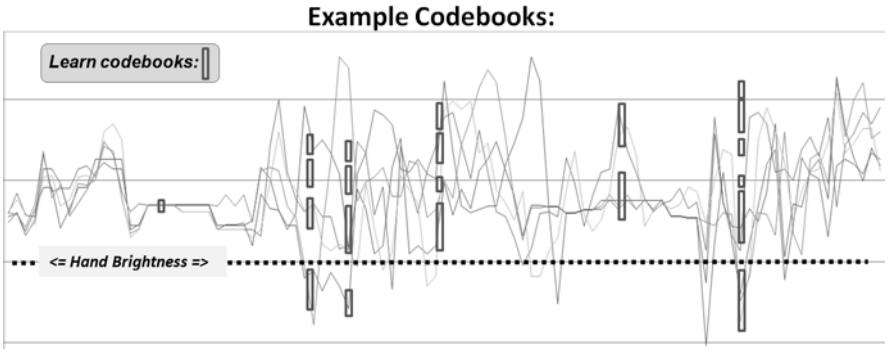


Figure 9-8.



[10] In this case, we have chosen several pixels at random from the scan line to avoid excessive clutter. Of course, there is actually a codebook for every pixel.

Figure 9-5: Intensity portion of learned codebook entries for fluctuations of six chosen pixels (shown as vertical boxes): codebook boxes accommodate pixels that take on multiple discrete values and so can better model discontinuous distributions; thus they can detect a foreground hand (value at dotted line) whose average value is between the values that background pixels can assume. In this case the codebooks are one-dimensional and only represent variations in intensity

In the codebook method of learning a background model, each box is defined by two thresholds (`max` and `min`) over each of the three-color axes. These box boundary thresholds will expand (`max` getting larger, `min` getting smaller) if new background samples fall within a learning threshold (`learnHigh` and `learnLow`) above `max` or below `min`, respectively. If new background samples fall outside of the box and its learning thresholds, then a new box will be started. In the *background difference* mode, there are acceptance thresholds `maxMod` and `minMod`; using these threshold values, we say that if a pixel is "close enough" to a `max` or a `min` box boundary then we count it as if it were inside the box. At runtime, the threshold for inclusion in a "box" can be set to a different value than was used in the construction of the boxes; often this threshold is simply set to zero in all three dimensions.

> A situation we will not cover is a pan-tilt camera surveying a large scene. When working with a large scene, it is necessary to stitch together learned models indexed by the pan and tilt angles.

Structures

It's time to look at all of this in more detail, so let's create an implementation of the codebook algorithm. First, we need our codebook structure, which will simply point to a bunch of boxes in YUV space:

```
class CodeBook : public vector<CodeElement> {
public:
  int t;                                              // count every access
  CodeBook() { t=0; }                                 // Default is an empty book
  CodeBook( int n ) : vector<CodeElement>(n) { t=0; } // Construct book of size n
};
```

The codebook is derived from an STL vector of `CodeElement` objects (see below). The variable `t` counts the number of points we've accumulated since the start or the last clear operation. Here's how the actual codebook elements are described:

```
#define CHANNELS 3
class CodeElement {
public:
  uchar learnHigh[CHANNELS];   // High side threshold for learning
  uchar learnLow[CHANNELS];    // Low side threshold for learning
  uchar max[CHANNELS];         // High side of box boundary
  uchar min[CHANNELS];         // Low side of box boundary
  int   t_last_update;         // Allow us to kill stale entries
  int   stale;                 // max negative run (longest period of inactivity)

  CodeElement& operator=( CodeElement& ce ) {
    for( i=0; i<CHANNELS; i++ ) {
      learnHigh[i] = ce.learnHigh[i];
      learnLow[i]  = ce.learnLow[i];
      min[i]       = ce.min[i];
      max[i]       = ce.max[i];
    }
    t_last_update = ce.t_last_update;
    stale         = ce.stale;
  }
  CodeElement( CodeElement& ce ) { *this = ce; }
};
```

Each codebook entry consumes four bytes per channel plus two integers, or (4 * `CHANNELS` + 4 + 4) bytes (20 bytes when we use three channels). We may set `CHANNELS` to any positive number equal to or less

than the number of color channels in an image, but it is usually set to either 1 (“Y,” or brightness only) or 3 (YUV, HSV). In this structure, for each channel, `max` and `min` are the boundaries of the codebook box. The parameters `learnHigh[]` and `learnLow[]` are the thresholds that trigger generation of a new code element. Specifically, a new code element will be generated if a new pixel is encountered whose values do not lie between `min – learnLow` and `max + learnHigh` in each of the channels. The time to last update (`t_last_update`) and `stale` are used to enable the deletion of seldom-used codebook entries created during learning. Now we can proceed to investigate the functions that use this structure to learn dynamic backgrounds.

Learning the background

We will have one `CodeBook` of `CodeElements` for each pixel. We will need an array of such codebooks that is equal in length to the number of pixels in the images we’ll be learning. For each pixel, `updateCodebook()` is called for as many images as are sufficient to capture the relevant changes in the background. Learning may be updated periodically throughout, and `clearStaleEntries()` can be used to learn the background in the presence of (small numbers of) moving foreground objects. This is possible because the seldom-used “stale” entries induced by a moving foreground will be deleted. The interface to `updateCodebook()` is as follows:

```
// Updates the codebook entry with a new data point
// NOTES:
//      cbBounds must be of length equal to numChannels
//
int updateCodebook(       // return CodeBook index
  cv::Vec3b& p,           // incoming YUV pixel
  CodeBook&  c,           // CodeBook for the pixel
  unsigned*  cbBounds,    // Learning bounds for codebook (Rule of thumb: {10,10,10})
  int        numChannels  // Number of color channels we're learning
) {
  unsigned int high[3], low[3], n;
  for( n=0; n<numChannels; n++ ) {
    high[n] = p[n] + *(cbBounds+n);    if( high[n] > 255 ) high[n] = 255;
    low[n]  = p[n] - *(cbBounds+n);    if( low[n]  < 0   ) low[n]  = 0;
  }
  int matchChannel;
  // SEE IF THIS FITS AN EXISTING CODEWORD
  //
  int i;
  for( i=0; i<c.size(); i++ ) {
    matchChannel = 0;
    for( n=0; n<numChannels; n++ ) {
      if( // Found an entry for this channel
       ( c[i]->learnLow[n] <= p[n] ) && ( p[n] <= c[i]->learnHigh[n])
      )
      matchChannel++;
    }
    if( matchChannel == numChannels ) {     // If an entry was found
      c[i]->t_last_update = c.t;

      // adjust this codeword for the first channel
      //
      for( n=0; n<numChannels; n++ ) {
        if( c[i]->max[n] < p[n] )       c[i]->max[n] = p[n];
        else if( c[i]->min[n] > p[n] )  c[i]->min[n] = p[n];
      }
      break;
    }
  }
. . .continued below
```

