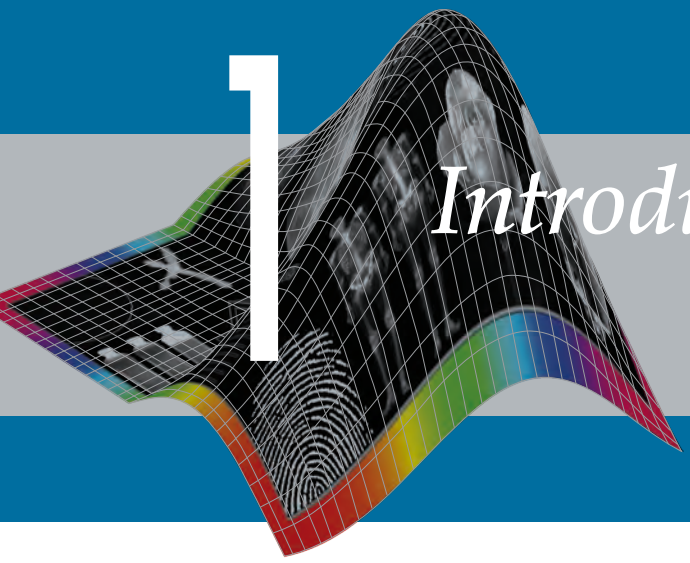


1 Introduction

If you can't explain it simply, you don't understand it well enough.
Albert Einstein

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

1.1 BACKGROUND

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

Relatively little has been written in the way of instructional material to bridge the gap between theory and application in a well-supported software environment for image processing. As with earlier editions, the main objective of this revision is to integrate under one cover a broad base of theoretical concepts with the knowledge required to implement those concepts using state-of-the-art image processing software tools. As before, our focus is on presenting fundamental concepts thoroughly—as simply and clearly as possible.

The theoretical foundation of the material in the following chapters is from the 4th edition of the leading textbook in the field—*Digital Image Processing* by Gonzalez and Woods [2018]. The software code and supporting tools are from the leading software in the field—*MATLAB*® and the *Image Processing Toolbox,*™ from MathWorks. We also take a look at a few selected functions from the MATLAB *Computer Vision, Deep Learning*, *Signal Processing*, and *Wavelet Toolboxes*™.

The material in the book shares the same design, notation, and style of presentation as the Gonzalez-Woods text, thus simplifying cross-referencing between the two.

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

The book is based on principles. It is organized and presented in a textbook format, not as a manual. Thus, basic ideas of both theory and software are explained prior to the development of any new programming concepts. The material is illustrated and clarified further by numerous examples ranging from medicine and industrial inspection to remote sensing and astronomy. This approach allows orderly progression from simple concepts to sophisticated implementation of image processing algorithms. Readers already familiar with MATLAB, the Image Processing Toolbox, and image processing fundamentals, can proceed directly to specific topics of interest, in which case the functions in the book can be used as extensions of the family of Toolbox functions. All new functions developed in the book are fully documented and the code for each is included either in the book or in the *DIPUM3E Support Package* (see Section 1.8). In this edition, we also include for the first time MATLAB projects at the end of every chapter. In total, 130 new projects are part of this edition. Partial project solutions for students and full solutions for instructors are included in the Support Package.

We will use the term *Toolbox* throughout the book to refer specifically to the *Image Processing Toolbox*.

See Section 1.6 for an explanation of blue italic text and other special notation used in the book.

Over 200 *custom functions* are developed in the chapters that follow. These functions extend by approximately 40% the set of about 500 functions in the Image Processing Toolbox. In addition to addressing specific applications, the new functions are good examples of how to combine existing MATLAB and Toolbox functions with new code to develop prototype solutions to a broad spectrum of problems in digital image processing. The custom functions run in all the environments that MATLAB does.

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

1.2 WHAT IS DIGITAL IMAGE PROCESSING, AND WHY IS IT IMPORTANT?

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

Vision is the most advanced of our senses, so it is not surprising that images play the single most important role in human perception. However, unlike humans, who are limited to the visual band of the electromagnetic (EM) spectrum, imaging machines can cover the entire EM spectrum, ranging from gamma to radio waves. They can operate also on images generated by sources that humans do not customarily associate with images. These include ultrasound, electron microscopy, and computer-generated images. Thus, digital image processing encompasses a broad and varied field of applications.

There are two principal factors underlying the current widespread interest in digital image processing. One factor is the unprecedented growth in the worldwide generation of digital data. It has not been too long since we used to measure large amounts of digital data in *terabytes* (one trillion or 10^{12} bytes). Now, we talk in units of *zettabytes* (1 trillion gigabytes or 10^{21} bytes). It has been estimated that 90% of all the data in the world today has been generated in the past two years, reaching a rate of one zettabyte/year at the time of this writing, and expected to more than double every year for the foreseeable future. The second factor, which brings digital image processing into the mix, is an old established estimate that close to 90% of the information received by the human brain is visual. You may disagree with the overall accuracy of these estimates, but the fact is undeniable that data is growing at a rate that is making it harder and harder for humans to process it—and the rate of growth is accelerating. Image processing is playing an increasingly important role in helping us process, understand, and extract value from this ever increasing stream of digital data. In the chapters that follow, you will learn the foundation of the techniques that make this possible.

There is no general agreement among authors regarding where image processing stops and other related areas, such as image analysis and computer vision, begin. Sometimes a distinction is made by defining image processing as a discipline in which both the input and output of a process are images. We believe this to be a limiting and somewhat artificial boundary. For example, under this definition, even the trivial task of computing the average intensity of an image would not be considered an image processing operation. On the other hand, there are fields, such as computer vision, whose ultimate goal is to use computers to emulate human vision, including learning and being able to make inferences and take actions based on visual inputs. This area itself is a branch of *artificial intelligence* (AI), whose objective is to emulate human intelligence. Despite some impressive recent breakthroughs, especially in the field of *deep learning* (the topic of Chapter 14), the field of AI is still in its infancy in terms of practical applications, with progress having been much slower than originally anticipated. The area of *image analysis* (also called *image understanding*) is in between the scopes of image processing and computer vision.

There are no clear-cut boundaries in the continuum from image processing at one end to computer vision at the other. However, a useful paradigm is to consider three types of computerized processes in this continuum: low-, mid-, and high-level processes. *Low-level* processes involve primitive operations, such as image preprocessing to reduce noise, contrast enhancement, and image sharpening. A low-level process is characterized by the fact that both its inputs and outputs typically are

images. *Mid-level* processes on images involve tasks such as segmentation (partitioning an image into regions or objects), description of those objects to reduce them to a form suitable for computer processing, and classification (recognition) of individual objects. A mid-level process is characterized by the fact that its inputs generally are images, but its outputs are attributes extracted from those images, such as edges, regions, and the identity of individual objects. Finally, *high-level* processing involves "making sense" of an ensemble of recognized objects, as in image analysis, and, at the far end of the continuum, performing the cognitive functions normally associated with human vision.

Based on the preceding comments, we see that a logical place of overlap between image processing and image analysis is the area of recognition of individual regions or objects in an image. Thus, what we call in this book *digital image processing* encompasses processes whose inputs and outputs are images and, in addition processes that extract attributes from images, up to and including the recognition of individual objects. As a simple illustration to clarify these concepts, consider the area of automated analysis of text. The processes of acquiring an image of a region containing the text, preprocessing that image, extracting (segmenting) the individual characters, describing the characters in a form suitable for computer processing, and recognizing those individual characters, are in the scope of what we call digital image processing in this book. Making sense of the content of the text as a whole may be viewed as being in the domain of image analysis and even computer vision, depending on the level of complexity implied by the statement "making sense of." Digital image processing, as we have defined it, is used successfully in a broad range of areas of exceptional social and economic value.

1.3 BACKGROUND ON MATLAB, THE IMAGE PROCESSING TOOLBOX, AND OTHER RELATED TOOLBOXES

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

- Math and computation
- Algorithm development
- Data acquisition
- Modeling, simulation, and prototyping
- Data analysis, exploration, and visualization
- Scientific and engineering graphics
- Application development, including app building
- Deployment of algorithms in production systems

MATLAB is an interactive system whose basic data element is a matrix. This allows formulating solutions to many technical computing problems, especially those

involving matrix representations, in a fraction of the time it would take to write a program in a scalar non-interactive language.

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

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

In university environments, MATLAB is the standard computational tool for introductory and advanced courses in mathematics, engineering, and science. In industry, MATLAB is the computational tool of choice for research, development, analysis, and deployment. MATLAB is complemented by a family of application-specific solutions called *toolboxes*. The Image Processing Toolbox is a collection of MATLAB functions that extend the capability of the MATLAB environment for the solution of digital image processing problems. Other toolboxes that sometimes are used in conjunction with the Image Processing Toolbox are the Computer Vision, Signal Processing, Deep Learning, Fuzzy Logic, and Wavelet Toolboxes.

The *MATLAB and Simulink Student Suite* is a low-priced bundle that includes full-featured MATLAB, Simulink, Image Processing Toolbox, and several other add-on products that are most commonly used in engineering and scientific courses. The Computer Vision, Deep Learning, Wavelet, and Fuzzy Logic Toolboxes can be added to the bundle for a small extra cost. The bundle can be purchased directly from the MathWorks web site (**www.mathworks.com**). In addition, many universities and research institutions have campus-wide or site-wide licenses for general use.

1.4 THE MATLAB DESKTOP

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

See Section 1.6 for an explanation of bold black text, blue italics, and other special notation used in the book.

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

The window on the left in Fig. 1.1 shows the contents of the ***Current Folder***, which contains the folders and files with which a user is working at a given time. The path to the **Current Folder** is displayed in the ***Current Folder Field***.

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

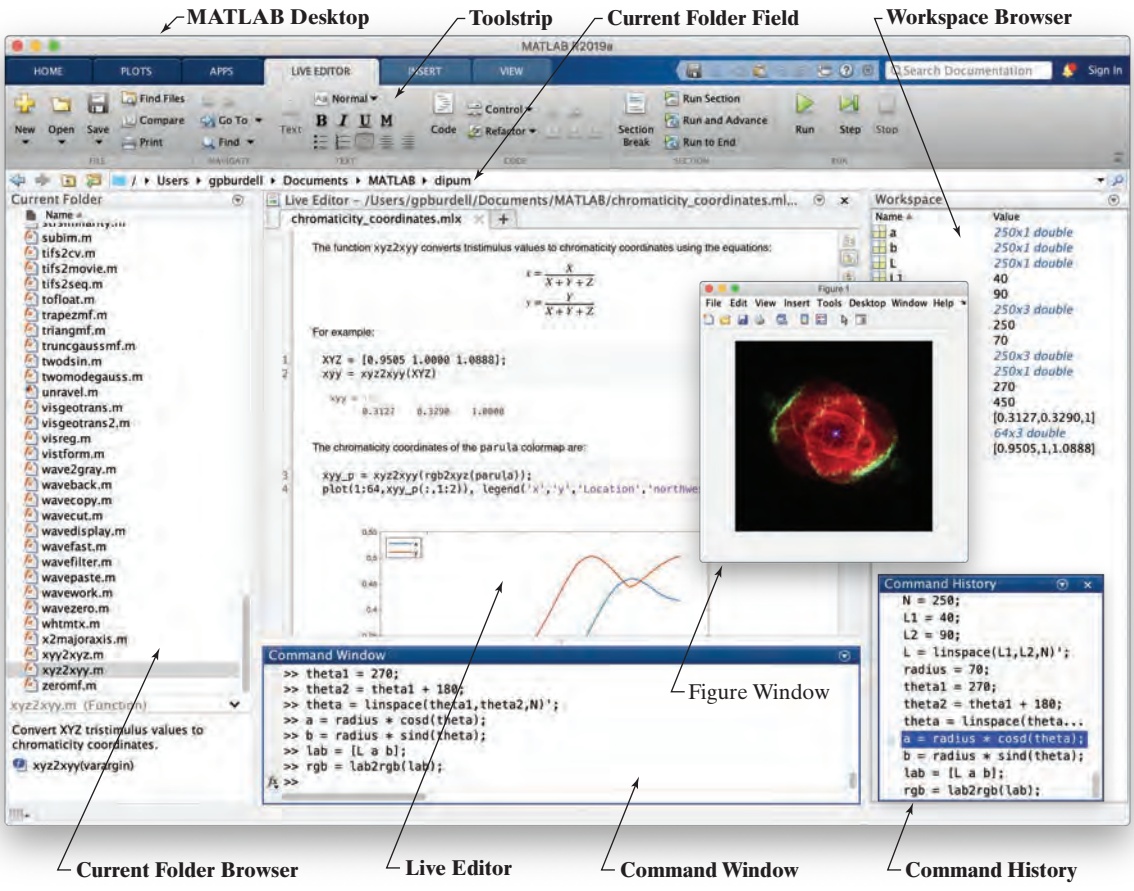


FIGURE 1.1 The MATLAB **Desktop** showing its components in a typical configuration.

During a typical MATLAB session for image processing, images and graphs are displayed to visualize the results of an operation. Each image is displayed in a separate ***Figure Window***. A **Figure Window** has pull-down menus that can be used to edit and save figures in several file formats. **Figure Windows** are not docked and can be moved anywhere in your display screen(s). Most of the other windows shown in Fig. 1.1, can be undocked and similarly moved.

MATLAB uses a ***Search Path*** to find files. Any file run in MATLAB must reside in the **Current Folder** or in a folder that is on the **Search Path**. By default, the files supplied with MATLAB and MathWorks toolboxes are included in the **Search Path**. The easiest way to see which folders and files are on the **Search Path**, or to add or modify a **Search Path**, is to use the **Set Path** icon in the **HOME** tab in the **Toolstrip**. It is good practice to add commonly used folders to the **Search Path** to avoid repeatedly having to browse to the locations of these folders.

TABLE 1.1 MATLAB Desktop tools.

Tool	Description
Array Editor	View and edit array contents.
Command History Window	View a log of statements entered in the **Command Window**; search for previously executed statements, copy them, and re-execute them.
Command Window	Run MATLAB statements.
Current Folder Browser	View and manipulate files in the current folder.
Current Folder Field	Shows the path leading to the current folder.
Editors	Editor/Debugger and Live Editor (explained in the text).
Figure Windows	Display, modify, annotate, and print MATLAB graphics.
File Comparisons	View detailed differences between two files.
Help Browser	View and search product documentation.
Profiler	Measure execution time of MATLAB functions and lines; count how many times code lines are executed.
Start Button	Run product tools and access product documentation.
Workspace Browser	View and modify contents of the workspace.

Table 1.1 shows all the available **Desktop** tools. The MATLAB **Desktop** can be configured to show one, several, or all of these tools. Favorite **Desktop** layouts can be saved for future use.

USING THE EDITOR/DEBUGGER

The MATLAB ***Editor/Debugger*** (or just the ***Editor***) is one of the most important and versatile of the **Desktop** tools. Its primary purpose is to create and edit MATLAB script, function, and class files. The **Editor** highlights different MATLAB code elements in color and analyzes code to offer improvement suggestions. With the **Editor**, a user can also analyze executing code by setting debugging breakpoints, inspecting variables during code execution, and executing code in small, discrete steps. The user can also include code output, graphics output, formatted text, equations, and images in code files, and these files can be exported to PDF, HTML, LaTeX, and Word. To open the **Editor**, type `edit` at the prompt in the **Command Window**, or press **New** or **Open** in the **Toolstrip**. Type `edit filename` at the prompt to open the code file with the specified name. If a code file with that name is not in the **Current Folder** or on the **Search Path**, MATLAB will offer to create it for you in the **Current Folder**.

Throughout the book, code elements are shown in the color they would appear in the MATLAB **Editor** or **Live Editor**. See Section 1.6 for more details on the notation used in the book.

LIVE SCRIPTS AND THE LIVE EDITOR

Live scripts are program files that contain your code, output, and formatted text together in a single interactive environment called the ***Live Editor***. In live scripts, you can write your code and view the generated output and graphics along with the code that produced it. You can add formatted text, images, hyperlinks, and equations

to create an interactive narrative that you can share with others. The editor shown in Fig. 1.1 is an open session in the **Live Editor**.

GETTING HELP

The principal way to get help is to use the MATLAB **Help Browser**, opened as a separate window either by clicking on the question mark symbol (**?**) in the **Toolstrip**, or by typing `doc` (one word) at the prompt in the **Command Window**. The **Help Browser** consists of two panes, the help navigator pane, used to find information, and the display pane, used to view the information. It is good practice to open the **Help Browser** at the beginning of a MATLAB session to have help readily available during code development and other MATLAB tasks.

Another way to obtain help for a specific function is by typing `doc` followed by the function name at the command prompt. For example, typing `doc fileName` displays the reference page for the function called `fileName` in the display pane of the Help Browser. This command opens the browser if it is not open already. The `doc` function works also for user-written code files that contain help text. See Section 2.9 for details on how to include help text in a function.

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

Finally, the MathWorks web site (**www.mathworks.com**) contains a large database of help material, examples, contributed functions, and other resources that should be utilized when the local documentation contains insufficient information about a desired topic. Consult the book website (see Section 1.7) for additional MATLAB support resources.

SAVING AND RETRIEVING A MATLAB SESSION

There are several ways to save or load an entire work session (the contents of the **Workspace Browser**) or selected **Workspace** variables in MATLAB. The simplest is to save the entire **Workspace** by right-clicking on any blank area in the **Workspace Browser** window and selecting **Save Workspace As** from the menu that appears. This opens a directory window that allows you to name the file and select any folder in which to save it. Then click **Save**. To save a selected variable from the **Workspace**, select the variable with a left click and right-click on the highlighted area. Then select **Save Selection As** from the menu that appears. This opens a window from which a folder can be selected to save the variable. To select multiple variables, use shift-click or control-click in the familiar manner, and then use the procedure just described for a single variable. All files are saved in a binary format with the extension `.mat`. These saved files are referred to as *MAT-files*. For example, a session named `mywork_2019_02_10` would appear as

MATLAB code files have a `.m` extension.

the MAT-file `mywork_2019_02_10.mat` when saved. Similarly, a saved image called `final_image` (which is a single variable in the workspace) will appear as `final_image.mat` when saved.

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

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

1.5 AREAS OF IMAGE PROCESSING COVERED IN THE BOOK

This edition is a major revision of the book that brings it up to date in the technical areas covered and in the MATLAB software functionality. Every chapter in the book contains the pertinent MATLAB and Image Processing Toolbox material needed to implement the image processing methods discussed. When a MATLAB or Toolbox function does not exist to implement a specific method, a custom function is developed and documented. The source code of every new function is available, either in the book or in the DIPUM3E Support Package discussed in the next section. The following are short summaries of the material covered in Chapters 2 through 14.

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

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

Chapter 4: Filtering in the Frequency Domain. The material in this chapter shows how to use Toolbox functions for computing the forward and inverse 2-D fast Fourier transform (FFT), how to visualize the Fourier spectrum and filter transfer functions, and how to implement filtering in the frequency domain. Included also is a method for generating frequency domain filters from specified spatial filters.

Chapter 5: Image Restoration and Reconstruction. A suite of linear and nonlinear filters for image denoising are developed and illustrated. We also discuss traditional linear restoration methods, such as the Wiener filter, and extend the discussion to

iterative, nonlinear methods, such as the Richardson-Lucy method and maximum-likelihood estimation for blind deconvolution. We also derive from basic principles the foundational algorithms for image reconstruction from projections and how they are used in computed tomography.

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

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

Chapter 8: Wavelet and Other Image Transforms. We discuss a set of linear transforms that decompose functions into weighted sums of orthogonal basis functions, including the discrete cosine, Walsh-Hadamard, and Haar transforms. In addition, we develop a set of self-contained wavelet functions that are compatible with the MATLAB Wavelet Toolbox and extend its capabilities. We also apply and illustrate how to use these transforms in image processing.

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

Chapter 10: Morphological Image Processing. The extensive set of functions available in the Toolbox for morphological image processing are explained and illustrated in this chapter using both binary and grayscale images.

Chapter 11: Image Segmentation I: Edge Detection, Thresholding, and Region Detection. In this chapter we cover a wide range of image segmentation methods, ranging from edge detection, thresholding, and region growing to Gabor filters, clustering, superpixels, and graph cuts. We also discuss image segmentation using morphological watersheds.

Chapter 12: Image Segmentation II—Active Contours: Snakes and Level Sets. In this chapter we discuss image segmentation using active contours. We develop and illustrate a family of custom segmentation functions for both snakes and level sets.

Chapter 13: Feature Extraction. We develop and illustrate several functions for feature detection and description, including chain-codes, polygonal approximations, Fourier descriptors, texture, moment invariants, and principal components. These functions complement an extensive set of region property functions available in the Image Processing Toolbox. We also discuss whole-image features such as corners,

maximally-stable extremal regions, and keypoint features using methods such as FAST, SURF, and BRISK.

Chapter 14: Classical and Deep Learning Methods for Image Pattern Classification. We conclude our coverage of digital image processing with a discussion of methods for image pattern classification. We cover and develop functions for classical classification methods that include minimum-distance and optimal statistical classifiers. A significant portion of the chapter is devoted to deep learning techniques, which have experienced significant growth since the last edition of the book. The focus of our coverage in this field is first to derive and implement the fundamental equations for fully-connected and convolutional neural networks. The objective is that, when you master this material, you will understand the underpinnings of these important concepts and also be able to program them in MATLAB. We conclude the chapter with a brief illustration of the capabilities of the MATLAB Deep Learning Toolbox for implementing large-scale deep learning systems.

1.6 NOTATION AND ICONS USED IN THE BOOK

Equations in the book are typeset using familiar italic and Greek symbols such as $f(x, y) = A\sin(ux + vy)$ and $\phi(u, v) = \tan^{-1}\left[I(u, v)/R(u, v)\right]$. All MATLAB function names, symbols, and text are typeset in monospace font,; for example, `fft2(f)`, `logical(A)`, and `roipoly(f,c,r)`. Note that monospace text is set in a light shade of gray to help it stand out from normal text.

When a MATLAB function is first defined, we use the following icon:


```
f = matlabFunction(g)
```

Similarly, the first occurrence of an Image Processing Toolbox function is coupled with the following icon;


```
f = toolboxFunction(g)
```

Occasionally, we will work with functions from other MATLAB toolboxes. The first occurrence of those is denoted by the following icon:


```
f = otherToolboxFunction(g)
```

Finally, we use the following icon when a custom function is mentioned for the first time:


```
f = customFunction(g)
```

Occasionally, we will use the same icons more than once when a function is used in different ways. You can find all the important occurrences of any function in the Index. Custom functions are additionally listed in the Appendix.

All words that are included in the Index—or are related to Index entries—are shown in blue italics in the text to make them easier to find. For example: "... when x, y, and the amplitude values of f are all finite, discrete quantities, we call the image a *digital image*." When emphasizing words not relevant to the Index, we use normal italics. For example: "... where x_{max} denotes the *maximum* coordinate value."

We use bold letters when referring to keyboard keys such as **Return** and **Tab**, and also when referring to items on a computer screen or menu, such as **File** and **Edit**. We use the acronym *DIPUM3E*, meaning *Digital Image Processing, 3rd ed.*, throughout the book and in the book website.

We display all MATLAB code words using the same colors that you will see in the MATLAB editors. For example, `for` and `end` are shown in blue, strings such as `'fileNames'` are shown in purple, and comments are shown in green, preceded by the symbol %, as in `% This is a comment`. All remaining code word themselves are shown in gray in the book to help differentiate them from normal text, as in the line of code `f = zeros(10)`. All MATLAB code words are displayed using the same standard monospace font.

1.7 THE BOOK WEBSITE

An important feature of the book is the support contained in the book website, whose address is

www.ImageProcessingPlace.com

The site provides support for the book in the following areas:

- MATLAB source code.
- Tutorials on a wide range of topics relevant to the material in the book.
- Teaching materials.
- Image databases.
- Errata sheets.
- Publications in the field of image processing and related areas.

The same site supports all recent editions of the Gonzalez-Woods book and thus provides complementary materials. The site also hosts the support package explained in the next section.

1.8 THE DIPUM3E SUPPORT PACKAGE

For this edition of the book, we created the *DIPUM3E Support Package* for students and faculty. The Package comes in two versions that contain the support materials available for the new edition, organized into one easy download.

The *DIPUM3E Student Support Package* contains:

- Access to the source code for all the functions developed in the book.
- All the original images in the book.

- Detailed answers to selected projects at the end of every chapter.

The *DIPUM3E Faculty Support Package contains*:

- Access to the source code for all the functions developed in the book.
- All the original images in the book.
- Detailed answers to all projects in the book.
- PowerPoint slides that contain all the art in the book and are ideally suited for building classroom presentations.

The names of all custom functions in the Package are listed in the Index and in the Appendix.

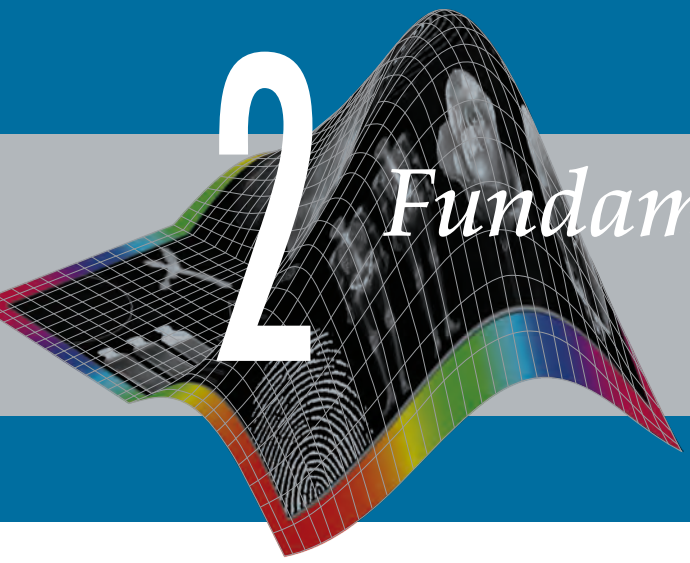
One Support Package comes free of charge with every new book. Applications for the Package for students and faculty are submitted in the book web site.

1.9 HOW REFERENCES ARE ORGANIZED IN THE BOOK

All references in the book are listed in the Bibliography by author and date as, for example, Sejnowski [2018]. Most of the background references for the theoretical content of the book are from Gonzalez and Woods [2018]. In cases where this is not true, the appropriate new references are identified at the point in the discussion where they are needed. Reference details are included in the Bibliography.

Summary

In addition to a brief introduction to notation and basic MATLAB tools, the material in this chapter emphasizes the importance of a comprehensive prototyping environment in the solution of digital image processing problems. In the following chapter, we begin to lay the foundation needed to understand Image Processing Toolbox functions and introduce a set of fundamental programming concepts that are used throughout the book. The material in Chapters 3 through 14 spans a wide cross section of topics that are in the mainstream of digital image processing. Although the topics covered are varied, the discussion of those topics follows the same basic theme of demonstrating how combining MATLAB and Toolbox functions with new code can be used in a wide range of digital image processing applications.



2 Fundamentals

By failing to prepare you are preparing to fail.
Benjamin Franklin

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

Functions Developed in this Chapter:

- `imblend` this is the first custom function presented in the book. It is a simple function designed to illustrate the principal components of a MATLAB function.
- `average` is a function that illustrates conditional branching and the use of dot notation.
- `subim` is a function that extracts a subimage from a larger image. This function is designed to illustrate the use of `for` loops and the importance of memory allocation.
- `sinfunX`—a series of three functions for `X = 1`, `2`, and `3` designed to illustrate function timing, function handles, memory preallocation, `for` loops, and code vectorization.
- `twodsinX`—also a series of three functions for `X = 1`, `2`, and `3` designed to illustrate image generation using `for` loops and code vectorization using the function `meshgrid`.
- `imageStatsX`—a series of five functions for `X = 1`, `2`, `3`, `4`, and `5` used to illustrate programming with cell arrays and structures.
- `interactive` is a function that illustrates interactive I/O using a keyboard and mouse for data entry.

2.1 DIGITAL IMAGE REPRESENTATION

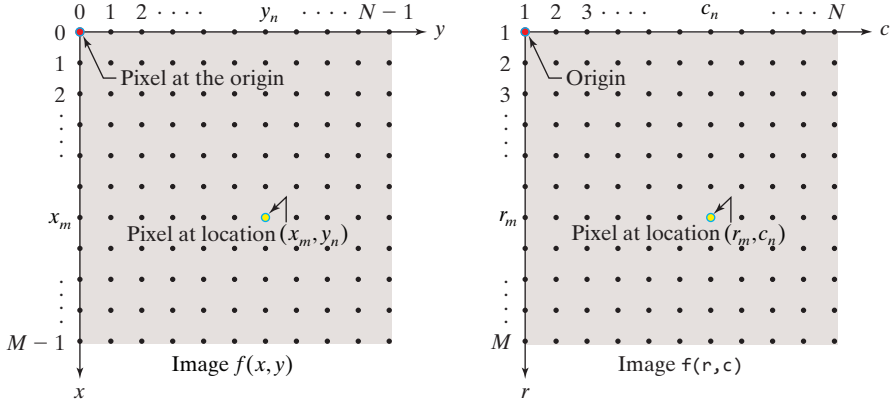
An *image* may be defined as a two-dimensional function, $f(x, y)$, where x and y are spatial (plane) coordinates and the amplitude of f at any pair of coordinates, (x, y), is called the *intensity* of the image at that point. The term *gray level* is used often to refer to the intensity of *monochrome*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

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

COORDINATE CONVENTIONS

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

The coordinate convention used in the Image Processing Toolbox is different from the preceding paragraph in two ways. First, instead of using (x, y), the toolbox



a b

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

uses the notation (r, c) to indicate *rows* and *columns*. The order of the coordinates is the same as above, in the sense that the first element of a coordinate tuple, (a, b), refers to a row and the second to a column. The other difference is that the origin of the coordinate system is at $(r, c) = (1, 1)$; thus, r ranges from 1 to M and c from 1 to N, in integer increments. Figure 2.1(b) illustrates this coordinate convention.

The Image Processing Toolbox documentation refers to the coordinates in Fig. 2.1(b) as *pixel indices* or (less frequently) *intrinsic coordinates*. In some cases, MATLAB and the Toolbox employ another coordinate convention, called *spatial coordinates*, that uses x to refer to columns and y to refers to rows. This is the opposite of our use of variables x and y and can be a source of confusion because you will encounter it in Toolbox and MATLAB documentation. Be aware that MATLAB itself is not always consistent in the meaning it assigns to a tuple (a, b). For instance, when referring to matrices (and by implication images), MATLAB assumes that the first element refers to rows and the second to columns. In other contexts (e.g., in plotting functions), the reverse is true. We will generally mention it whenever this reversal occurs in our discussions.

IMAGES AS MATRICES

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

$$f(x, y) = \begin{bmatrix} f(0,0) & f(0,1) & \cdots & f(0,N-1) \\ f(1,0) & f(1,1) & \cdots & f(1,N-1) \\ \vdots & \vdots & & \vdots \\ f(M-1,0) & f(M-1,1) & \cdots & f(M-1,N-1) \end{bmatrix} \tag{2-1}$$

The right side of this equation is a digital image by definition and each of its elements is a *pixel*.

A digital image can be represented as a MATLAB matrix:

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

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

where `f(1,1)` = $f(0,0)$ (note the use of a monospace font to denote MATLAB quantities) and `f` is any valid MATLAB variable name[†]. Clearly, the two representations are equivalent, except for the notational shift in the origin. Typically, we use the letters M and N, respectively, to denote the number of rows and columns

`namelengthmax`

[†] A *valid* MATLAB *variable name* starts with a letter and is followed by letters, digits, or underscores (no spaces are allowed). MATLAB is case sensitive. The maximum *length* of a variable name is the value returned when you type `namelengthmax` in your version of MATLAB.

in a matrix. A $1 \times N$ matrix is a *row vector*, an $M \times 1$ matrix is a *column vector*, and a 1×1 matrix is a *scalar*.

Matrices in MATLAB are stored in variables with names such as `A`, `a`, `RGB`, `realArray`, and so on. As noted earlier, all MATLAB quantities in this book are written using monospace characters. We use conventional Roman, italic notation, such as $f(x, y)$, for mathematical expressions.

2.2 READING IMAGES

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

`imread`

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

Recall from Section 1.6 that we use margin icons to highlight the first use of a MATLAB, Toolbox, or Custom function. Sometimes, we also use icons to denote other important MATLAB elements such as ";" and ">>".

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

`semicolon(;)`

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

reads the image from the file `'chestXray.jpg'` into image array `f` in the MATLAB **Workspace**. Single quotes (`'`) delimit the string filename. A semicolon at the end of a statement is used to suppress output. If a semicolon is not included, MATLAB displays on the screen the results of the operation(s) specified in that line. The prompt symbol (`>>`) designates the beginning of a command line.

`prompt(>>)`

When no path information is included in `'filename'`, `imread` reads the file from the **Current Folder** and, if that fails, it tries to find the file in the MATLAB Search Path (see Section 1.7). To read an image from a specified folder, include a full or relative path to that folder of `'filename'`. For example,

MathWorks recently adopted standard Windows and Mac terminology by using the term "folder" instead of "directory," but you will still encounter the latter in some MATLAB documentation.