This function grows or adds a codebook entry when the pixel `p` falls outside the existing codebook boxes. Boxes grow when the pixel is within `cbBounds` of an existing box. If a pixel is outside the `cbBounds` distance from a box, a new codebook box is created. The routine first sets `high` and `low` levels to be used later. It then goes through each codebook entry to check whether the pixel value `p` is inside the learning bounds of the codebook "box." If the pixel is within the learning bounds for all channels, then the appropriate `max` or `min` level is adjusted to include this pixel and the time of last update is set to the current timed count `c.t`. Next, the `updateCodebook()` routine keeps statistics on how often each codebook entry is hit:

```
. . . continued from above

  // OVERHEAD TO TRACK POTENTIAL STALE ENTRIES
  //
  for( int s=0; s<c.size(); s++ ) {
    // Track which codebook entries are going stale:
    //
    int negRun = c.t - c[s]->t_last_update;
    if( c[s]->stale < negRun ) c[s]->stale = negRun;
  }

. . .continued below
```

Here, the variable `stale` contains the largest *negative runtime* (i.e., the longest span of time during which that code was not accessed by the data). Tracking stale entries allows us to delete codebooks that were formed from noise or moving foreground objects and hence tend to become stale over time. In the next stage of learning the background, `updateCodebook()` adds a new codebook if needed:

```
. . . continued from above

  // ENTER A NEW CODEWORD IF NEEDED
  //
  if( i == c.size() ) {          // if no existing codeword found, make one
    CodeElement ce;
    for( n=0; n<numChannels; n++ ) {
      ce->learnHigh[n] = high[n];
      ce->learnLow[n]  = low[n];
      ce->max[n]       = p[n];
      ce->min[n]       = p[n];
    }
    ce->t_last_update = c.t;
    ce->stale = 0;
    c.push_back( ce );
  }
. . .continued below
```

Finally, `updateCodebook()` slowly adjusts (by adding 1) the `learnHigh` and `learnLow` learning boundaries if pixels were found outside of the box thresholds but still within the `high` and `low` bounds:

```
. . . continued from above

  // SLOWLY ADJUST LEARNING BOUNDS
  //
  for( n=0; n<numChannels; n++ ) {
    if( c[i]->learnHigh[n] < high[n]) c[i]->learnHigh[n] += 1;
    if( c[i]->learnLow[n]  > low[n] ) c[i]->learnLow[n]  -= 1;
  }
  return i;
}
```

The routine concludes by returning the index of the modified codebook. We've now seen how codebooks are learned. In order to learn in the presence of moving foreground objects and to avoid learning codes for spurious noise, we need a way to delete entries that were accessed only rarely during learning.

Learning with moving foreground objects

The following routine, `clearStaleEntries()`, allows us to learn the background even if there are moving foreground objects:

```
// During learning, after you've learned for some period of time,
// periodically call this to clear out stale codebook entries
//
//
int clearStaleEntries(     // return number of entries cleared
  CodeBook &c              // Codebook to clean up
){
  int staleThresh = c.t>>1;
  int *keep = new int[c.size()];
  int keepCnt = 0;

  // SEE WHICH CODEBOOK ENTRIES ARE TOO STALE
  //
  for( int i=0; i<c.size(); i++ ){
    if(c[i]->stale > staleThresh)
      keep[i] = 0; // Mark for destruction
    else
    {
      keep[i] = 1; // Mark to keep
      keepCnt += 1;
    }
  }

  // move the entries we want to keep to the front of the vector and then
  // truncate to the correct length once all of the good stuff is saved.
  //
  int k = 0;
  int numCleared = 0
  for( int ii=0; ii<c.size(); ii++ ) {
    if( keep[ii] ) {
      c[k] = c[ii];
      // We have to refresh these entries for next clearStale
      cc[k]->t_last_update = 0;
      k++;
    } else {
      numCleared++;
    }
  }
  c.resize( keepCnt );
  delete[] keep;

  return numCleared;
}
```

The routine begins by defining the parameter `staleThresh`, which is hardcoded (by a rule of thumb) to be half the total running time count, `c.t`. This means that, during background learning, if codebook entry `i` is not accessed for a period of time equal to half the total learning time, then `i` is marked for deletion (`keep[i] = 0`). The vector `keep[]` is allocated so that we can mark each codebook entry; hence it is `c.size()` long. The variable `keepCnt` counts how many entries we will keep. After recording which codebook entries to keep, we go through the entries and move the ones we want to the front of the vector in the codebook. Finally, we resize that vector so that all of the stuff hanging off of the end is chopped off.

Background differencing: Finding foreground objects

We've seen how to create a background codebook model and how to clear it of seldom-used entries. Next we turn to `background_diff()`, where we use the learned model to segment foreground pixels from the previously learned background:

```
// Given a pixel and a codebook, determine if the pixel is
// covered by the codebook
//
// NOTES:
// minMod and maxMod must have length numChannels,
// e.g. 3 channels => minMod[3], maxMod[3]. There is one min and
//      one max threshold per channel.
//
uchar backgroundDiff(     // return 0 => background, 255 => foreground
  cv::Vec3b& p,            // Pixel (YUV)
  CodeBook&  c,            // Codebook
  int        numChannels,  // Number of channels we are testing
  int*       minMod,       // Add this (possibly negative) number onto max level
                           //   when determining if new pixel is foreground
  int*       maxMod        // Subtract this (possibly negative) number from min level
                           //   when determining if new pixel is foreground
) {
  int matchChannel;

  // SEE IF THIS FITS AN EXISTING CODEWORD
  //
  for( int i=0; i<c.size(); i++ ) {
    matchChannel = 0;
    for( int n=0; n<numChannels; n++ ) {
      if(
        (c[i]->min[n] - minMod[n] <= p[n] ) && (p[n] <= c[i]->max[n] + maxMod[n])
      ) {
        matchChannel++; // Found an entry for this channel
      } else {
        break;
      }
    }
    if(matchChannel == numChannels) {
      break; // Found an entry that matched all channels
    }
  }
  if( i >= c.size() ) return 0;
  return 255;
}
```

The background differencing function has an inner loop similar to the learning routine `updateCodebook`, except here we look within the learned `max` and `min` bounds plus an offset threshold, `maxMod` and `minMod`, of each codebook box. If the pixel is within the box plus `maxMod` on the high side or minus `minMod` on the low side for each channel, then the `matchChannel` count is incremented. When `matchChannel` equals the number of channels, we've searched each dimension and know that we have a match. If the pixel is not within a learned box, 255 is returned (a positive detection of foreground); otherwise, 0 is returned (the pixel is background).

The three functions `updateCodebook()`, `clearStaleEntries()`, and `backgroundDiff()` constitute a codebook method of segmenting foreground from learned background.