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

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

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

reads the image from the `myimages` subfolder contained in the **Current Folder**. Table 2.1 lists some of the image/graphics formats supported by `imread` and `imwrite` (we discuss the latter function in Section 2.4).

The MATLAB **Desktop** displays the path to the **Current Folder** on the toolbar, which provides an easy way to change it.

Typing `size(f)` at the prompt gives the row and column dimensions of `f`:

`size`

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

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

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

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

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

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

returns the size of `A` as if it were a `K`-dimensional array. The sizes of any dimensions larger than `K` are folded into `DK`. This function is particularly useful in programming to determine automatically the size of an image:

```
>> [M,N] = size(f); % For grayscale images, dim = 2;
>> [M,N,K] = size(g); % For RGB images, dim = 3;
```

Typing,

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

gives the size of `f` along its first (vertical) dimension; that is, the number of rows of `f`. The second dimension is in the horizontal direction, so the statement `size(f,2)` gives the number of columns in `f`. For an RGB image, `K = size(f,3)` gives 3 because an RGB image consists of three grayscale images stacked in the third dimension. A *singleton dimension* is any dimension, `dim`, for which `size(A,dim) = 1`.

The `whos` function displays additional information about an array. For instance, if `f` is the `'chestXray.jpg'` image, the statement

```
>> whos f
```

gives

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

whos

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

TABLE 2.2
Some additional MATLAB functions that are used routinely when working in the MATLAB **Desktop**.

Function name	Symtax	Explanation
`which`	`which item`	Locates functions and files.
`lookfor`	`lookfor keyword`	Searches for the specified `keyword` in the first comment line (the H1 line) of the help text in all MATLAB program files found on the search path.

The **Workspace Browser** in the MATLAB **Desktop** displays similar information. The `uint8` entry shown above refers to one of several MATLAB data classes discussed in Section 2.4. A semicolon at the end of a `whos` line has no effect, so normally one is not used. Table 2.2 lists two other MATLAB functions that you will find quite useful. It is time well spent getting to know these functions.

2.3 DISPLAYING IMAGES

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

`imshow`

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

```
imshow(f)
```

where `f` is an image. Using the syntax

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

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

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

sets variable `low` to the minimum value of `f` and `high` to its maximum value. This form of `imshow` is useful for displaying images that have a low dynamic range or that have positive and negative values. These last two syntax forms do not work for RGB images.

EXAMPLE 2.1: Reading and displaying images.

The following statements read from the **Current Folder** an image called `'rose512.tif'`, extract information about the image, and display it using `imshow`:

```
>> f = imread('rose512.tif');
>> whos f
 Name        Size              Bytes   Class    Attributes
  f        512x512             262144  uint8
```

FIGURE 2.2
Screen capture showing how an image appears on the MATLAB **Desktop**. Note the figure number on the top, left of the window. In most of the examples throughout the book, we show images without the display window.

```
>> imshow(f)
```

Figure 2.2 shows what the output looks like on the screen. The figure number appears on the top, left of the window. The various pull-down menus and utility buttons are used for tasks such as scaling, saving, and exporting the contents of the display window. The **Edit** menu has functions for editing and formatting the contents before they are printed or saved to disk.

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

figure

When used without an argument, as shown here, function `figure` creates a new figure window. Typing `figure(n)` (or clicking on its window) forces figure number `n` to become visible.

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

Using the statement

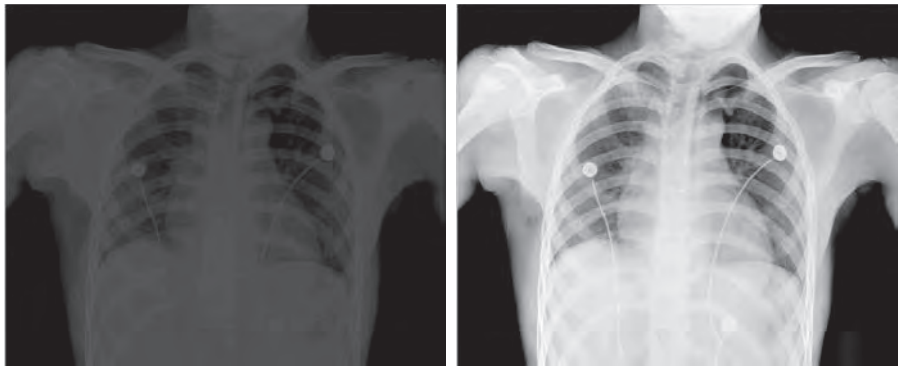
```
>> imshow(f), figure, imshow(g)
```

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

Finally, suppose that we are working with an image, `h`, and find that using `imshow(h)` displays the image in Fig. 2.3(a). This image has a low dynamic range, which can usually be remedied for display purposes using the auto range syntax in `imshow`:

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

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



a b

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

Although `imshow` is the image display function used most frequently, the Toolbox provides an *Image Viewer app* that contains a number of utilities (called *tools*) useful for interactive image exploration, as detailed in Table 2.3. To start the Image Viewer, we type `imtool` at the prompt:

`imtool`

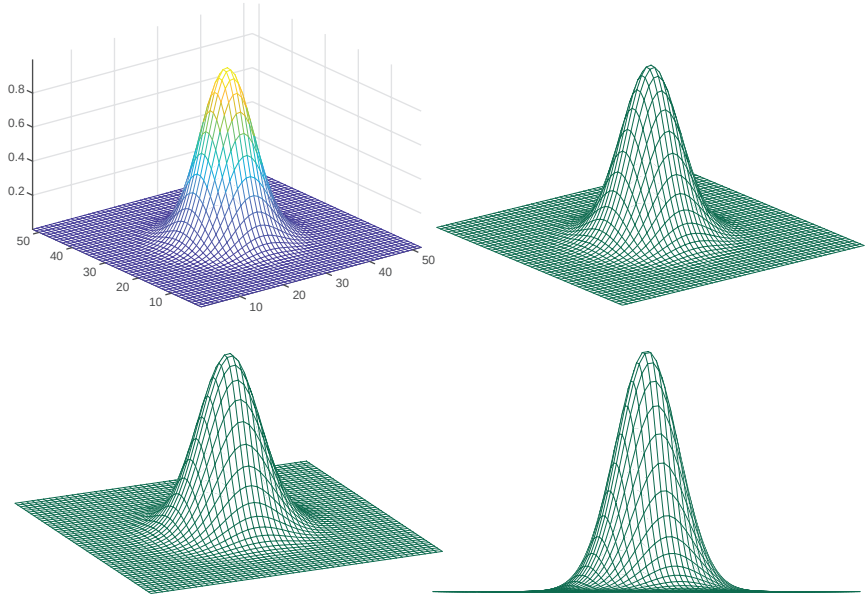
An *app* in MATLAB is an interactive application written to perform computing tasks. In addition to the Image Viewer app, the Toolbox has several other apps, some of which we discuss later in the book.

```
>> f = imread('lunar-shadows.tif');
>> imtool(f)
```

Figure 2.4 shows some of the windows available when using the Image Viewer. The large, central window (titled **Image Tool**) is the main view that opens when you type `imtool(f)` at the prompt. The main view is showing the image at 200% magnification. The status text at the bottom, left of the main window shows the column/row location (244, 375) and value (101) of the pixel lying under the mouse cursor (the origin of the image is at the top, left). The straight line between the two large black regions in the main window is the result of selecting Measure Distance from the **Tools** pull-down menu; it shows a distance of 242.32 pixels. If the physical scale of the image is known, the actual distance can be obtained by multiplying

TABLE 2.3
Functions comprising the Image Viewer app. Also, the tools listed can be run as independent functions.

Function	Description
`imtool`	Starts the **Image Viewer app**.
`imageinfo`	**Image Information tool**.
`imcontrast`	**Adjust Contrast tool**.
`imdisplayrange`	**Display Range tool**.
`imdistline`	**Distance tool**.
`impixelinfo`	**Pixel Information tool**.
`impixelinfoval`	**Pixel Information tool** without text label.
`impixelregion`	**Pixel Region tool**.
`immagbox`	**Magnification box** for scroll panel.
`imoverview`	**Overview tool** for image displayed in scroll panel.



a b
c d

FIGURE 2.14
(a) Plot obtained using function `mesh` and displayed in its default colors. (b) Axes and grid removed and colors changed to a single color. (c) A different view. (d) Another view.

EXAMPLE 2.17: Wireframe plotting.

Using the function `lpfilter` from Chapter 4, we generate a matrix H of size 500×500 elements that are values of a 2-D Gaussian (ignore the details of `lpfilter` for now—we are interested only in H in this example). Figure 2.14(a) shows the result of the commands

```
>> H = fftshift(lpfilter('Gaussian',500,500,50));
>> wf = mesh(H(1:10:500,1:10:500)); % Plot every tenth point.
>> wf.EdgeColor = [0 106 78]/255;
>> axis tight % Fig. 2.14(a).
>> axis off
>> grid off % Fig. 2.14(b).
>> view(-25,30) % Fig. 2.14(c).
>> view(-25,0) % Fig. 2.14(d).
```

We use 3-D plotting is various parts of the book, especially in Chapter 4.

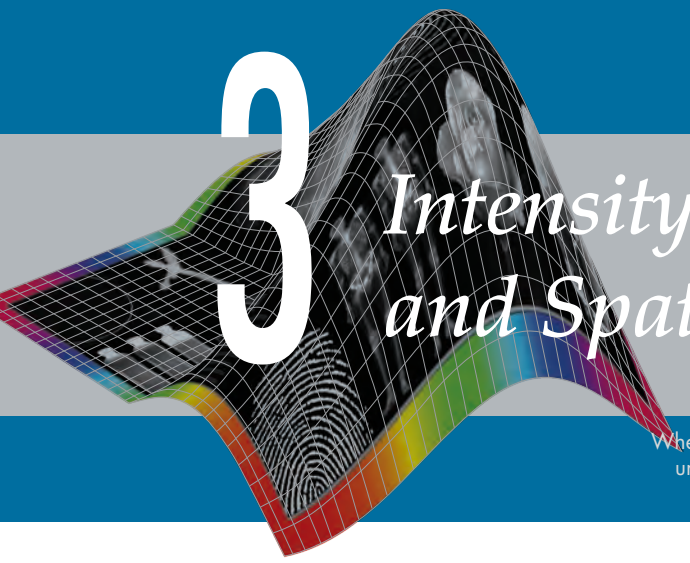
Surface Plots

Sometimes it is desirable to plot a function as a surface instead of as a wireframe. Function `surf` does this. Its basic syntax is

`surf`

```
surf(H)
```

This function generates plots that are identical to `mesh`, except that the quadrilaterals in the mesh are filled with colors by default (this is called *facet shading*). To convert the colors to gray we use the command



3 Intensity Transformations and Spatial Filtering

When you look at the world with knowledge, you realize that things are unchangeable and at the same time are constantly being transformed.
Yukio Mishima

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

Functions Developed in this Chapter:

- `intensityTransformations` performs gray-scale intensity transformations.
- `intensityScaling` scales the intensities of an image to the full [0,1] range.
- `fun2hist` converts a discrete function into a histogram.
- `manualhist` is used for interactive histogram specification.
- `triangmf` implements one of eight membership functions used for fuzzy image processing.
- `lambdafcns` outputs functions for computing rule strength.
- `implfcns` computes implication functions.
- `aggfcn` computes an aggregation function.
- `defuzzify` is a defuzzification function.
- `fuzzysysfcn` implements a complete fuzzy system.
- `approxfcn` approximates the output of function `fuzzysysfcn`.
- `fuzzyfilt` performs fuzzy edge detection.
- `makefuzzyedgesys` is a script that supports function `fuzzyfilt`.

3.1 BACKGROUND

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

$$g(x, y) = T\left[f(x, y)\right] \tag{3-1}$$

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

The principal approach for defining spatial neighborhoods about a point (x_0, y_0) is to use a square or rectangular region centered at (x_0, y_0), as Fig. 3.1 shows. The region is moved from pixel to pixel starting, say, at the top, left corner and, as it moves, it encompasses different neighborhoods. Operator T is applied at each location (x, y) to yield the output, g, at that location. Only the pixels in the neighborhood centered at (x, y) are used in computing the value of g at (x, y).

3.2 INTENSITY TRANSFORMATION FUNCTIONS

When working with grayscale (monochrome) images, intensity transformation functions are often referred to as *gray-level transformation functions*.

The simplest form of the transformation T is when the neighborhood in Fig. 3.1 is of size 1×1 (a single pixel). In this case, the value of g at (x, y) depends only on the intensity of f at that point and T becomes an *intensity transformation function*.

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

$$s = T(r) \tag{3-2}$$

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

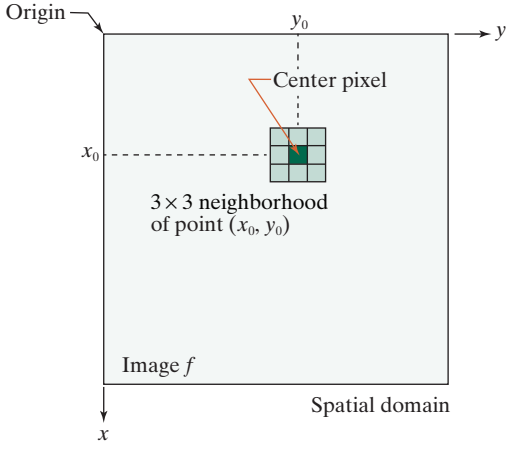
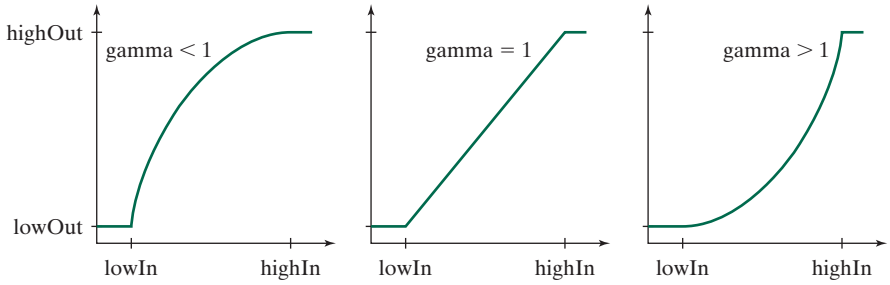


FIGURE 3.1 A neighborhood of size 3×3 centered at a point (x_0, y_0) in a digital image.

a b c

FIGURE 3.2 The mappings available in function `imadjust`.



FUNCTIONS `imadjust` AND `stretchlim`

Function `imadjust` is the basic Toolbox function for intensity transformations. It has the general syntax

`imadjust`

```
g = imadjust(f,[lowIn highIn],[lowOut highOut],gamma)
```

As Fig. 3.2 illustrates, this function maps the intensity values in image `f` to new values in `g`, such that values between `lowIn` and `highIn` map to values between `lowOut` and `highOut`. Values below `lowIn` and above `highIn` are *clipped*; that is, all values below `lowIn` map to `lowOut` and those above `highIn` map to `highOut`. The output image has the same class as the input. All inputs to function `imadjust`, other than `f` and `gamma`, are specified as values between 0 and 1, independently of the class of `f`. If, for example, `f` is of class `uint8`, `imadjust` multiplies its values by 255 to determine the actual values to use. Using the empty matrix (`[]`) for `[lowIn highIn]` or for `[lowOut highOut]` results in the default values `[0 1]`. If `highOut` is less than `lowOut`, the output intensities are reversed.