Using the Codebook Background Model

To use the codebook background segmentation technique, typically we take the following steps:

1. Learn a basic model of the background over a few seconds or minutes using `updateCodebook()`.
2. Clean out stale entries with `clearStaleEntries()`.
3. Adjust the thresholds `minMod` and `maxMod` to best segment the known foreground.
4. Maintain a higher-level scene model (as discussed previously).
5. Use the learned model to segment the foreground from the background via `backgroundDiff()`.
6. Periodically update the learned background pixels.
7. At a much slower frequency, periodically clean out stale codebook entries with `clearStaleEntries()`.

A Few More Thoughts on Codebook Models

In general, the codebook method works quite well across a wide number of conditions, and it is relatively quick to train and to run. It doesn't deal well with varying patterns of light—such as morning, noon, and evening sunshine—or with someone turning lights on or off indoors. This type of global variability can be taken into account by using several different codebook models, one for each condition, and then allowing the condition to control which model is active.

Connected Components for Foreground Cleanup

Before comparing the averaging method to the codebook method, we should pause to discuss ways to clean up the raw segmented image using connected-components analysis. This form of analysis is useful for noisy input mask images, and such noise is more the rule than the exception. The basic idea behind the method is to use the morphological operation *open* to shrink areas of small noise to 0 followed by the morphological operation *close* to rebuild the area of surviving components that was lost in opening. Thereafter, we find the "large enough" contours of the surviving segments and can take statistics of all such segments. Finally, we retrieve either the largest contour or all contours of size above some threshold. In the routine that follows, we implement most of the functions that you would want for this connected component analysis:

- Whether to approximate the surviving component contours by polygons or by convex hulls
- Setting how large a component contour must be in order not to be deleted
- Returning the bounding boxes of the surviving component contours
- Returning the centers of the surviving component contours

The connected components header that implements these operations is as follows:

```
// This cleans up the foreground segmentation mask derived from calls
// to backgroundDiff
//
void findConnectedComponents(
  cv::Mat&   mask,              // Is a grayscale (8-bit depth) "raw" mask image that
                                // will be cleaned up
  int        poly1_hull0 = 1,   // If set, approximate connected component by
                                //  (DEFAULT) polygon, or else convex hull (0)
  float      perimScale  = 4,   // Len = image (width+height)/perimScale. If contour
                                //  len < this, delete that contour (DEFAULT: 4)
  vector<cv::Rect>&  bbs        // Reference to bounding box rectangle return vector
  vector<cv::Point>& centers    // Reference to contour centers return vector
 );
```

The function body is listed below. First, we do morphological opening and closing in order to clear out small pixel noise, after which we rebuild the eroded areas that survive the erosion of the opening operation. The routine takes two additional parameters, which here are hardcoded via `#define`. The defined values work well, and you are unlikely to want to change them. These additional parameters control how simple

the boundary of a foreground region should be (higher numbers are more simple) and how many iterations the morphological operators should perform; the higher the number of iterations, the more erosion takes place in opening before dilation in closing.[11] More erosion eliminates larger regions of blotchy noise at the cost of eroding the boundaries of larger regions. Again, the parameters used in this sample code work well, but there's no harm in experimenting with them if you like:

```
// polygons will be simplified using DP algorithm with 'epsilon' a fixed
// fraction of the polygons length. This number is that divisor.
//
#define DP_EPSILON_DENOMINATOR 20.0

// How many iterations of erosion and/or dilation there should be
//
#define CVCLOSE_ITR 1
```

We now discuss the connected-component algorithm itself. The first part of the routine performs the morphological open and closing operations:

```
void findConnectedComponents(
  cv::Mat&   mask,
  int        poly1_hull0,
  float      perimScale,
  vector<cv::Rect>&  bbs,
  vector<cv::Point>& centers
) {

  // CLEAN UP RAW MASK
  //
  cv::morphologyEx(
    mask, mask, cv::MOP_OPEN,  cv::Mat(), cv::Point(-1,-1), CVCLOSE_ITR
  );
  cv::morphologyEx(
    mask, mask, cv::MOP_CLOSE, cv::Mat(), cv::Point(-1,-1), CVCLOSE_ITR
  );
```

Now that the noise has been removed from the mask, we find all contours:

```
  // FIND CONTOURS AROUND ONLY BIGGER REGIONS
  //
  vector< vector<cv::Point> > contours_all; // all contours found
  vector< vector<cv::Point> > contours;     // just the ones we want to keep
  cv::findContours(
    mask,
    contours_all,
    CV_RETR_EXTERNAL,
    CV_CHAIN_APPROX_SIMPLE
  );
```

Next, we toss out contours that are too small and approximate the rest with polygons or convex hulls:

```
  for(
    vector< vector<cv::Point> >::iterator c = contours_all.begin();
    c != contours.end();
    ++c
  ) {

    // length of this contour
    //
    int len = cv::arcLength( *c, true );

    // length threshold a fraction of image perimeter
```

[11] Observe that the value CVCLOSE_ITR is actually dependent on the resolution. For images of extremely high resolution, leaving this value set to 1 is not likely to yield satisfactory results.

```
    //
    double q = (mask->height + mask->width) / DP_EPSILON_DENOMINATOR;

    if( len >= q ) {        // If the contour is long enough to keep...
      vector<cv::Point> c_new;
      if( poly1_hull0 ) {   // If the caller wants results as reduced polygons...
        cv::approxPolyDP( *c, c_new, len/20.0, true );
      } else {              // Convex Hull of the segmentation
        Cv::convexHull( *c, c_new );
      }
      contours.push_back(c_new );
    }
  }
```

In the preceding code, we use the Douglas-Peucker approximation algorithm to reduce polygons (if the user has not asked us to just return convex hulls). All this processing yields a new list of contours. Before drawing the contours back into the mask, we define some simple colors to draw:

```
  // Just some convenience variables
  const cv::Scalar CVX_WHITE   = cv::RGB(0xff,0xff,0xff);
  const cv::Scalar CVX_BLACK   = cv::RGB(0x00,0x00,0x00);
```

We use these definitions in the following code, where we first analyze each of the contours separately, then zero out the mask and draw the whole set of clean contours back into the mask:

```

  // CALC CENTER OF MASS AND/OR BOUNDING RECTANGLES
  //
  int idx = 0;
  cv::Moments moments;
  cv::Mat scratch = mask.clone();
  for(
    vector< vector<cv::Point> >::iterator c = contours.begin();
    c != contours.end;
    c++, idx++
  ) {

    cv::drawContours( scratch, contours, idx, CVX_WHITE, CV_FILLED );

    // Find the center of each contour
    //
    moments = cv::moments( scratch, true );
    cv::Point p;
    p.x = (int)( moments.m10 / moments.m00 );
    p.y = (int)( moments.m01 / moments.m00 );
    centers.push_back(p);

    bbs.push_back( cv::boundingRect(c) );

    Scratch.setTo( 0 );
  }

  // PAINT THE FOUND REGIONS BACK INTO THE IMAGE
  //
  mask.setTo( 0 );
  cv::drawContours( mask, contours, -1, CVX_WHITE );
}
```

That concludes a useful routine for creating clean masks out of noisy raw masks

A Quick Test

We start with an example to see how this really works in an actual video. Let's stick with our video of the tree outside of the window. Recall (Figure 9-1) that at some point in time, a hand passes through the scene.