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

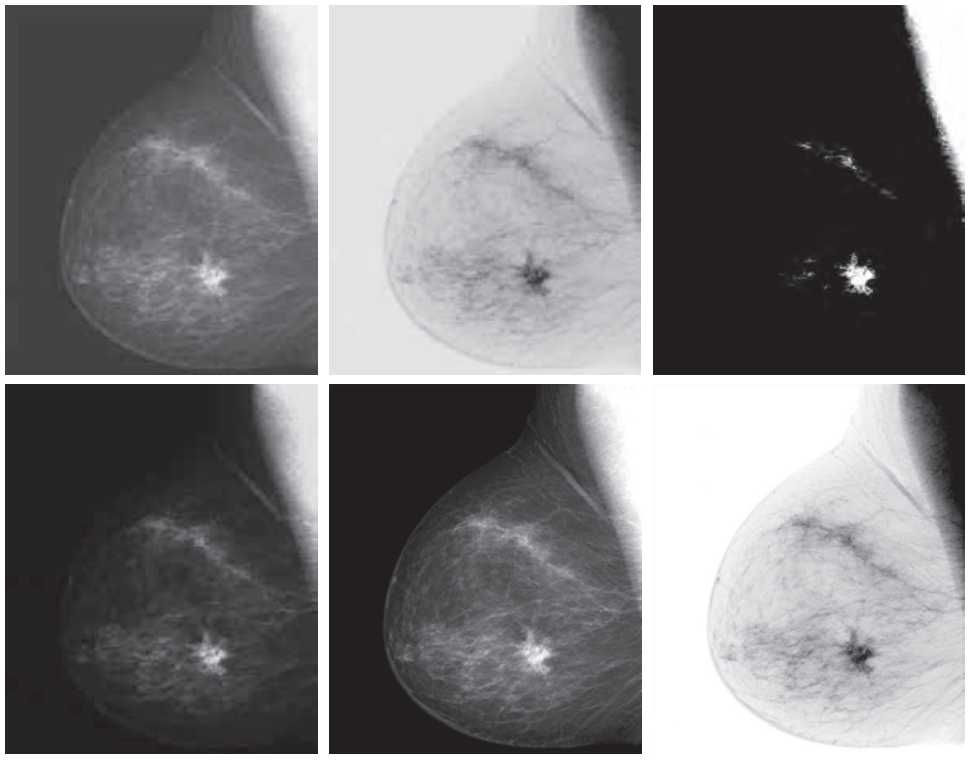
Parameter `gamma` specifies the shape of the transformation curve that maps the intensity values from `f` to `g`. If `gamma` is less than 1, the mapping is weighted toward higher (brighter) output values, as shown Fig. 3.2(a). If `gamma` is greater than 1, the mapping is weighted toward lower (darker) output values, as Fig. 3.2(c) shows. If it is not specified, `gamma` defaults to 1, which is the linear mapping in Fig. 3.2(b).

EXAMPLE 3.1: Using function `imadjust`.

Figure 3.3(a) is a digital mammogram image, `f`, showing a small lesion. We obtained Fig. 3.3(b) using the commands

```
>> f = imread('breastXray.tif');
>> g1 = imadjust(f,[0 1],[1 0]);
>> figure, imshow(g1) % Fig. 3.3(b).
```

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



a b c
d e f

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

`imcomplement`

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

We obtained Fig. 3.3(c) using the command

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

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

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

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

Sometimes, it is of interest to be able to use function `imadjust` "automatically," without having to deal with the low and high parameters discussed above. Function `stretchlim` can be used for this purpose; its basic syntax is

`stretchlim`

```
LowHigh = stretchlim(f)
```

where `LowHigh` is a two-element vector of a lower and upper limit that can be used to achieve contrast stretching (see the following section for a definition of this term).

Function narginchk can be used in the body of a function to check if the correct number of arguments was passed. The syntax is

narginchk

```
msg = narginchk(minargs,maxargs)
```

This function returns the message Not enough input arguments if the number of input arguments is less than minargs or Too many input arguments if the number is greater than maxargs. Program execution stops in either of these two cases. If the number is between minargs and maxargs (inclusive), narginchk does nothing.

It is useful to be able to write functions in which the number of input and/or output arguments is variable. For this, we use varargin and varargout. For example,

varargin

varargout

Both varargin and varargout are cell arrays.

```
function [varagout] = myFunction(varargin)
```

accepts a variable number of inputs into myFunction and returns a variable number of outputs. This type of formulation is useful when the number of inputs depends on parameters chosen by the user and in which the number of outputs depends on the inputs provided for a given set of choices. For example, a function that computes various features of a region would require different inputs for different features and the outputs would depend on which features were specified. The following custom function illustrates the advantages of being able to provide a variable number of inputs. Variable outputs are treated in a similar manner.

A Custom Function for Intensity Transformations

In this section we write a function that computes the following transformation functions: negative, log, gamma, and contrast stretching. The function can also perform user-specified transformations. These transformations were selected because we will need them later and also to illustrate several of the concepts discussed thus far in the book. In writing this function we used function tofloat, introduced in Section 2.8.

Observe our use of several *local functions* in the body of the custom function intensityTransformations. Note also how the various input options are formatted in the Help section of the code, how a variable number of inputs is handled, how error checking is interleaved in the code, and how the class of the output image is matched to the class of the input. Keep in mind when studying this code that varargin is a cell array, so its elements are selected by using *curly braces*.


```
function g = intensityTransformations(f,method,varargin)
%intensityTransformations Grayscale image intensity transformations.
%   G = intensityTransformations(F,'neg') computes the negative of the
%   input image F.
%
%   G = intensityTransformations(F,'log',C,CLASS) computes C*log(1 + F)
%   and multiplies the result by (positive) constant C. If the last
%   parameters is omitted, C defaults to 1. Because the log is used
%   frequently to display Fourier spectra, parameter CLASS offers the
%   option to specify the class of the output as 'uint8' or 'uint16'. If
%   parameter CLASS is omitted, the output is of the same class as the
%   input.
```

EXAMPLE 3.4: Computing and plotting image histograms.

We discussed in Section 2.10 the four principal ways to plot histograms in MATLAB:

- Default plot by typing `imhist` at the prompt.
- Bar plot of `h` using function `bar`.
- Stem plot of `h` using function `stem`.
- Plot of `h` using function `plot`.

Figures 3.7(b)–(f) illustrate these four manners of plotting a histogram for the image in Fig. 3.7(a). Figure 3.7(c) is a copy of the code we used to generate Fig. 3.7(d). You will recognize the form of these commands as those we used to generate Fig. 2.12(b), but using a different color scheme. The details of how we generated the other plots in Fig. 3.7 are explained in the discussion of Fig. 2.12.

HISTOGRAM EQUALIZATION

Histogram equalization is one of the most effective intensity transformations for image enhancement. In histogram equalization, we think of intensities and their histograms as *discrete random variables* and their *discrete probability distributions*, respectively. The *histogram equalization transformation* of a specific random intensity value r_k into a corresponding random intensity value s_k is defined as a scaling constant, K, times the *cumulative distribution function* (CDF) of the random variable r, evaluated at r_k:

$$s_k = K \cdot CDF(r_k) \tag{3-8}$$

We know from basic probability that

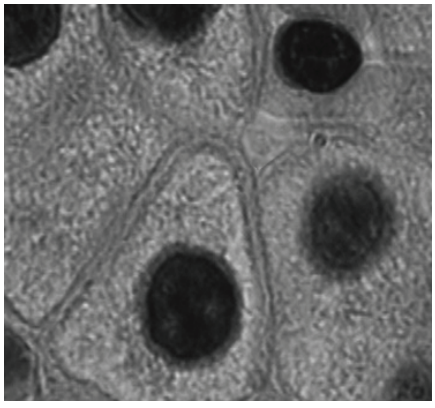
$$CDF(r_k) = \sum_{j=0}^{k} p(r_j) \tag{3-9}$$

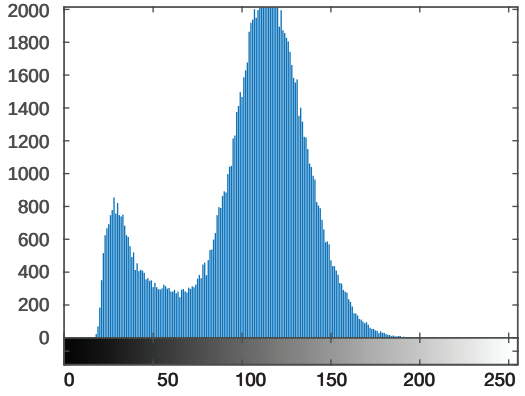
Therefore, with reference to Eq. (3-2), we have that histogram equalization is implemented using the *transformation function*

$$s_k = T(r_k) = K\sum_{j=0}^{k} p(r_j) \tag{3-10}$$

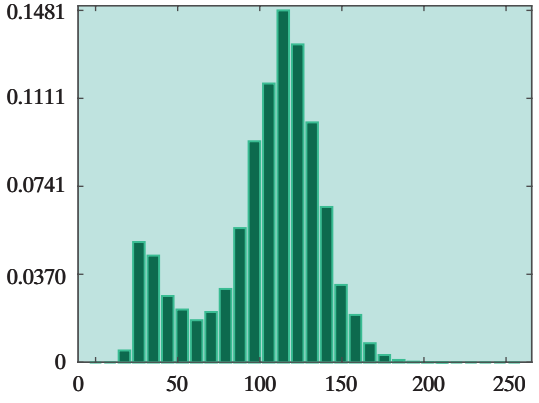
Constant K scales the output to the intensity scale of the input. When r is floating point in the range $r \in [0,1]$, we set $K = 1$. When r represents integer-valued intensities, its values are in the range $r \in [0, L-1]$ and we set $K = L-1$, where L is the number of intensity levels (e.g., 256 for an `uint8` image).

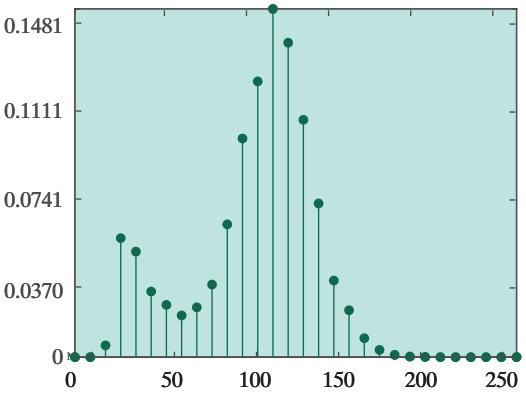
When s and r are *continuous random variables*, it is not difficult to show that using integration instead of a sum in Eq. (3-10) results in an output intensity distribution that is perfectly uniform, *regardless* of the intensity distribution of the input [see Gonzalez and Woods [2018]). A uniform distribution is flat over the full intensity scale. This means that all intensities in the output image would be equally likely to occur. This implies in turn that the output intensities would have a much wider dynamic range. In an image, this translates into increased contrast and usually an

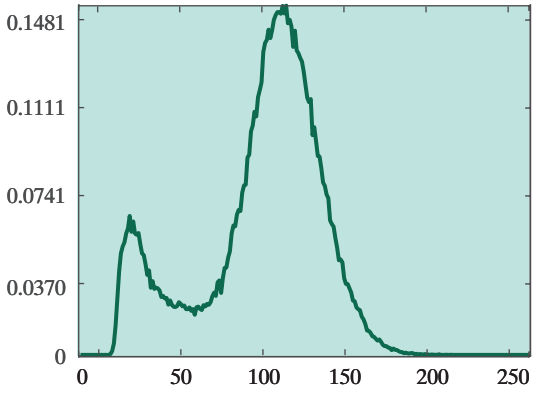





```
f = imread('liver-cells-gray.tif');
h = imhist(f,30)/numel(f);
horz = linspace(0,255,30);
figure, bar(horz,h,...
      'FaceColor',[0 106 78]/255,...
      'EdgeColor', [0 212 156]/255,...
      'LineWidth',0.75)
ax = gca;
ax.Color = [190 228 223]/255;
ax.YTick = 0:max(h(:))/4:max(h(:));
ax.FontName = 'Times Ten';
ax.FontSize = 8;
```







a b
c d
e f

FIGURE 3.7 (a) A transmission electron microscope image of liver cells. (b) Default histogram obtained by typing `imhist(f)` at the prompt. (c) Code used to generate the bar graph on the right. (d) Bar graph obtained using the code on the left. (e) Stem plot. (f) Result of plotting `h` directly using function `plot`. (Image courtesy of NIH.)

enhancement of details hidden in dark regions of the image. Because this result is independent of the intensity distribution of the input, histogram equalization is an automatic enhancement tool. When the intensities are discrete, we can no longer guarantee that the output histogram will be uniform. However, as the next example shows, even an approximation can yield dramatic improvements in the appearance of an image. The fact that histogram equalization depends only on the input histogram, which is simple to compute, makes equalization a widely-used tool in digital image processing.

The Toolbox function that implements histogram equalization is `histeq`:

`histeq`

```
g = histeq(f,n)
```

where `f` is the input image and `n` is the number of intensity levels specified for the output image. If `n` is equal to `L`, then `histeq` implements histogram equalization directly. If `n` is less than `L`, then `histeq` attempts to distribute the levels so that they will approximate a flat histogram. Unlike function `imhist`, the default value in `histeq` is `n = 64`. For the most part, we use the maximum possible number of levels for `n` because this produces a true implementation of the histogram-equalization method just discussed.

EXAMPLE 3.5: Histogram equalization.

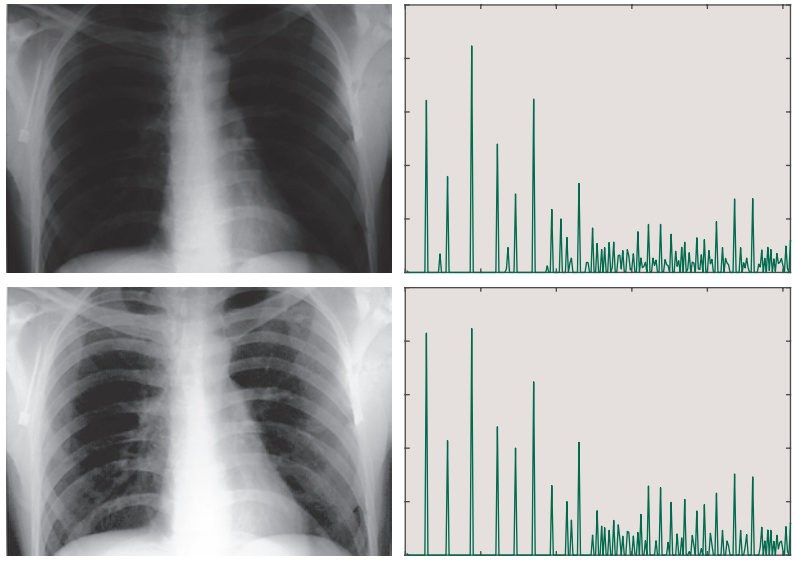
The image in Fig. 3.8(a) is a partial view of the NASA Phoenix Lander craft on the surface of Mars. The dark appearance of this image was caused by the camera adjusting itself to compensate for strong reflections from the sun. The distribution of intensities is toward the dark shades of gray, so the image histogram in Fig. 3.8(c) is biased toward the lower end of the intensity scale. This type of histogram tells us that the image is a candidate for histogram equalization, which will spread the histogram over the full range of intensities, thus increasing visible detail:

```
>> f = imread('phoenix-lander.tif');
>> figure, imshow(f); % Fig. 3.8(a).
>> g = histeq(f,256);
>> figure, imshow(g) % Fig. 3.8(b).
>> figure, imhist(f) % Fig. 3.8(c).
>> figure, imhist(g) % Fig. 3.8(d).
```

The result of histogram equalization in Fig. 3.8(b) confirms the statement made earlier that histogram equalization often brings out detail not visible in the original image. Visible detail in this image is much improved over the original. As expected, the histogram of this image [see Fig. 3.8(d)] spans the full intensity scale. Although this histogram is far from flat, the discrete histogram equalization method did an excellent job of enhancing visible detail.

As noted in Eqs. (3-8) and (3-10), the histogram equalization transformation function is the cumulative sum of the probability distribution. We compute the cumulative sum using function `cumsum`:

```
>> hnorm = imhist(f)/numel(f); % Normalized histogram.
```



a b
c d

FIGURE 3.10
(a) A dark chest X-ray image. (b) Golden histogram retrieved from storage. (c) Image processed to match the golden histogram. (d) Actual histogram of the image in (c). (Original image courtesy of Dr. Thomas R. Gest, University of Michigan Medical School.)

The following custom functions allow you to generate histograms that can be used with function `histeq` in the manner described in the preceding example. You can find these functions in your Support Package.


```
function h = fun2hist(fun,S)
```

This function converts a discrete function contained in vector `fun`, to a histogram, `h`.

The function


```
function h = manualhist(c,nb,np,plotmode)
```

allows the user to specify a shape interactively and then generates a histogram, `h`, from the specified shape.

The Toolbox provides another function for adjusting the histogram of an image to match the histogram of a reference image. The basic syntax is

`imhistmatch`

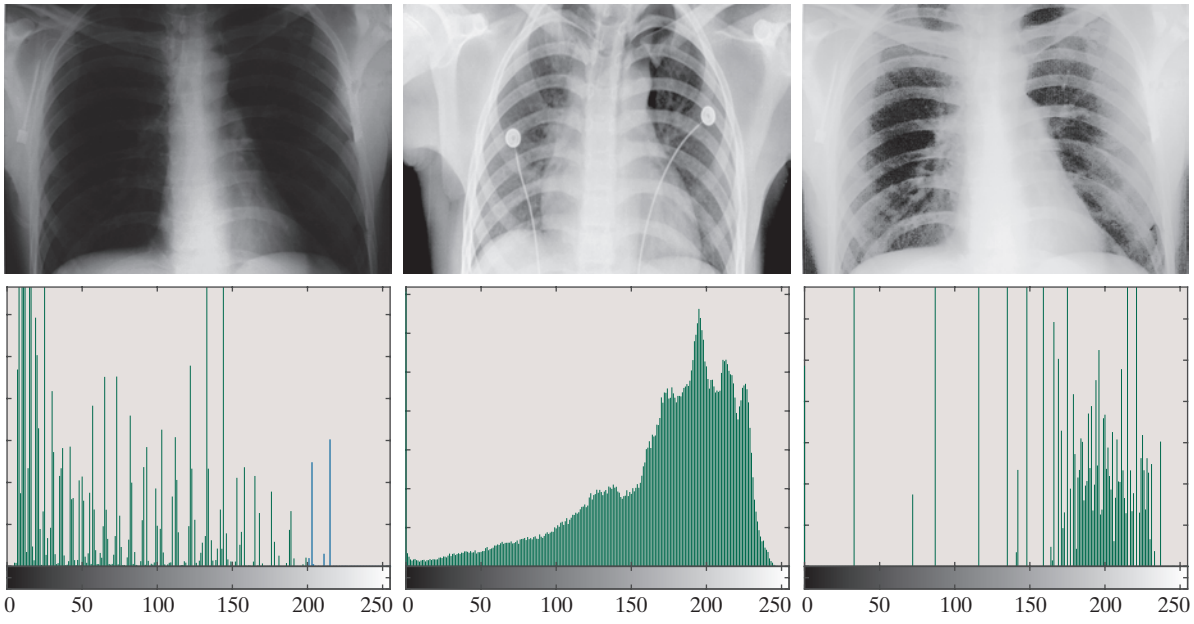
```
g = imhistmatch(f,refim,nbins)
```

where `f` is the input image, `refim` is the reference image, and `nbins` is such that the output image has no more than `nbins` discrete levels.

EXAMPLE 3.7: Using function `imhistmatch`.

Figure 3.11(a) is the same as Fig. 3.10(a). In this example, we work with function `imhistmatch` to enhance this image using the histogram of the reference image in Fig. 3.11(b), which is from a different subject. Observe that the histogram of the reference image has a much better intensity distribution. We obtained the results in Fig. 3.11 using the following commands:

```
>> f = imread('chestXray-dark.tif');
```



a b c
d e f

FIGURE 3.11 (a) Dark chest X-ray image. (b) Reference image from a different subject. (c) Result of using function `imhistmatch`. (d)–(f) Histograms of the images in the first row. (Image (a) courtesy of Dr. Thomas R. Gest, University of Michigan Medical School. Image (b) courtesy of Dr. David R. Pickens, Vanderbilt University.)

```
>> figure, imshow(f) % Fig. 3.11(a).
>> refim = imread('chestXray2-cropped.tif');
>> figure, imshow(refim) % Fig. 3.11(b).
>> g = imhistmatch(f,refim,256);
>> figure, imshow(g) % Fig. 3.11(c).
>> figure, imhist(f) % Fig. 3.11(d).
>> figure, imhist(refim)% Fig. 3.11(e).
>> figure, imhist(g) % Fig. 3.11(f).
```

As Fig. 3.11(c) shows, the quality of the enhancement is not quite as good as the result we obtained in Fig. 3.10(c), but it is a considerable improvement over the original, dark image, considering that the reference image is from a subject with a totally different anatomy. The histograms in the second row of the figure show that function `imhistmatch` transformed the original histogram so that its shape was biased toward the light end of the intensity scale, which is the dominant characteristic of the histogram of the reference image.

ADAPTIVE HISTOGRAM EQUALIZATION

Image Processing Toolbox function `adapthisteq` performs *contrast-limited adaptive histogram equalization*. Unlike the methods discussed in the previous two sections,

a b
c d

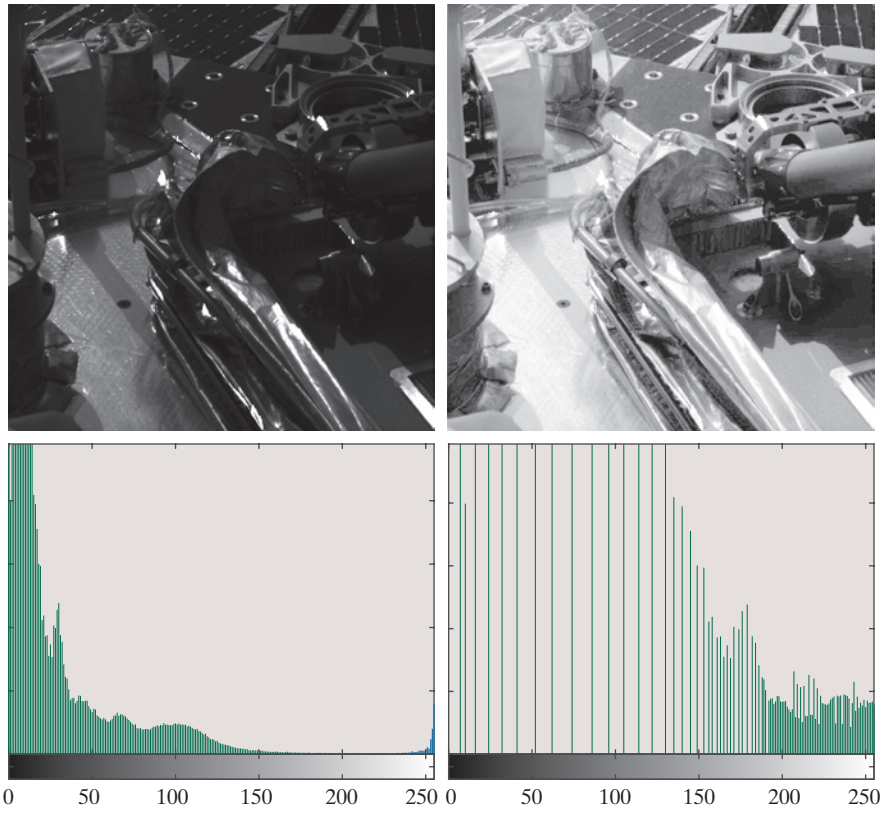


FIGURE 3.8
(a) Image from the Phoenix Lander. (b) Result of histogram equalization. (c) Histogram of (a). (d) Histogram of (b). (The colors in (c) and (d) were generated using the approach discussed in Section 2.10. Interest here is on the histogram shapes, so we do not show the bin height values. (Original image courtesy of NASA.)

`cumsum`

```
>> cdf = cumsum(hnorm);
>> r = linspace(0,1,256); % 256 values in the range [0,1].
>> figure, plot(r,cdf); % Fig. 3.9.
```

Because we normalized the histogram, we used $K = 1$ in Eq. (3-10). The transformation function in Fig. 3.9 shows that the intensities in the lower end of the intensity scale for the input image were spread out, while the intensities in the higher end were compressed to a much narrower range. This agrees with the shape of the resulting histogram in Fig. 3.8 (d).

HISTOGRAM MATCHING

There are applications that require that images be processed so that their histograms match a target shape. For example, a security imaging system may be calibrated for a specified level of ambient illumination. Images acquired during periods when the illumination changes often can be enhanced by processing them so that their histograms match a specified base histogram. This type of processing is helpful for image comparison. Another application often requiring this type of processing is medical imaging, in which an imaging system is calibrated based on a "golden" image

a b c

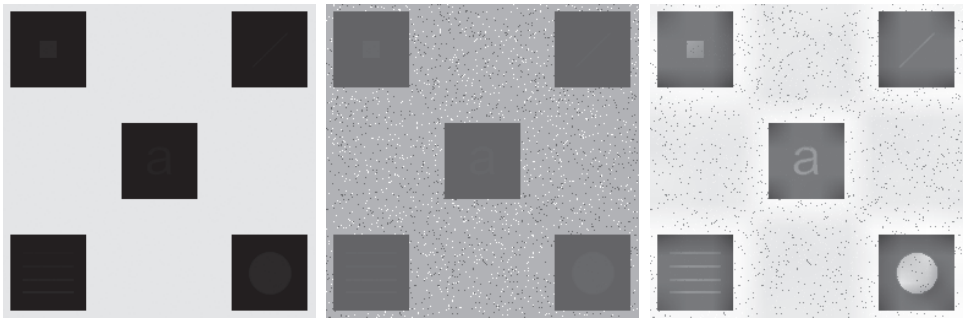


FIGURE 3.12
(a) Original image. (b) Result of histogram equalization. (c) Result of adaptive histogram equalization.

3.4 LINEAR SPATIAL FILTERING

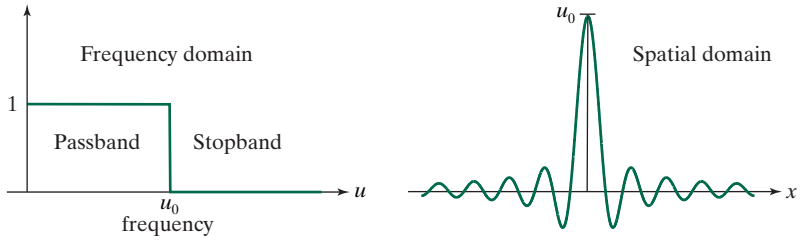
As mentioned in Section 3.1 and illustrated in Fig. 3.1, neighborhood processing consists of: (1) selecting a point (x, y) in the image; (2) performing an operation that involves only the pixels in a predefined neighborhood about (x, y); (3) letting the result of that operation be the "response" of the process at that point; and (4) repeating the process for every point in the image. Moving the neighborhood with respect to its center point creates new neighborhoods, one for each pixel in the input image. The two principal terms used to identify this operation are *neighborhood processing* and *spatial filtering*, with the second term being more prevalent. As explained in the following section, if the computations performed on the pixels of the neighborhoods are linear, the operation is called *linear spatial filtering*; otherwise it is called *nonlinear spatial filtering*. We discuss linear spatial filtering in this section and nonlinear spatial filtering in Sections 3.5 and 5.3.

Linear filtering has its roots in the use of the Fourier transform for signal processing in the frequency domain, the topic of Chapter 4. In the present chapter, we are interested in filtering operations that are performed directly on the pixels of an image. Use of the term linear spatial filtering differentiates this type of process from frequency domain filtering. The linear operations of interest in this section consist of multiplying each pixel in a neighborhood by a numerical coefficient and summing the results to obtain the response at each point (x, y). If the neighborhood is of size $m \times n$, then mn coefficients are required. The coefficients are arranged as a matrix, called a *kernel*, *mask*, *template*, or *window*, with the term kernel being the most prevalent. For reasons that will become obvious shortly, the terms *convolution filter* and *convolution kernel* also are used.

Figure 3.13 illustrates the mechanics of linear spatial filtering. The process consists of moving the center of the filter kernel w, from point to point in an image, f. The response of the filter at a point (x, y) is the sum of products of the filter kernel coefficients and the corresponding neighborhood pixels of (x, y) in the region spanned by the kernel. For a kernel of size $m \times n$, we assume typically that $m = 2a + 1$ and $n = 2b + 1$ where a and b are nonnegative integers. All this says is that our principal focus is on kernels of odd sizes. Although it certainly is not a requirement, working with odd-size kernels is more intuitive because they have an unambiguous center point that has integer coordinates.

a b

FIGURE 3.16
(a) 1-D ideal low-pass filter transfer function in the frequency domain. (b) Corresponding filter kernel in the spatial domain.



Fourier transform of the filtered frequency-domain function. The result will be a blurred spatial domain function.

Because of the duality between the spatial and frequency domains, we can obtain the same result in the spatial domain by convolving the equivalent spatial domain filter kernel with the input spatial function. The equivalent spatial filter kernel is the inverse Fourier transform of the frequency-domain filter transfer function. Figure 3.16(b) shows the spatial filter kernel corresponding to the frequency domain filter transfer function in Fig. 3.16(a). The ringing characteristics of the kernel are evident in the figure. A central theme of digital filter design theory is obtaining faithful (and practical) approximations to the sharp cut off of ideal frequency domain filters while reducing their ringing characteristics in the spatial domain.

Highpass, Bandreject, and Bandpass Filters

Spatial and frequency domain linear filters are classified into four broad categories: *lowpass*, *highpass*, *bandreject*, and *bandpass* filters. We introduced lowpass filters in the previous section. Here, we focus on the other three categories, which, as it turns out, are all derivable from lowpass filters.

Figure 3.17(a) is the same as Fig. 3.16(a); it shows the transfer function of a 1-D ideal lowpass filter in the frequency domain. We know that lowpass filters attenuate or delete high frequencies while passing low frequencies. A highpass filter behaves in exactly the opposite manner. As Fig. 3.17(b) shows, a highpass filter transfer function deletes or attenuates all frequencies below a cut-off value, u_0, and passes all frequencies above this value. From Figs. 3.17(a) and (b), we see that a highpass filter function can be obtained by subtracting a lowpass function from 1. This operation is in the frequency domain. As mentioned earlier, a constant in the frequency domain is an impulse in the spatial domain. Thus, we obtain a highpass filter kernel in the spatial domain by subtracting a lowpass filter kernel from a unit impulse with the same center as the kernel. An image filtered with this kernel is the same as an image obtained by subtracting a lowpass-filtered image from the original image.

Figure 3.17(c) shows a bandreject filter transfer function. This function can be constructed from the sum of a lowpass and a highpass filter function with different cut-off frequencies. The bandpass filter transfer function in Fig. 3.17(d) can be obtained by subtracting the bandreject function from 1, which, as you know, is a unit impulse in the spatial domain. Sometimes bandreject filters are referred to as *notch*

```
>> f8 = im2uint8(f);
>> g8r = imfilter(f8,w,'replicate');
>> figure, imshow(g8r,[])
```

Figure 3.18(f) shows the result of these operations. Here, when the output was converted to the class of the input (`uint8`) by `imfilter`, clipping caused some data loss. The reason is that the coefficients of the kernel did not sum to the range [0, 1], resulting in filtered values outside the [0, 255] range. Thus, to avoid this difficulty, we have the option of normalizing the coefficients so that their sum is in the range [0,1] (in the present case we would divide the coefficients by 31^2 so the sum would be 1), or converting the image to floating point. However, even if the second option were used, the data usually would still have to be normalized to a valid image format at some point (e.g., for storage). Either approach is valid; the key point is that data ranges have to be kept in mind to avoid unexpected filtering results.

FILTER KERNELS

The result of spatial convolution or correlation depends on the kernel size and the nature of its elements. In this section we discuss several properties of filter kernels and list the standard kernels supported by the Toolbox.

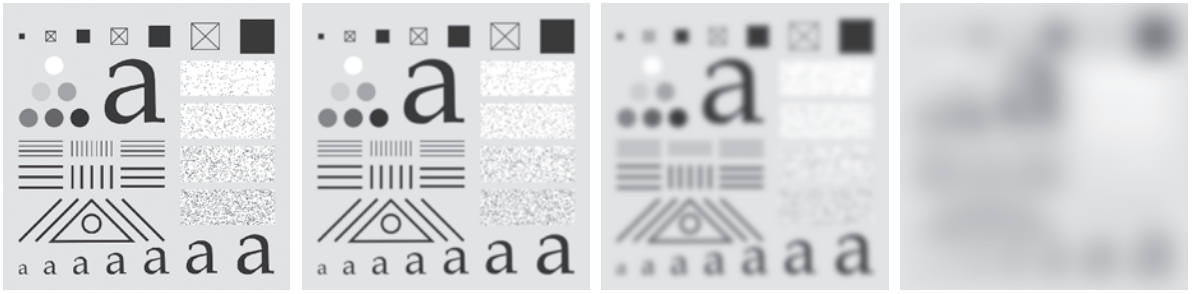
How Spatial Filter Kernels Are Constructed

There are three basic approaches for constructing spatial filter kernels.

1. Formulate a kernel based on mathematical properties. For example, an image-blurring kernel can be based on computing the average of pixels in neighborhoods of an image. Computing an average is analogous to integration. Conversely, a kernel based on computing the local (neighborhood) derivatives of an image sharpens the image.
2. Sample a 1-D spatial function that has a desired property and then generate a 2-D circularly-symmetric kernel by rotating the function about its center. Analogously, we can sample a 2-D spatial function whose shape has a desired property and set the 2-D kernel equal to the samples.
3. Design a spatial filter with a specified frequency response. This approach falls in the area of *digital filter design*. A 1-D spatial filter with the desired response is obtained (typically using *filter design software*). The 1-D filter values can be expressed as a vector, `v`, and a 2-D separable kernel can then be obtained using the methods discussed in the following section. Alternatively, the 1-D filter function can be rotated about its origin to generate a 2-D kernel that approximates a circularly-symmetric function.