One might expect that we could find this hand relatively easily with a technique such as frame differencing (discussed previously in its own section). The basic idea of frame differencing was to subtract the current frame from a "lagged" frame and then threshold the difference.

Sequential frames in a video tend to be quite similar. Hence one might expect that, if we take a simple difference of the original frame and the lagged frame, we'll not see too much unless there is some foreground object moving through the scene.[12] But what does "not see too much" mean in this context? Really, it means "just noise." Thus, in practice the problem is sorting out that noise from the signal when a foreground object does come along.

To understand this noise a little better, first consider a pair of frames from the video in which there is no foreground object—just the background and the resulting noise.

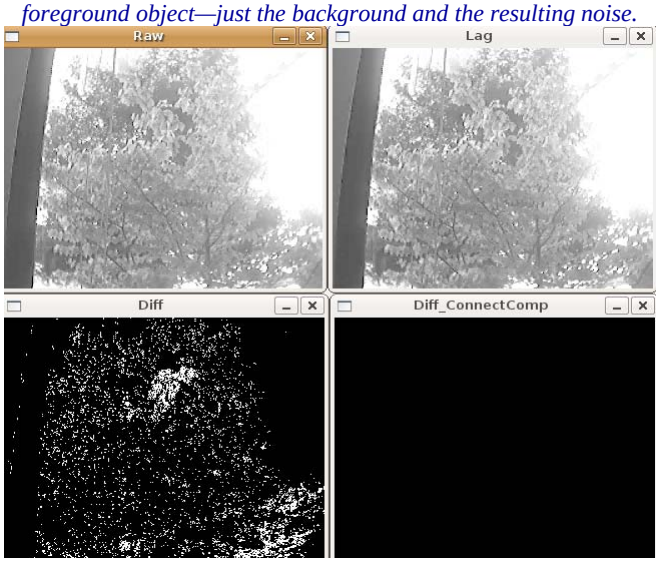


Figure 9-6 shows a typical frame from such a video (upper-left) and the previous frame (upper-right). The figure also shows the results of frame differencing with a threshold value of 15 (lower-left). You can see substantial noise from the moving leaves of the tree. Nevertheless, the method of connected components is able to clean up this scattered noise quite well[13] (lower-right). This is not surprising, because there is no

[12] In the context of frame differencing, an object is identified as "foreground" mainly by its velocity. This is reasonable in scenes that are generally static or in which foreground objects are expected to be much closer to the camera than background objects (and thus appear to move faster by virtue of the projective geometry of cameras).

[13] *The size threshold for the connected components has been tuned to give zero response in these empty frames. The real question then is whether or not the foreground object of interest (the hand) survives*

reason to expect much spatial correlation in this noise and so its signal is characterized by a large number of very small regions.

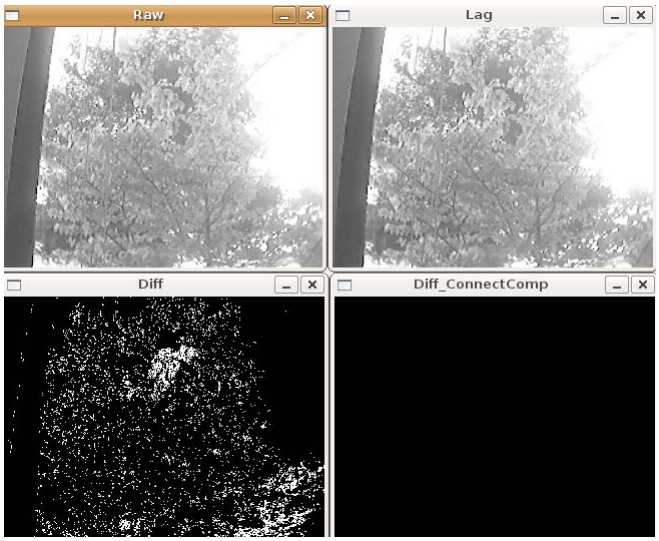


Figure 9-6: Frame differencing: a tree is waving in the background in the current (upper-left) and previous (upper-right) frame images; the difference image (lower-left) is completely cleaned up (lower-right) by the connected-components method

---

pruning at this size threshold. We will see

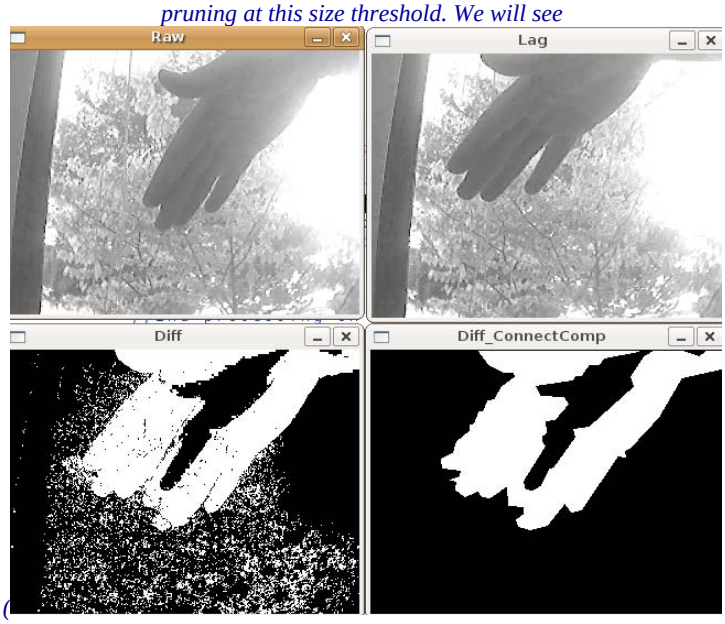


Figure 9-7) that it does so nicely.

Now consider the situation in which a foreground object (our ubiquitous hand) passes through the view of the imager.