Separable Filter Kernels

A 2-D function $G(x, y)$ is said to be *separable* if it can be written as the product of two 1-D functions: $G(x, y) = G_1(x)G_2(y)$. A spatial filter kernel is a matrix and a *separable kernel* is a matrix that can be expressed as the outer product of two vectors. For example, the 2×3 kernel



a b c d

FIGURE 3.19 (a) Test pattern of size 1024 × 1024 pixels. (b)–(d) Results of using functions `fspecial` and `imfilter` to lowpass filter the image in (a) with Gaussian kernels of various sizes, as explained in the text.

```
>> r = ceil(6*sig)
r =
    62
>> r = r - 1;
>> w2 = fspecial('gaussian',r,sig);
>> g2 = imfilter(f,w2,'replicate');
>> figure, imshow(g2) % Fig. 3.19(c).
```

As Fig. 3.19(c) shows, setting `sig` to 1% of the image height resulted in significant blurring, but all objects are still recognizable. Finally, we try

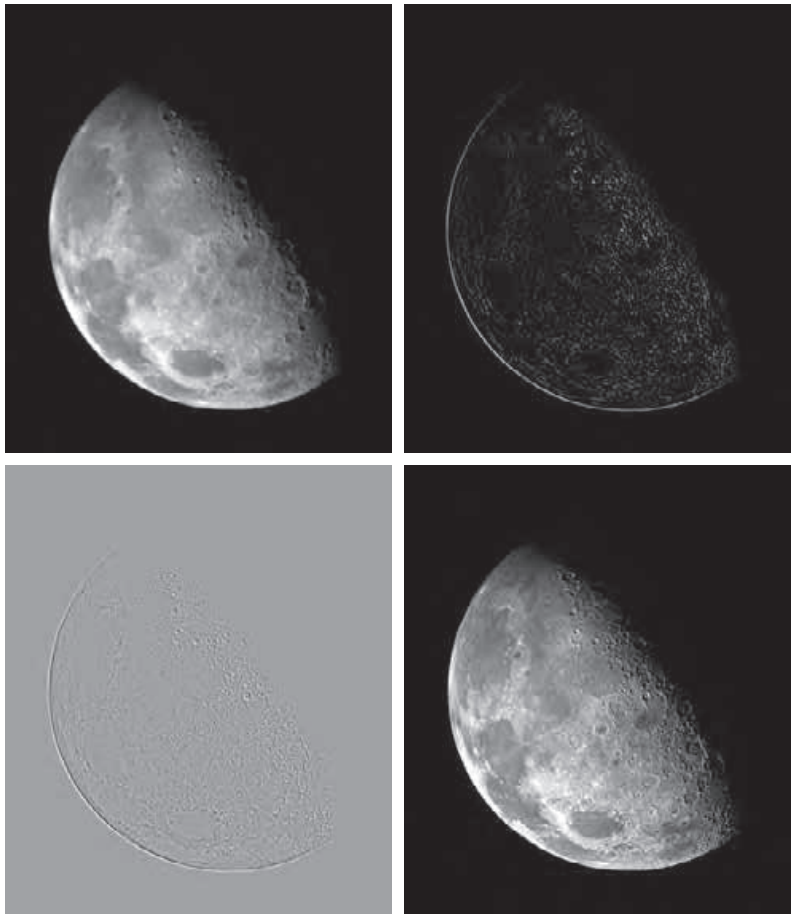
```
>> sig = 0.05*M;
r = ceil(6*sig)
r =
    308
>> r = r - 1;
>> w3 = fspecial('gaussian',r,sig);
>> g3 = imfilter(f,w3,'replicate');
>> figure, imshow(g3) % Fig. 3.19(d).
```

As Fig. 3.19(d) shows, a kernel of 0.5% of the image height resulted in extreme blurring. Mild blurring is used routinely for noise reduction, while extreme blurring (typically combined with thresholding) is used to extract dominant regions of an image.

EXAMPLE 3.11: Using functions `fspecial` and `imfilter` for image sharpening.

We illustrate the use of functions `fspecial` and `imfilter` for image sharpening by enhancing an image with a Laplacian filter kernel. We also discuss some important aspects of using `imfilter` with integer-class images.

Figure 3.20(a) is a mildly blurred image of the North Pole of the moon. Enhancement in this case consists of sharpening the image, while preserving as much of its gray tonality as possible. We use function `fspecial` to obtain a Laplacian kernel:



a b
c d

FIGURE 3.20
(a) Image of the North Pole of the moon. (b) Laplacian filtered image obtained with an input image of class `uint8`. (c) Laplacian filtered image obtained using a floating point input image. (d) Enhanced result obtained by subtracting (c) from (a). (Original image courtesy of NASA.)

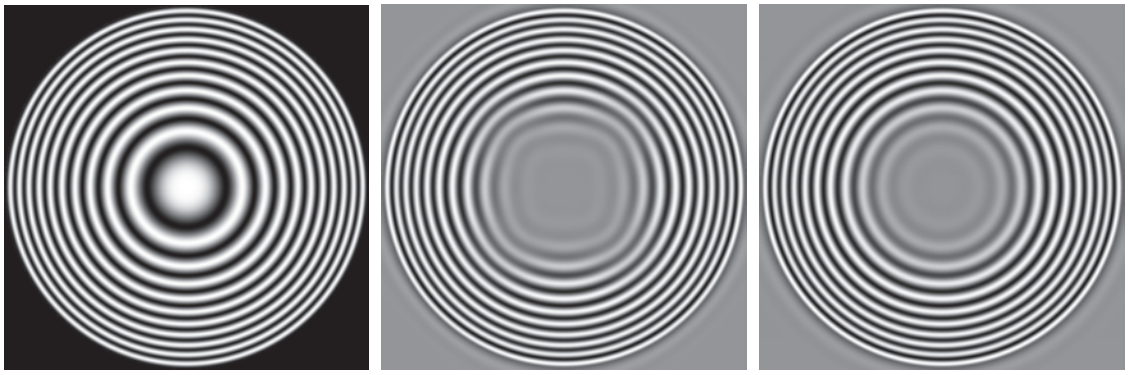
```
>> w = fspecial('laplacian',0)
w =
    0.0000    1.0000    0.0000
    1.0000   -4.0000    1.0000
    0.0000    1.0000    0.0000
```

Note that this kernel is of class `double`.

Next we apply `w` to the input image, [Fig 3.20(a)], which is of class `uint8`:

```
>> f = imread('moon-blurry.tif');
>> figure, imshow(f) % Fig. 3.20(a).
>> g1 = imfilter(f,w,'replicate');
>> imshow(g1,[]) % Fig. 3.20(b).
```

Figure 3.20(b) shows the resulting image. This result looks reasonable, but it has a problem: all its pixels are positive. Because the coefficients of the kernel sum to zero, we know, as mentioned earlier, that



a b c

FIGURE 3.24 (a) Zone plate image. (b) Result of highpass filtering with kernel `w1HP2D` (observe the slight distortion of the concentric rings). (c) Result of highpass filtering with kernel `w2HP2D`.

is the same as filtering with an impulse minus filtering with a lowpass kernel. But filtering with an impulse yields the image itself, so the net result is the image minus a lowpass-filtered version of that image.

Unsharp Masking Filters

We discussed in Examples 3.11 and 3.12 how to use the Laplacian for image sharpening. A process for the same purpose that has been used since the 1930s by the printing and publishing industry to sharpen images is based on subtracting an unsharp (smoothed) version of an image from the original image. This process, called *unsharp masking*, consists of the following steps:

1. Blur a copy of the original image.
2. Subtract the blurred image from the original (the resulting difference is called the *mask*.)
3. Add the mask to the original.

In case you are wondering why a high-pass (sharpened) image is constructed from a blurred image, recall that unsharp masking originated in the analog photography field, which had no easy way to sharpen an image directly.

As indicated in Table 3.3, subtracting a lowpass kernel from an impulse yields a highpass kernel. This implies that filtering an image with a lowpass kernel and subtracting the result from the original is equivalent to highpass filtering, which sharpens an image (but reduces gray-level tonality). Thus, the mask in Step 2 above is just a sharpened (highpass-filtered) image. Adding the mask to the original image (Step 3) recovers the tonality lost by highpass filtering.

In earlier releases of MATLAB, the Toolbox function `fspecial` had an option for performing unsharp masking. This option has been replaced in favor of a separate function called `imsharpen`, which has the following syntax:

`imsharpen`

```
g = imsharpen(f,Name1,Value1,Name2,Value2,...)
```

a b c
d e f

FIGURE 3.25 (a) Original, soft-tone image of size 496×600 pixels. (b) Result of using function `imsharpen` with its default values. (c) Result using a `Radius` equal to 5. (d) Result using a `Radius` of 5 and an `Amount` of 1.5. (e) Same as (d) but with an `Amount` of 2. (f) Same as (e), but with a `Threshold` of 0.5.

This result is not appreciably different from the original. The reason is that increasing the `Threshold` parameter allowed significantly fewer pixels to be enhanced. This parameter can be used to exclude intensities in an image from being sharpened. An example where this could be helpful is in images with large dark noisy areas, which would otherwise be sharpened and potentially could become more visible.

3.5 NONLINEAR SPATIAL FILTERING

Nonlinear spatial filtering is based on neighborhood operations also and the mechanics of sliding the center point of an $m \times n$ rectangle through an image are the same as discussed in the previous section for linear filtering. However, whereas linear spatial filtering is based on computing the sum of products (which is a linear operation), nonlinear spatial filtering is based, as the name implies, on nonlinear operations involving the pixels in a neighborhood being processed. For example, letting the response at a point be equal to the maximum pixel value in its neighborhood is a nonlinear filtering operation. Another difference is that the concept of a kernel is not as prevalent in nonlinear processing. The idea of filtering carries over, but the filter should be visualized as a nonlinear function that operates on the pixels of a neighborhood and whose response constitutes the result of the nonlinear operation.

```
>> g = colfilt(f,[m n],'sliding',maxfilt);
```

Note that we called `maxfilt` without `@` because `maxfilt` had already been defined as a function handle. If, instead, `maxfilt` had been a regular function, we would have used `@maxfilt` as the rightmost input argument in `colfilt`. Also, keep in mind that `A` always has `mn` rows, but the number of columns is variable. Therefore `maxfilt` (or any other function handle passed to `colfilt`) has to be able to cope with a variable number of columns. The syntax `max(A,[],1)` does precisely this.

See the margin note in the next section indicating that max filters implemented using morphology (the topic of Chapter 10) run much faster and use less memory.

The filtering process in this case consists of computing the maximum of all pixels in every $m \times n$ neighborhood. As noted earlier, the key requirements are that the function operate on the columns of `A`, no matter how many there are, and return a row vector containing the result for all individual columns (our function `maxfilt` does this because of the syntax we used in function `max`). Function `colfilt` then takes those results and rearranges them to produce the output image, `g`.

Finally, we remove the padding we inserted using `padarray` (as noted earlier, `colfilt` removes its own padding before outputting `g`):

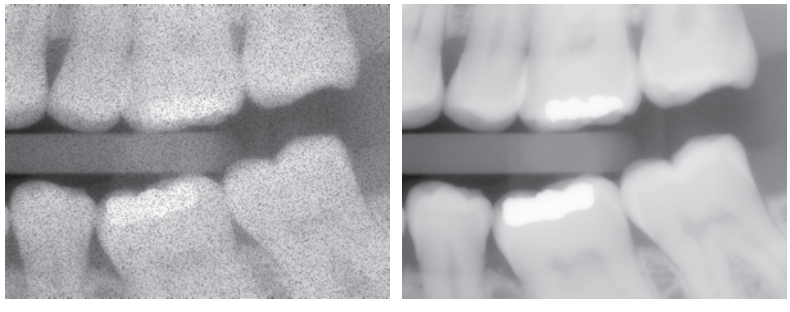
```
>> [M,N] = size(f); % Size of padded f.
>> g = g(1:M - 2*m,1:N - 2*n);
```

so that `g` is of the same size as `f`.

The following code illustrates the concepts in the preceding discussion, using an image corrupted by pepper noise:

We discuss pepper noise and other noise models in Section 5.2.

```
>> % Read uint8 dental Xray image corrupted by pepper noise.
>> f = imread('dentalXray-pepper-noise.tif');
>> figure, imshow(f) % Fig. 3.26(a).

>> % Construct the max filter.
>> maxfilt = @(A) max(A,[],1);

>> % Neighborhood size.
>> m = 3; n = 3;

>> % Manually pad the input image.
>> fp = padarray(f,[m n],'replicate');

>> % Apply the max filter to the padded image.
>> g = colfilt(fp,[m n],'sliding', maxfilt);
```



a b

FIGURE 3.26 (a) Image corrupted by pepper noise. (b) Result of performing max filtering using function `colfilt`.

a b c

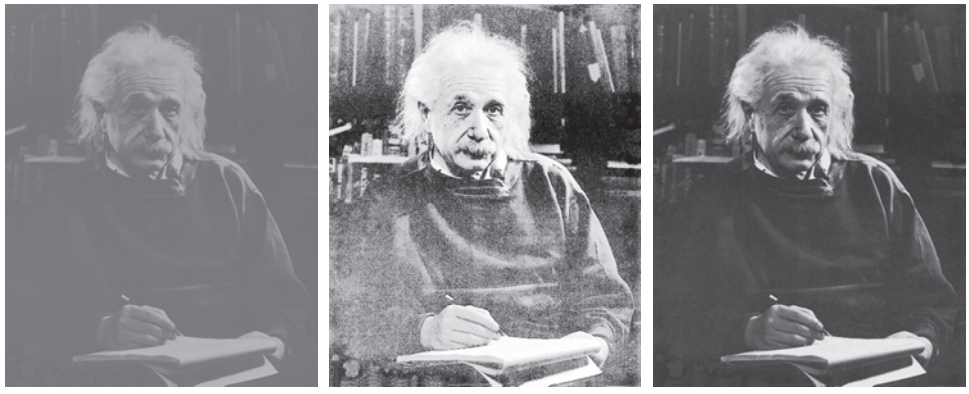


FIGURE 3.36 (a) Low-contrast image. (b) Result of histogram equalization. (c) Result of fuzzy, rule-based contrast enhancement.

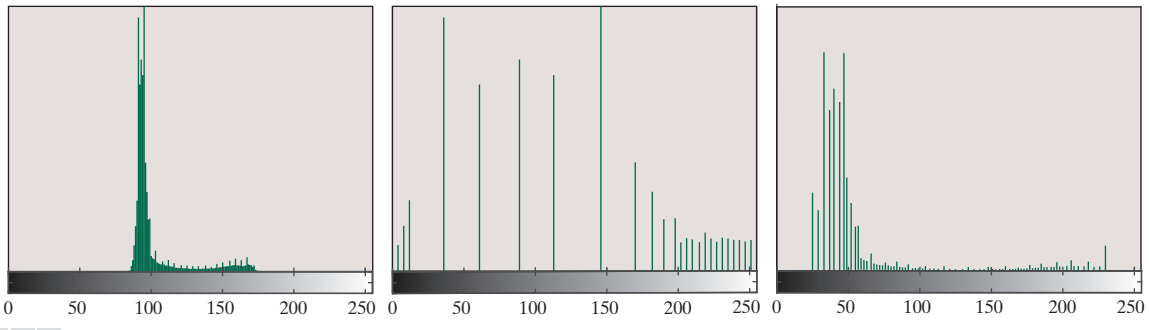
EXAMPLE 3.19: Using fuzzy functions to implement fuzzy contrast enhancement.

Figure 3.36(a) shows an image, `f`, whose intensities span a narrow range of the gray scale, as the histogram in Fig. 3.37(a) (obtained using `imhist`) shows. The net result is an image with low contrast.

Figure 3.36(b) is the result of using histogram equalization to increase image contrast. As the histogram in Fig. 3.37(b) shows, the entire gray scale was spread out but, in this case, the spread was excessive in the sense that contrast was increased, but the result is an image with an "over exposed" appearance. For example, the details in Professor Einstein's forehead and hair are mostly lost.

Figure 3.36(c) shows the result of the following fuzzy operations:

```
>> % Specify the input membership functions.
>> udark = @(z) 1 - sigmamf(z,0.35,0.5);
>> ugray = @(z) triangmf(z,0.35,0.5,0.65);
>> ubright = @(z) sigmamf(z,0.5,0.65);

>> % Plot the input membership functions. See Fig. 3.35(a).
>> fplot(udark,[0 1],20)
>> hold on
```



a b c

FIGURE 3.37 Histograms of the images in Fig. 3.36(a), (b), and (c), respectively.

a b c

FIGURE 3.41 (a) Image from a CT scan of a human head. (b) Result of fuzzy spatial filtering using the membership functions in Fig. 3.39 and the rules in Fig. 3.40. (c) Result after intensity scaling. The thin black picture borders in (b) and (c) were added for clarity; they are not part of the data. (Original image courtesy of Dr. David R. Pickens, Vanderbilt University.)