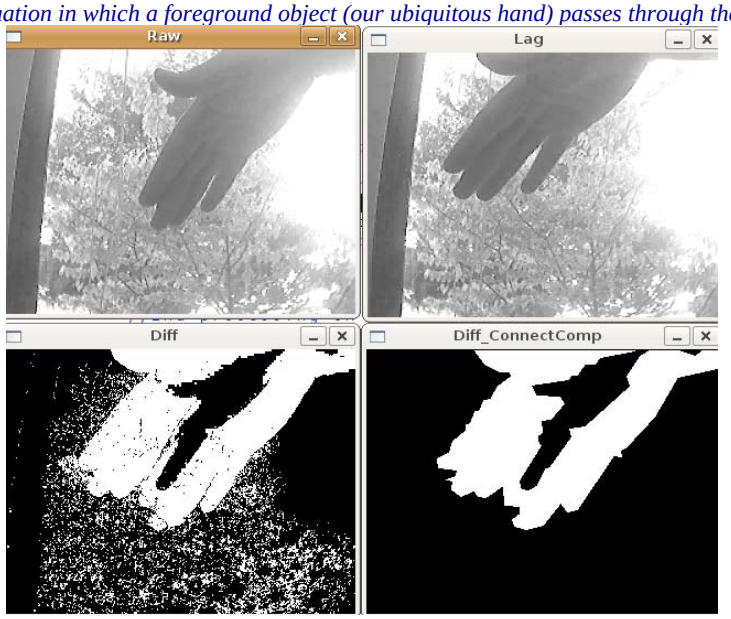


Figure 9-7 shows two frames that are similar to those in

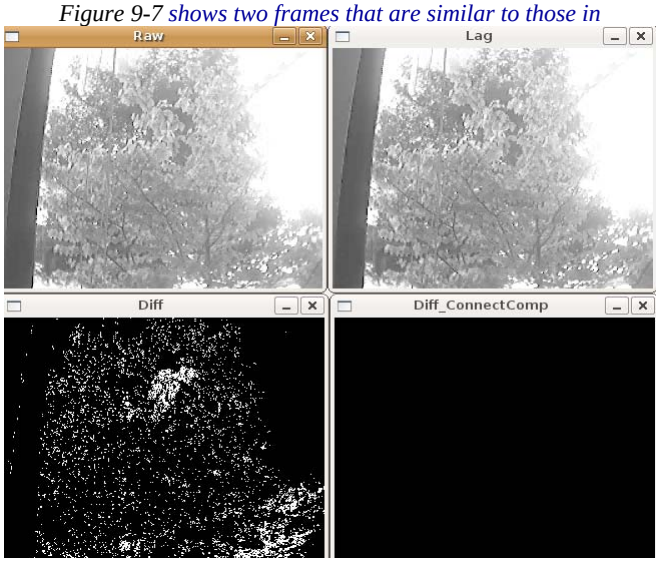


Figure 9-6 except that now there is a hand moving across from left to right. As before, the current frame (upper-left) and the previous frame (upper-right) are shown along with the response to frame differencing (lower-left) and the fairly good results of the connected-component cleanup (lower-right).

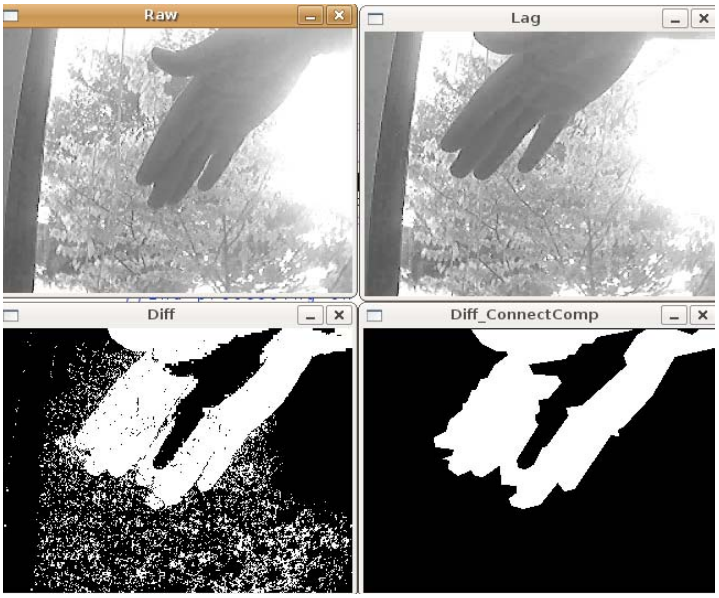


Figure 9-7: Frame difference method of detecting a hand, which is moving left to right as the foreground object (upper two panels); the difference image (lower-left) shows the "hole" (where the hand used to be) toward the left and its leading edge toward the right, and the connected-component image (lower-right) shows the cleaned-up difference

We can also clearly see one of the deficiencies of frame differencing: it cannot distinguish between the region from where the object moved (the "hole") and where the object is now. Furthermore, in the overlap region, there is often a gap because "flesh minus flesh" is 0 (or at least below threshold).

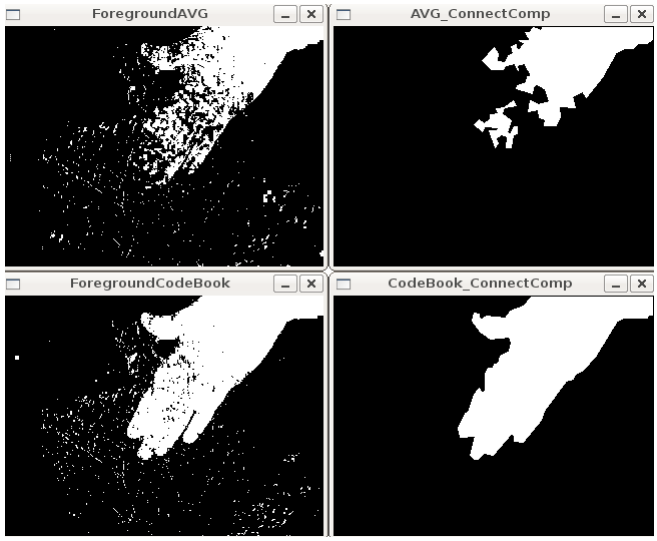
Thus, we see that using connected components for cleanup is a powerful technique for rejecting noise in background subtraction. As a bonus, we were also able to glimpse some of the strengths and weaknesses of frame differencing.

Comparing Two Background Methods

We have discussed two classes of background modeling techniques so far in this chapter: the average distance method (and its variants) and the codebook method. You might be wondering which method is better, or, at least, when you can get away with using the easy one. In these situations, it's always best to just do a straight bake off[14] between the available methods.

We will continue with the same tree video that we've been using throughout the chapter. In addition to the moving tree, this film has a lot of glare coming off a building to the right and off portions of the inside wall on the left. It is a fairly challenging background to model.

[14] For the uninitiated, "bake off" is actually a bona fide term used to describe any challenge or comparison of multiple algorithms on a predetermined data set.



In Figure 9-8, we compare the average difference method at top against the codebook method at bottom; on the left are the raw foreground images and on the right are the cleaned-up connected components. You can see that the average difference method leaves behind a sloppier mask and breaks the hand into two components. This is not so surprising; in Figure 9-2, we saw that using the average difference from the mean as a background model often included pixel values associated with the hand value (shown as a dotted line in that figure). Compare this with

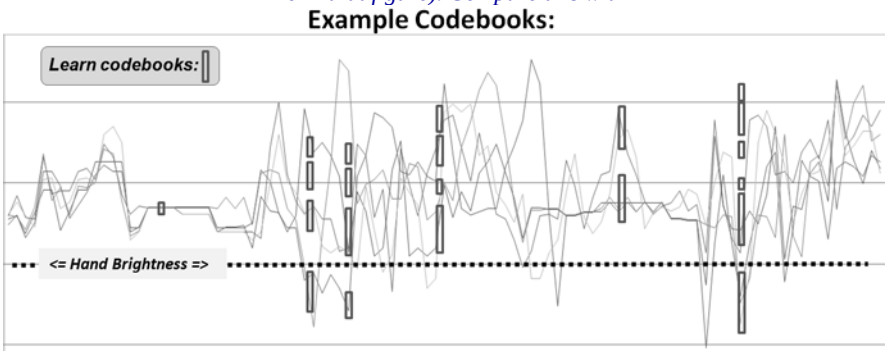


Figure 9-5, where codebooks can more accurately model the fluctuations of the leaves and branches and so more precisely identify foreground hand pixels (dotted line) from background pixels.

Figure 9-8 confirms not only that the background model yields less noise but also that connected components can generate a fairly accurate object outline.

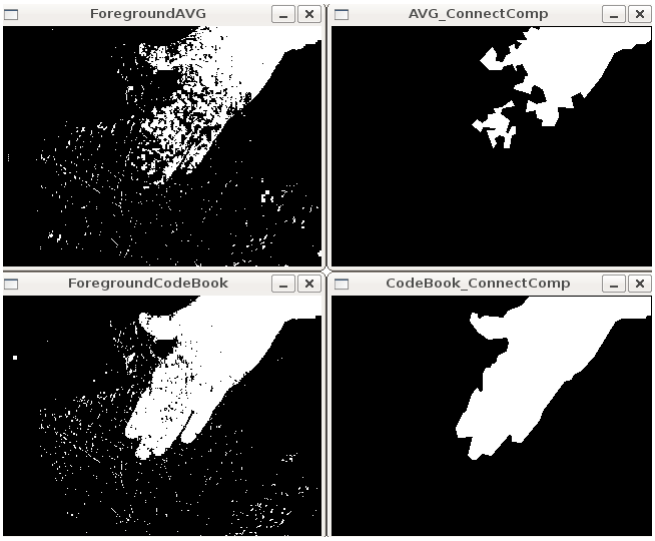


Figure 9-8: With the averaging method (top row), the connected-components cleanup knocks out the fingers (upper-right); the codebook method (bottom row) does much better at segmentation and creates a clean connected-component mask (lower-right)

OpenCV Background Subtraction Encapsulation

Thus far, we have looked in detail at how you might implement your own basic background subtraction algorithms. The advantage of that approach is that it is much more clear what is going on and how everything is working. The disadvantage is that as time progresses, newer and better methods are developed which, though rooted in the same fundamental ideas, become sufficiently complicated so that you would like to be able to regard them as "black boxes" and just use them without getting too deep in the gory details.

To this end, OpenCV now provides a genericized class-based interface to background subtraction. At this time, there are two implementations which use this interface, but as time progresses, there are expected to be more. In this section we will first look at the interface in its generic form, then investigate the two implementations which are available. Both implementations are based on a *mixture of gaussians* (MOG) approach, which essentially takes the statistical modeling concept we introduced for our simplest background modeling scheme (see "Accumulating Means, Variances, and Covariances") and marries it with the multimodal capability of the codebook scheme (the one developed in "A More Advanced Background Subtraction Method"). Both of these MOG methods are 21st century algorithms suitable for many practical day-to-day situations.

The `cv::BackgroundSubtractor` Base Class

The `cv::BackgroundSubtractor` (abstract) base class[15] specifies only the minimal number of necessary methods. It has the following definition:

```
class cv::BackgroundSubtractor : public Algorithm {
```

[15] Actually, as shown below, this base class is not literally abstract (i.e., it does not contain any pure virtual functions). However, it is always used in the library as if it were abstract; meaning that though the compiler will let you instantiate an instance of `cv::BackgroundSubtractor` there is no purpose in, nor meaning to, doing so. We considered coining the phrase "relatively abstract" for the circumstance, but later thought better of it.

```
public:
  virtual ~BackgroundSubtractor();
  virtual void apply(
    cv::InputArray  image,
    cv::OutputArray fgmask,
    double          learningRate = 0
  );
  virtual void getBackgroundImage(
    cv::OutputArray backgroundImage
  ) const;
};
```

As you can see, after the destructor, there are only two methods defined[16]. The first is the `apply()` function, which in this context is used to both ingest a new image and to produce the calculated foreground mask for that image. The second function produces an image representation of the background. This image is primarily for visualization and debugging; after all there is much more information associated with any single pixel in the background than just a color. As a result the image produced by `getBackgroundImage()` can only be a partial presentation of the information that exists in the background model.

One thing that might seem to be a glaring omission is the absence of a method that accumulates background images for training. The reason for this is that there came to be (relative) consensus in the academic literature that any background subtraction algorithm that was not essentially continuously training was an undesirable algorithm. The reasons for this are many, with the most obvious of which being the effect of gradual illumination change on a scene (e.g., as the sun rises and sets outside the window). The more subtle issues arise from the fact that in many practical scenes there is not opportunity to expose the algorithm to a prolonged period in which no foreground objects are present. Similarly, in many cases, things that seem to be background for an extended period (such as a parked car) might finally move, leaving a permanent foreground "hole" at the location of their absence. For these reasons, essentially all modern background subtraction algorithms do not distinguish between training and running modes; rather, they continuously train and build models in which those things that are seen rarely (and can thus be understood to be foreground) are removed and those things that are seen a majority of the time (which are understood to be the background) are retained.

KadewTraKuPong and Bowden Method

The first of the available algorithms brings us several new capabilities which address real-world challenges in background subtraction. These are: a multimodal model, continuous online training, two separate (automatic) training modes that improve initialization performance, and explicit detection and rejection of shadows [KaewTraKulPong2001]. All of this is largely invisible to you, the user. Not unexpectedly, however, this algorithm does have some parameters which you may want to tune to your particular application. They are the *history*, the *number of Gaussian mixtures*, the *background ratio*, and the *noise strength*.[17]

---

[16] You will also note that there is no constructor at all (other than the implied default constructor). We will see that the construction of the subclasses of `cv::BackgroundSubtractor` will be the things we actually want to create, so they provide their own construction scheme.

[17] If you find yourself looking up the citation given for this algorithm, the first three parameters—*history*, *number of Gaussian mixtures*, and *background ratio*—are referred to in the paper as: L, K, and T respectively. The last, and *noise strength*, can be thought of as the initialization value of σ_0 for a newly created component.