Summary

The material in this chapter is a foundation for numerous topics that you will encounter in subsequent chapters. For example, we use spatial processing in Chapter 5 in connection with image restoration, where we also take a closer look at noise reduction and noise generating functions in MATLAB. Some of the spatial kernels that were mentioned briefly here are used extensively in Chapter 11 for edge detection in segmentation applications. The concepts of convolution and correlation are explained again in Chapter 4 from the perspective of the frequency domain. Conceptually, neighborhood processing and the implementation of spatial filters will surface in various discussions throughout the book. In the process, we will extend many of the discussions we began here and introduce additional aspects of how spatial filters can be implemented efficiently in MATLAB.

MATLAB Projects

*Solutions to the projects marked with an asterisk * are in the DIPUM3E Student Support Package (consult the book web site). All your code must be documented so that typing* `help` *at the prompt, followed by the script or function name, gives enough detail for a user to be able to run it. Test the functionality of all your code thoroughly.*

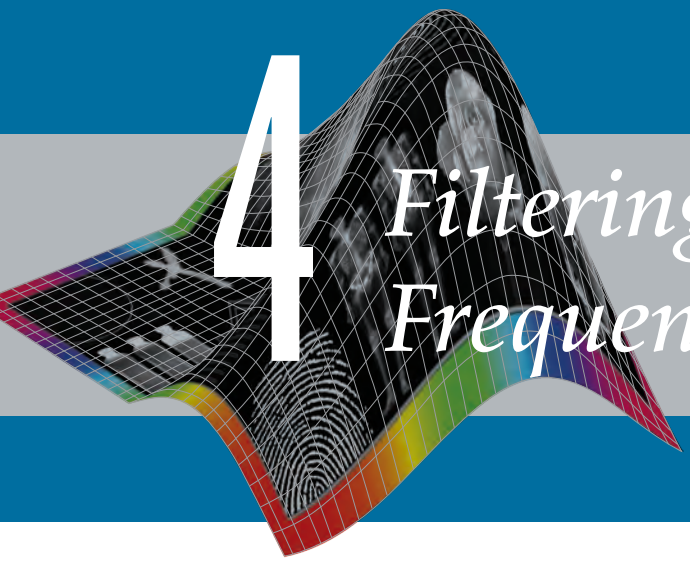
3.1 Read the image `spillway.tif` and enhance it to bring out details in the coastal road that are barely visible in the original image. Try enhancement techniques based on the following intensity transformations:

(a)* log.

(b) gamma.

(c) stretch.

(d) Your specified transformation function.



4 Filtering in the Frequency Domain

If you want to find the secrets of the universe, think in terms of energy, frequency and vibration.
Nikola Tesla

For the most part, this chapter parallels the filtering topics discussed in Chapter 3, but with all filtering done in the frequency domain using the discrete Fourier transform. In addition to being a cornerstone of linear filtering, the Fourier transform offers considerable flexibility in the design and implementation of filtering solutions in areas such as image enhancement, image restoration, image data compression, and a host of other applications of practical interest. In this chapter, the focus is on the foundation of how to perform frequency domain filtering using MATLAB. As we did in Chapter 3, we illustrate filtering in the frequency domain using examples of image enhancement, including lowpass filtering for image smoothing, highpass and high-frequency emphasis filtering for image sharpening, and selective filtering for the removal of periodic interference. Although most of the examples in this chapter deal with image enhancement, the concepts and techniques developed in the following sections are quite general, as illustrated by other applications of this material in Chapters 5, 9, 11, and 13.

Functions Developed in this Chapter:

- `paddedsize` determines the minimum padding sizes needed for filtering in the frequency domain.
- `dftfilt` performs filtering in the frequency domain.
- `dftuv` generates a meshgrid array for computing distances used in forming filter transfer functions.
- `lpfilter` generates lowpass filter transfer functions.
- `hpfilter` generates highpass filter transfer functions.
- `bandfilter` generates bandpass and band-reject filter transfer functions.
- `cnotch` implements notchpass and notch-reject filter transfer functions.
- `recnotch` implements rectangular pass and reject filter transfer functions.
- `iseven` determines which elements of an array are even numbers.
- `isodd` determines which elements of an array are odd numbers.

4.1 THE 2-D DISCRETE FOURIER TRANSFORM

Let $f(x, y)$ for $x = 0, 1, 2, \ldots, M-1$ and $y = 0, 1, 2, \ldots, N-1$ denote a digital image of size $M \times N$ pixels. The 2-D *discrete Fourier transform* (DFT) of $f(x, y)$, denoted by $F(u, v)$, is given by the equation

$$F(u,v) = \sum_{x=0}^{M-1} \sum_{y=0}^{N-1} f(x,y) e^{-j2\pi(ux/M + vy/N)} \tag{4-1}$$

for $u = 0, 1, 2, \ldots, M-1$ and $v = 0, 1, 2, \ldots, N-1$. We could expand the exponential term into sine and cosine functions using Euler's formula, with the variables u and v representing *sinusoidal frequencies* (x and y are summed out). The *frequency domain* is the coordinate system spanned by $F(u, v)$ with u and v as *frequency variables*. This is analogous to the *spatial domain* studied in Chapter 3, which is the coordinate system spanned by $f(x, y)$, with x and y as *spatial variables*. The $M \times N$ rectangular region defined by $u = 0, 1, 2, \ldots, M-1$ and $v = 0, 1, 2, \ldots, N-1$ is often referred to as the *frequency rectangle*. Clearly, the frequency rectangle is of the same size as the input image.

The *inverse discrete Fourier transform* (IDFT) of $F(u, v)$ is given by

$$f(x,y) = \frac{1}{MN} \sum_{u=0}^{M-1} \sum_{v=0}^{N-1} F(u,v) e^{j2\pi(ux/M + vy/N)} \tag{4-2}$$

for $x = 0, 1, 2, \ldots, M-1$ and $y = 0, 1, 2, \ldots, N-1$. Thus, given $F(u, v)$, we can get $f(x, y)$ back using the IDFT. The values of $F(u, v)$ in this equation are referred to as the *coefficients of the Fourier transform*.

In some formulations of the DFT, the $1/MN$ term appears in front of the transform and in others it is used in front of the inverse. MATLAB's implementation uses the term in front of the inverse, as shown in the preceding equation. Because array indices in MATLAB start at 1 rather than 0, `F(1,1)` and `f(1,1)` in MATLAB correspond to the mathematical quantities $F(0,0)$ and $f(0,0)$ in the transform and its inverse. In general `F(i,j)` = $F(i-1, j-1)$ and `f(i,j)` = $f(i-1, j-1)$, for i,`i` $= 1, 2, \ldots, M$ and j,`j` $= 1, 2, \ldots, N$.

The value of the transform at the origin of the frequency domain, $F(0,0)$, is called the *dc term* or *dc component* of the Fourier transform. This terminology is from electrical engineering, where "dc" signifies direct current (current of zero frequency). It is not difficult to show that the average value of $f(x, y)$ is equal to $F(0,0)$ divided by MN:

$$\bar{f} = \frac{1}{MN} F(0,0) \tag{4-3}$$

Even if $f(x, y)$ is a real function, its transform, $F(u, v)$, is complex in general. We analyze a Fourier transform visually by computing its *spectrum*, which is defined as

$$|F(u,v)| = \left[R^2(u,v) + I^2(u,v) \right]^{\frac{1}{2}} \tag{4-4}$$

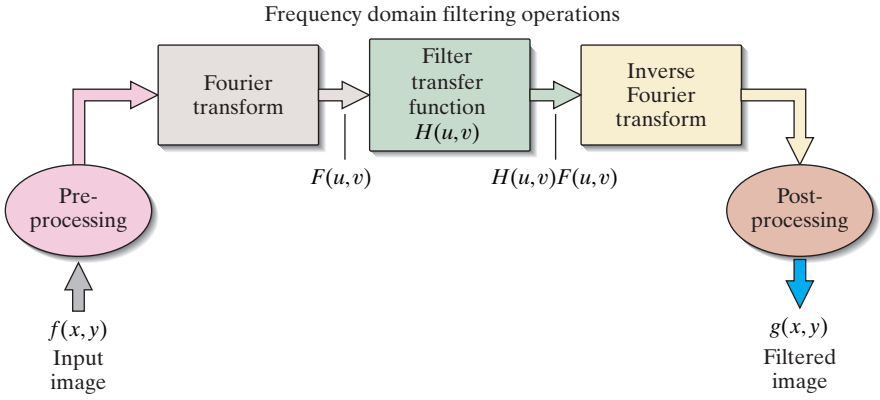


FIGURE 4.8 Steps for filtering an image in the frequency domain.

Figure 4.8 is a schematic of the preceding filtering steps. The preprocessing stage performs the tasks of computing the padding parameters, padding the input image, and generating a transfer function. Postprocessing typically includes cropping the output image and converting it to the class of the input.

The filter transfer function $H(u,v)$ in Fig. 4.8 multiplies both the real and imaginary parts of $F(u,v)$. If $H(u,v)$ is real, then the phase of the result is not changed, as you can see by noting in Eq. (4-5) that if the multipliers of the real and imaginary parts are equal, they cancel out, leaving the phase angle unchanged. Filters with this property are called *zero-phase-shift filters*. These are the only types of linear filters considered in this chapter.

It is well known from linear system theory that, under certain mild conditions, inputting an impulse into a linear system completely characterizes the system. When using the techniques developed in this chapter, the response of a linear system, including the response to an impulse, also is finite. If the linear system is a filter, then we can completely determine the filter transfer function by observing the filter response to an impulse. A filter characterized in this manner is called a *finite-impulse-response* (FIR) filter. All the linear filters in this book are FIR filters.

A FUNCTION FOR FILTERING IN THE FREQUENCY DOMAIN

The filtering steps just described are used throughout this chapter and parts of the next, so it will be convenient to have available a function that accepts as inputs an image and a filter transfer function, handles all the filtering details, and outputs the filtered, cropped image. The following function does this. It is assumed that the transfer function has been sized appropriately, as explained in Step 4 of the filtering procedure.


```
function g = dftfilt(f,H,padMethod,classOut)
%DFTFILT Performs frequency domain filtering.
%   g = DFTFILT(f,H,padMethod,classOut) filters f in the frequency
%   domain using the filter transfer function H. The output, g, is the
%   filtered image, which is of the same size as f. padMethod and
```

```
%   classOut are explained below. Any intermediate arguments that are
%   not specified should be replaced by []. For example, if classOut is
%   specified but padMethod is not, we use g = dftfilt(f,H,[],classOut).
%   Non-specified arguments are replaced by their default values.
%
%   Valid values of classOut are:
%
%   'same'      The output will be of the same class as the input.
%
%   'floating'  The output will be floating point of class double. This
%               is the default.
%
%   Valid values of padMethod are:
%
%   'zeros'     Pads the input image with 0s using the 'post' option in
%               the Toolbox function padarray. This is the default.
%   'symmetric' Pads the image using the 'symmetric' and 'post' options
%               in the Toolbox function padarray
%   'replicate' Pads the image using the 'replicate' and 'post' options
%               in the Toolbox function padarray.
%   'circular'  Pads the image using the 'circular' and 'post' options
%               in the Toolbox function padarray.
%
%   DFTFILT automatically pads f to be the same size as H. Both f and H
%   must be real. H must be an uncentered, symmetric filter transfer
%   function, as illustrated in Fig. 4.2(a). (You can uncenter a
%   centered transfer function using function fftshift.)

% Set Defaults. Note: Function padarray used below does not recognize a
% padvalue specified as 'zeros', which we use for notational consistency
% in the input to function dftfilt. A padvalue of 'zeros' is converted
% to the numerical zero (0);

if (nargin < 4) || isempty(classOut)
   classOut = 'floating';
end

if (nargin < 3) || isempty(padMethod) || isequal(padMethod,'zeros')
   padMethod = 0;
end

% Convert the input to floating point. Will need revertClass later.
[f,revertClass] = tofloat(f);
[M,N] = size(f);

% Pad f to the size of the transfer function, using the default or
% specified padmethod.
f = padarray(f, [size(H,1) - M, size(H,2) - N], padMethod, 'post');

% Obtain the FFT of the input image. The image was already padded to be
% of the same size as the filter transfer function.
F = fft2(f);

% Perform filtering.
g = ifft2(H.*F);
```

```
RI = D <= C0 - (W/2);   % Points of region inside the inner boundary of
                        % the reject band are labeled 1. All other
                        % points are labeled 0.

RO = D >= C0 + (W/2);   % Points of region outside the outer boundary
                        % of the reject band are labeled 1. All other
                        % points are labeled 0.

H = tofloat(RO | RI);   % Ideal bandreject transfer function.

%-----------------------------------------------------------------------%
function H = butterworthReject(D,C0,W,n)
H = 1./(1 + ((D*W)./(D.^2 - C0^2)).^(2*n));

%-----------------------------------------------------------------------%
function H = gaussReject(D,C0,W)
H = 1 - exp(-((D.^2 - C0^2)./(D.*W + eps)).^2);
```

EXAMPLE 4.7: Lowpass, highpass, and band filtering.

Figure 4.20 is the same zoneplate image we used in Example 3.13, which consists of concentric annular regions whose frequencies increase as a function of increasing distance from the center. This test pattern is very useful for showing filtering techniques that affect regions of the image based on their spatial frequency content.

First, we show that we can get results in the frequency domain that are equivalent to the spatial results in Example 3.13. We obtained Fig. 4.21(a) using the following commands:

```
>> f = imread('zoneplate.tif');
>> [M,N] = size(f);
>> % Lowpass filter the image.
>> HLP = lpfilter('butterworth',M,N,15,3);
>> gLP = dftfilt(f,HLP,'zeros');
>> figure, imshow(gLP) % Fig. 4.21(a).
```

This result is almost identical to the image in Fig. 3.24(b), which we obtained using spatial filtering. The low frequencies were passed without modification and the high frequencies were attenuated. Here we used zero padding because the image border is black.

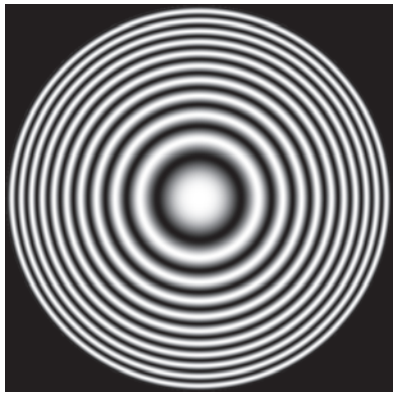
Figure 4.21(b) is the result of using function `hpfilter` with the same settings as above. As expected, the background of the filtered image is dark due to clipping of negative values by the display, as we explained earlier in connection with Fig. 4.16. Figure 4.21(c) is the result of using the function `intensityScaling` to scale the image intensities to the full intensity range. Observe how the low frequencies were attenuated and the high frequencies were passed without modification, as we expect from a highpass filter.