The first of these, the *history*, is the point at which the algorithm will switch out of the initialization mode and into its nominal run mode. The default value for this parameter is 200 frames. The *number of Gaussian mixtures* is the number of Gaussian components to the overall mixture model that is used to approximate the background in any given pixel. The default value for this parameter is 5.

Given some number of Gaussian components to the model, each will have a *weight*. This weight indicates the portion of the observed values of a pixel that are explained by that particular component of the model. They are not all necessarily "background," some are likely to be foreground objects which have passed by at one point or another. Ordering the components by weight, the ones which are included as true background are the first b of them, where b is the minimum number required to "explain" some fixed percentage of the total model. This percentage is called the background ratio, and its default value is 0.7 (or 70%). Thus, by way of example, if there are five components, with weights 0.40, 0.25, 0.20, 0.10, and 0.05, then b would be three, because it required the first three $0.40 + 0.25 + 0.20$ to exceed the required background ratio of 0.70.

The last parameter is the *noise strength*. This parameter sets the uncertainty assigned to a new Gaussian component when it is created. New components are created whenever new unexplained pixels appear, either because not all components have been assigned yet, or because a new pixel value has been observed which is not explained by any existing component (in which case the least valuable existing component is recycled to make room for this new information). In practice, the effect of increasing the noise strength is to allow the given number of Gaussian components to "explain" more. Of course, the tradeoff is that they will tend to explain perhaps even more than has been observed. The default value for the noise strength is 15 (measured in units of 0-255 pixel intensities).

`cv::createBackgroundSubtractorMOG()` and `cv::BackgroundSubtractorMOG`

When we would like to construct an algorithm object which actually implements a specific form of background subtraction, we rely on a creator function to generate a `cv::Ptr<>` smart pointer to an instance of the algorithm object. The algorithm object `cv::BackgroundSubtractorMOG` is a subclass of the `cv::BackgroundSubtractor` base class. In the case of `cv::BackgroundSubtractorMOG` that function is `cv::createBackgroundSubtractorMOG()`:

```
cv::Ptr<cv::BackgroundSubtractorMOG> cv::createBackgroundSubtractorMOG(
  int    history         = 200,      // Length of initialization history
  int    nmixtures       = 5,        // Number of Gaussian components in mixture
  double backgroundRatio = 0.7,      // Keep components which explain this fraction
  double noiseSigma      = 0         // Start uncertainty for new components
);
```

Once you have your background subtractor object, you can then proceed to make use of its `apply()` method. The default values used by `cv::createBackgroundSubtractorMOG()` should serve for the majority of cases. The last value is actually the one you are most likely to want to experiment with, the value of noiseSigma, in most cases, should be set to a larger value, `5`, `10`, or even `15`.

Zivkovic Method

This second background subtraction method is in many ways similar to the first, in that it also uses a *Gaussian mixture model* to model the distribution of colors observed in any particular pixel. One particularly notable distinction between the two algorithms is that the Zivkovic method does not use a fixed number of Gaussian components; rather, it adapts the number dynamically to give the best overall explanation of the observed distribution [Zivkovic04, Zivkovic06]. This has the downside that the more components there are, the more compute resources are consumed updating and comparing with the model. On the other hand, it has the upside that the model is capable of potentially much higher fidelity.

This algorithm has some parameters in common with the KB method, but introduces many new parameters as well. Fortunately, only two of the parameters are especially important, while the others are ones which we can mostly leave at their default values. The two particularly critical parameters are the *history* (also called the *decay parameter*) and the *variance threshold*.

The first of these, the *history*, sets the amount of time over which some "experience" of a pixel color will last. Essentially, it is the time it takes for the influence of that pixel to decay away to nothing. The default value for this period is 500 frames. That value is approximately the time before a measurement is "forgotten." Internally to the algorithm, however, it is slightly more accurate to think of this as an exponential *decay parameter* whose value is $\alpha = \frac{1}{500} = 0.002$ (i.e., the influence of a measurement decays like $(1-\alpha)^t$.

The second parameter, the *variance threshold* sets the confidence level with which a new pixel measurement must be within, relative to an existing Gaussian mixture component, to be considered part of that component. The units of the variance threshold are in squared-Mahalanobis distance. This means essentially that if you would include a pixel that is three sigma from the center of a component into that

component, then you would set the variance threshold to $3 \cdot 3 = 9$.[18] The default value for this parameter is actually $4 \cdot 4 = 16$.

`cv::createBackgroundSubtractorMOG2()` and `cv::BackgroundSubtractorMOG2`

Analogous to the previous `cv::BackgroundSubtractorMOG` case, the Zivkovic method is implemented by the object: `cv::BackgroundSubtractorMOG2`, which is another subclass of the `cv::BackgroundSubtractor` base class. As before, these objects are generated by an associated creator function: `cv::createBackgroundSubtractorMOG2()`, which not only allocates the algorithm object, but also returns a smart pointer to the allocated instance.

```
cv::Ptr<cv::BackgroundSubtractorMOG2> cv::createBackgroundSubtractorMOG2(
  int   history          = 500,    // Length of history
  float varThreshold     = 16,     // Threshold decides if new pixel is "explained"
  bool  bShadowDetection = true    // true if MOG2 should try to detect shadows
);
```

The history and variance threshold parameters `history` and `varThreshold` are just as described above. The new parameter `bShadowDetection` allows optional shadow detection and removal to be turned on. When operational, it functions much like the similar functionality in the KB algorithm but, as you would expect, slows the algorithm down slightly.

If you want to modify any of the values you set when you called `cv::createBackgroundSubtractorMOG2()`, there are getter/setter methods which can be used to change not only these values, but a number of the more subtle parameters of the algorithm as well.

```
int    cv::createBackgroundSubtractorMOG2::getHistory();                    // Get
void   cv::createBackgroundSubtractorMOG2::setHistory( int val );           // Set

int    cv::createBackgroundSubtractorMOG2::getNMixtures();                  // Get
void   cv::createBackgroundSubtractorMOG2::setNMixtures( int val );         // Set

double cv::createBackgroundSubtractorMOG2::getBackgroundRatio();            // Get
void   cv::createBackgroundSubtractorMOG2::setBackgroundRatio( double val ); // Set

double cv::createBackgroundSubtractorMOG2::getVarThresholdGen();            // Get
void   cv::createBackgroundSubtractorMOG2::setVarThresholdGen( double val ); // Set

double cv::createBackgroundSubtractorMOG2::getVarInt();                     // Get
void   cv::createBackgroundSubtractorMOG2::setVarInt(double val );          // Set

double cv::createBackgroundSubtractorMOG2::getComplexityReductionThreshold(); // Get
void   cv::createBackgroundSubtractorMOG2::setComplexityReductionThreshold(
                                          double val
                                        );                                  // Set

bool   cv::createBackgroundSubtractorMOG2::getDetectShadows();              // Get
void   cv::createBackgroundSubtractorMOG2::setDetectShadows( bool val );    // Set

double cv::createBackgroundSubtractorMOG2::getShadowThreshold();            // Get
void   cv::createBackgroundSubtractorMOG2::setShadowThreshold( double val ); // Set
```

---

[18] Recall that the Mahalanobis distance is essentially a z-score (i.e., a measurement of how far you are from the center of a Gaussian distribution—measured in units of that distribution's uncertainty) which takes into account the complexities of a distribution in an arbitrary number of dimensions with arbitrary covariance matrix Σ.

$$r_M^2 = (\vec{x} - \vec{\mu})\Sigma^{-1}(\vec{x} - \vec{\mu})$$

You can also see why computing the squared Mahalanobis distance is more natural, which is why you provide the threshold as z^2 rather than z.

```
int    cv::createBackgroundSubtractorMOG2::getShadowValue();                  // Get
void   cv::createBackgroundSubtractorMOG2::setShadowValue( int val );         // Set
```

The meaning of these functions is as follows: `setNMixtures()` resets the length of the history that you assigned with the constructor. `setNMixtures()` sets the maximum number of Gaussian components any pixel model can have (the default is 5). Increasing this improves model fidelity at the cost of runtime. `setBackgroundRatio()` sets the background ratio, which has the same meaning as in the KB algorithm[19] (default for this algorithm is 0.90).

The function `setVarThresholdGen()` controls when a new component of the muli-Gaussian model will be created. If the squared Mahalanobis distance from the center of the nearest component of the model exceeds this threshold for generation, then a new component will be added centered on the new pixel value. The default value for this parameter is $3^2 = 9$. Similarly, the function `setVarInit()` controls the *variance threshold* (described earlier in this section), which is the initial variance assigned to a new Gaussian component of a model. Don't forget that both the threshold for generation and the new model variance are squared distances, so typical values will be 9, 16, 25, etc. (not 3, 4, or 5).

The `setComplexityReductionThreshold()` function controls what Zivkovic et al. call the *complexity reduction prior*. It is related to the number of samples needed to accept that a component actually exists. The default value for this parameter is 0.05. Probably the most important thing to know about this value is that if you set it to 0.00, then the entire algorithm simplifies substantially[20] (both in terms of speed and result quality).

The remaining functions set variables associated with how shadows are handled. The `setDetectShadows()` function simply allows you to turn on and off the shadow detection behavior of the algorithm (effectively overriding whatever value you gave to `bShadowDetection` when you called the algorithm constructor). If shadow detection is turned on, you can set the threshold that is used to determine if a pixel is a shadow using the `setShadowThreshold()` function. The interpretation of the shadow threshold is that it is the relative brightness threshold for a pixel to be considered a shadow relative to something which is already in the model (e.g., if the shadow threshold is 0.60, then any pixel which has the same color as an existing component and is between 0.60 and 1.0 times as bright is considered a

[19] As a good rule of thumb, you can expect that a pixel whose value is not described by the existing model and which stays approximately constant for a number of frames equal to the history times the background ratio, will be updated in the model to become part of the background.

[20] For a more technical definition of "simplifies substantially," what really happens is that Zivkovic's algorithm simplifies into something very similar to the algorithm of Stauffer and Grimson. We do not discuss that algorithm here in detail, but it is cited in Zivkovic's paper and was a relatively standard benchmark relative to which Zivkovic's algorithm was an improvement.

shadow). The default value for this parameter is 0.50. Finally, the `setShadowValue()` function is used

if you want to change the numerical value assigned to shadow pixels in the foreground mask. By default, background will be assigned the value of `0`, foreground the value of `255`, and shadow pixels the value of `127`. You can change the value assigned to shadow pixels using `setShadowValue()` to any value (except `0` or `255`).

Summary

In this chapter, we looked at the specific problem of background subtraction. This problem plays a major role in a vast array of practical computer vision applications, ranging from industrial automation, to security, to robotics. Starting with the basic theory of background subtraction, we developed two basic models of how such subtraction could be accomplished based on simple statistical methods. From there we showed how connected component analysis could be used to increase the utility of background subtraction results and compared the two basic methods we had developed.

We concluded the chapter by looking at the more advanced background subtraction methods supplied by the OpenCV library as complete implementations. These methods are similar in spirit to the simpler methods we developed in detail at the beginning of the chapter, but contain improvements which make them suitable for more challenging real-world applications.

Exercises

1. Using `cv::accumulateWeighted()`, re-implement the averaging method of background subtraction. In order to do so, learn the running average of the pixel values in the scene to find the mean and the running average of the absolute difference (`cv::absdiff()`) as a proxy for the standard deviation of the image.
2. Shadows are often a problem in background subtraction because they can show up as a foreground object. Use the averaging or codebook method of background subtraction to learn the background. Have a person then walk in the foreground. Shadows will "emanate" from the bottom of the foreground object.
   a) Outdoors, shadows are darker and bluer than their surround; use this fact to eliminate them.
   b) Indoors, shadows are darker than their surround; use this fact to eliminate them.
3. The simple background models presented in this chapter are often quite sensitive to their threshold parameters. In Chapter 10, we'll see how to track motion, and this can be used as a reality check on the background model and its thresholds. You can also use it when a known person is doing a "calibration walk" in front of the camera: find the moving object and adjust the parameters until the foreground object corresponds to the motion boundaries. We can also use distinct patterns on a calibration object itself (or on the background) for a reality check and tuning guide when we know that a portion of the background has been occluded.
   a) Modify the code to include an auto-calibration mode. Learn a background model and then put a brightly colored object in the scene. Use color to find the colored object and then use that object to automatically set the thresholds in the background routine so that it segments the object. Note that you can leave this object in the scene for continuous tuning.
   b) Use your revised code to address the shadow-removal problem of exercise 2.
4. Use background segmentation to segment a person with arms held out. Investigate the effects of the different parameters and defaults in the `cv::findContours()` routine. Show your results for different settings of the contour approximation method:

a) `cv::CHAIN_APPROX_NONE`

b) `cv::CHAIN_APPROX_SIMPLE`

c) `cv::CHAIN_APPROX_TC89_L1`

d) `cv::CHAIN_APPROX_TC89_KCOS`

5. Although it might be a little slow, try running background segmentation when the video input is first pre-segmented by using `cv::pyrMeanShiftFiltering()`. That is, the input stream is first mean-shift segmented and then passed for background learning—and later testing for foreground—by the codebook background segmentation routine.

   a) Show the results compared to not running the mean-shift segmentation.

   b) Try systematically varying the maximum pyramid level (`max_level`), spatial radius (`sp`), and color radius (`cr`) of the mean-shift segmentation. Compare those results.

6. Set up a camera in your room or looking out over a scene. Use `cv::BackgroundSubtractorMOG` to "watch" your room or scene over several days.

   a) Detect lights on and off by looking at very instantaneous change in brightness.

   b) Segment out (save instances to a file) of fast changing objects (for example people) from medium changing objects (for example chairs).