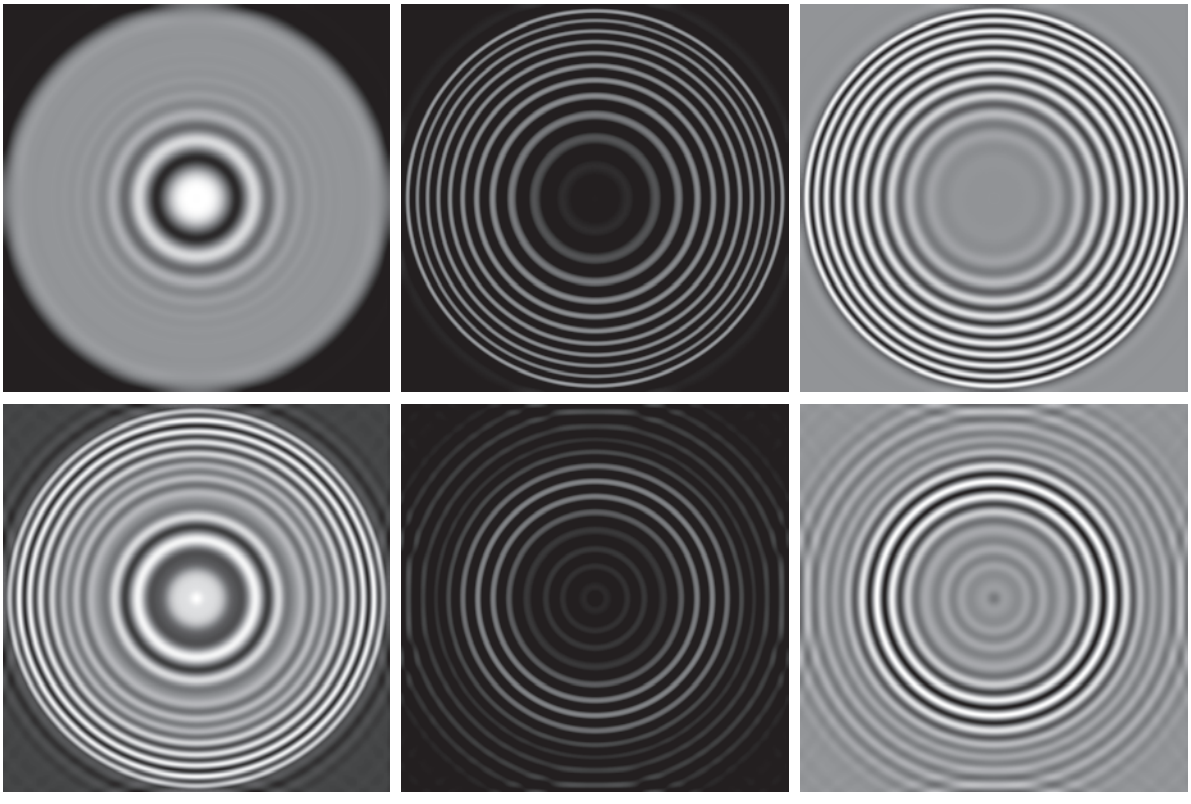
Figure 4.21(d) is the result of bandreject filtering using the commands:

```
>> HBR = bandfilter('butterworth','reject',M,N,30,8,3);
>> gBR = intensityScaling(dftfilt(f,HBR,'zeros'));
>> figure, imshow(gBR) % Fig. 4.21(d).
```

We moved the center of the filter band further out than before in order to show clearly that the mid

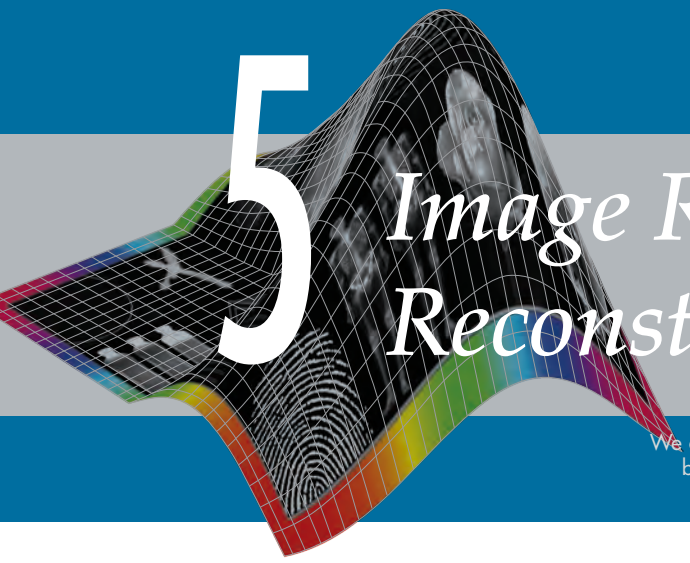
FIGURE 4.20
Zoneplate image.





a b c
d e f

FIGURE 4.21 (a) Lowpass-filtered zoneplate image. (b) Highpass filtered image. (c) Highpass filtered image after intensity scaling. (d) Result of bandreject filtering. (e) Result of bandpass filtering. (f) Bandpass filtered image after intensity scaling. Compare (a) with Fig. 3.24(c).



5 Image Restoration and Reconstruction

We are all hungry and thirsty for concrete images. Abstract art will have been good for one thing: To restore its exact virginity to figurative art.
Salvador Dali

The objective of restoration is to improve a given image in some predefined sense. Although there are areas of overlap between image enhancement and image restoration, the former is largely a subjective process, while image restoration is for the most part an objective process. Restoration attempts to reconstruct or recover an image that has been degraded by using a priori knowledge of the degradation phenomenon. Thus, restoration techniques are oriented toward modeling the degradation and applying the inverse process in order to recover the original image. This approach usually involves formulating a criterion of goodness that yields an optimal estimate of the desired result. In this chapter we explore how to use MATLAB and the Image Processing Toolbox to model degradation phenomena and to formulate restoration solutions. As in Chapters 3 and 4, some restoration techniques are best formulated in the spatial domain, while others are better suited for the frequency domain. Both methods are investigated in the sections that follow. We conclude the chapter with a discussion on the Radon transform and its use for image reconstruction from projections.

Functions Developed in this Chapter:

- `imnoise2` corrupts an image with noise of a specified PDF and also generates the noise pattern itself.
- `imnoise3` generates sinusoidal noise patterns that can be added to an image to simulate periodic interference.
- `statmoments` computes an arbitrary number of statistical central moments of an image histogram.
- `histroi` computes the intensity histogram of an arbitrary region of interest (ROI).
- `spfilt` implements spatial filtering using linear and nonlinear filters.
- `adpmedian` performs adaptive local median filtering.

returning the row and column indices, the third form also returns the nonzero values of A as a column vector, v.

The first form treats the matrix A in the linear index format A(:), so idx is a column vector. This form is quite useful in image processing. For example, to find and set to 0 all pixels in an image whose values are less than 128 we write

```
>> idx = find(A < 128);
>> A(idx) = 0;
```

See Section 2.8 regarding logical indexing.

We can do the same operation in one line of code using *logical indexing*:

```
>> A(A < 128) = 0;
```

Recall that the logical statement A < 128 returns a 1 for the elements of A that satisfy the logical condition and 0 for those that do not. As another example, to set to 128 all pixels in the interval [64, 192] we write

```
>> idx = find(A >= 64 & A <= 192);
>> A(idx) = 128;
```

Equivalently, we could write

```
>> A(A >= 64 & A <= 192) = 128;
```

The type of indexing just discussed is used frequently throughout the book.

Like imnoise, the following function, imnoise2, corrupts an image with noise of a specified PDF and, in addition, it generates a matrix, R, of the same size as the input image and whose elements are from a specified PDF. In other words, imnoise2 corrupts an image with noise and also outputs the spatial noise pattern itself. The user specifies the desired values for the noise parameters directly. For example, a noise matrix resulting from salt-and-pepper noise has three values: 1 corresponding to salt noise, 0 corresponding to pepper noise, and 0.5 corresponding to no noise. This array needs to be processed further to make it useful. To corrupt an image with this array, we find (using function find or the logical indexing illustrated above) all the coordinates in R that have value 1 and set all the coordinates in the image to the highest possible value (255 for an 8-bit image). Similarly, we find all the coordinates in R that have value 0 and set the corresponding coordinates in the image to the smallest possible intensity value (usually 0). All other pixels are left unchanged. This process simulates the manner in which salt-and-pepper noise affects an image. The function does this automatically to the input image, but it is important that you understand the structure of R in case you want to use it for some other purpose.

Note that function imnoise2 supports several PDFs not found in the Toolbox function imnoise, and vice versa.

Observe in the code for imnoise2 how the switch/case statements are kept simple; that is, unless case computations can be implemented with a few lines of code, they are delegated to individual, local functions appended at the end of the main program. This clarifies the logical flow of the code. The objective is to *modularize* the code as much as possible for ease of interpretation and maintenance.


```
function [fn,R] = imnoise2(f,type,a,b)
%IMNOISE2 Outputs noisy image and random matrix with given PDF.
%   [Fn,R] = IMNOISE2(F,TYPE,A,B) generates a noise matrix, R, of the
%   same size as input grayscale image F, whose elements are random
%   numbers of the specified TYPE, with parameters A and B. The output
%   noisy image is formed either by adding R to it or, in the case of
%   salt and pepper, by modifying F based on the values of R, as
%   explained in Section 5.2 of DIPUM3E. The noisy image Fn is of class
%   double, scaled to the full range [0,1]. The input image can be of
%   any valid grayscale class.
%
%   Valid values for TYPE and parameters A and B are:
%
%     'uniform'        Uniform random numbers in the interval (a,b). The
%                      default values are (0,1).
%
%     'gaussian'       Gaussian random numbers with mean a and standard
%                      deviation b. The default values are a = 0, b = 1;
%
%     'salt & pepper'  Salt and pepper random numbers of value 1 (salt)
%                      with probability Ps = a, and value 0 (pepper)
%                      with probability Pp = b. The default values are
%                      Ps = Pp = 0.05. The noise matrix R is assigned
%                      three values: R(x,y) = 1 (white), for salt noise
%                      at coordinates (x,y); R(x,y) = 0 (black) for
%                      pepper noise at coordinates (x,y); and R(x,y) =
%                      0.5 for no noise at coordinates (x,y). Therefore,
%                      R is not simply added to an image to make it
%                      noisy. Instead, we assign to the image a value of
%                      0 or 1 at the corresponding locations in R with
%                      values of 0 or 1. The image is unchanged at the
%                      coordinates where the values of R are 0.5.
%
%     'lognormal'      Lognormal random numbers with offset a and shape
%                      parameter b. The defaults are a = 1 and b = 0.25.
%
%     'rayleigh'       Rayleigh random numbers with parameters a and b.
%                      The default values are a = 0 and b = 1.
%
%     'exponential'    Exponential random numbers with parameter a. The
%                      default is a = 1.
%
%     'erlang'         Erlang (gamma) random numbers with parameters a
%                      and b. a must be a positive integer. The defaults
%                      are a = 2 and b = 5. Erlang random numbers are
%                      approximated as the sum of b exponential random
%                      numbers.
%
%    To generate only a matrix R of size M-by-N whose elements are from
%    any of the preceding PDFs, use the syntax
%
%       [~,R] = imnoise2(ones(M,N),type,a,b)
%
%   To generate a single random number from any of the preceding PDFs,
%   use the syntax
```

```
F = zeros(M,N);

% Insert in F the impulses and their conjugates, multiplied by the
% exponentials carrying the phase values. See Eq. (5-9).
for k = 1:K
   % Fourier transform coordinates for a given impulse.
   u1 = ucenter + C(k,1);
   v1 = vcenter + C(k,2);
   % Coordinates of the conjugate.
   u2 = ucenter - C(k,1);
   v2 = vcenter - C(k,2);
   % Form the Fourier transform.
   F(u1,v1) = 1i*M*N*(A(k)/2) * exp(-1i*2*pi*(u1*B(k,1)/M ...
                 + v1*B(k,2)/N));
   F(u2,v2) = -1i*M*N*(A(k)/2) * exp(1i*2*pi*(u2*B(k,1)/M ...
                + v2*B(k,2)/N));
end

% Compute the spectrum and spatial sinusoidal pattern.
S = abs(F);
r = real(ifft2(ifftshift(F)));
```

EXAMPLE 5.3: Using function `imnoise3`.

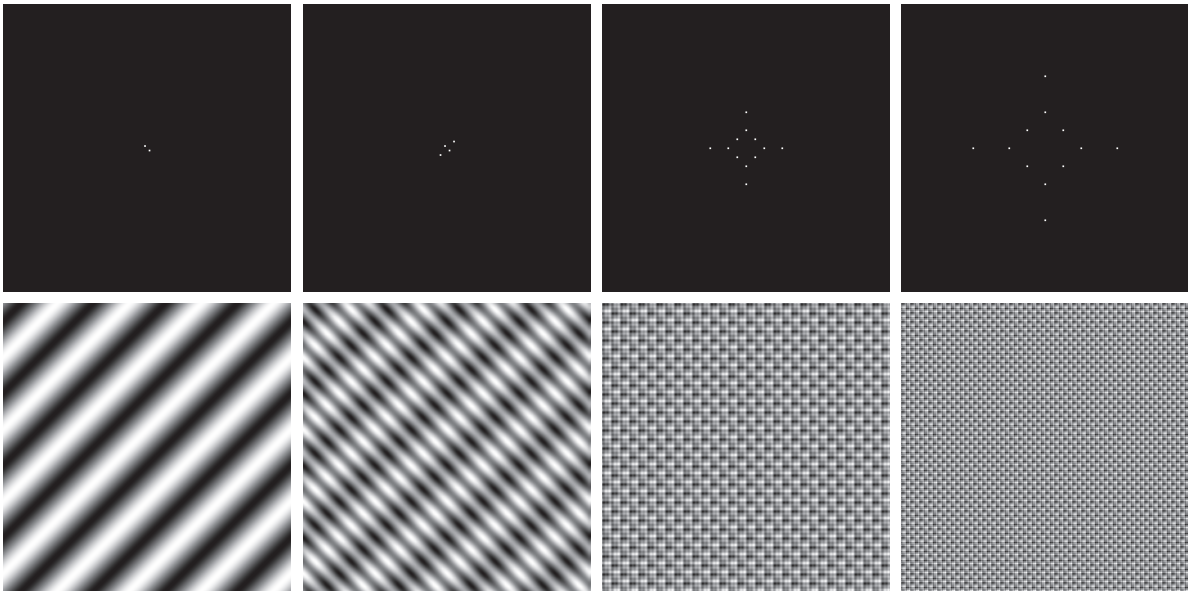
Figures 5.3(a) and (e) show the spectrum and spatial sine noise pattern generated using the following commands:

```
>> M = 512;
>> N = 512;
>> Ca = [4 4];
>> [ra,~,Sa] = imnoise3(M,N,Ca);
>> ra = mat2gray(ra); % Scale to the [0,1] range.
>> Sa = imdilate(Sa,ones(3)); % Dilate the single impulse dots (see Chapter 10 regarding dilation).
>> imshow(Sa) % Fig. 5.3(a).
>> figure, imshow(ra) % Fig. 5.3(e).
```

As mentioned in function `imnoise3`, the (u,v) coordinates of the impulses are specified with respect to the center of the frequency rectangle (see Section 4.2 for more details about the coordinates of this center point). As you can see, a pair of conjugate impulses in the frequency domain generate a pure sinusoidal function in the spatial domain. In Fig. 5.3(b) we added a second pair, orthogonal to the first, further from the origin, and with a higher value of amplitude:

```
>> Cb = [4 4;-12 12];
>> Ab = [1,1.5];
>> [rb,~,Sb] = imnoise3(M,N,Cb,Ab);
>> rb = mat2gray(rb); % Scale to the [0,1] range.
>> Sb = imdilate(Sb,ones(3)); % Dilate the single impulse dots (see Chapter 10 regarding dilation).
>> imshow(Sb) % Fig. 5.3(b).
>> figure, imshow(rb) % Fig. 5.3(f).
```

As Fig. 5.3(f) shows, we now have two sine waves. The frequency of the second sine wave is three times



a b c d
e f g h

FIGURE 5.3 Top row: Various impulse arrangements in the frequency domain. Bottom row: Corresponding sinusoidal spatial patterns. We dilated the sizes of the single dots in the top row to make them easier to see.

higher than the frequency of the first and the directions of the waves differ by 90°, facts that we could have ascertained by looking at the arrangement of the two impulse pairs in Fig. 5.3(b). We used slightly higher amplification in the second sine wave to make the figure clearer.

Similarly, the impulse pairs we used for Figs. 5.3(c) and (d) were

```
>> Cc = [0 32; 0 64; 16 16; 32 0; 64 0; -16 16];
>> Cd = [0 64; 0 128; 32 32; 64 0; 128 0; -32 32];
```

These produced eight superimposed sine waves that give the appearance of a texture pattern. The frequency of the second set is double that of the first, so the second spatial pattern is "tighter" than the first. We used the default amplitude and phases in both cases. Specifying a different phase would simply shift the sine waves with respect to the origin. As an exercise, you should experiment with function `imnoise3` to gain a deeper understanding of the relationship between impulses in the frequency domain and the spatial patterns they generate.

ESTIMATING NOISE PARAMETERS

The parameters of periodic noise typically are estimated by analyzing the Fourier spectrum. Periodic noise produces frequency spikes that often can be detected

```
>> figure, imshow(fsmin) % Fig. 5.5(f).
```

Other solutions using `spfilt` are implemented in a similar manner.

ADAPTIVE SPATIAL DENOISING FILTERS

The filters discussed in the previous section are applied to an image independently of how image characteristics vary from one location to another. In some applications, results can be improved by using filters capable of adapting their behavior based on the characteristics of the image in the region being filtered. As an illustration of how to implement adaptive spatial filters in MATLAB, we consider in this section an *adaptive median filter*. As before, S_{xy} denotes a neighborhood centered at location (x, y) in the image being processed. The algorithm, due to Eng and Ma [2001] and explained in more detail in Gonzalez and Woods [2018], is as follows. Let

$$\begin{aligned} z_{\min} &= \text{minimum intensity value in } S_{xy} \\ z_{\max} &= \text{maximum intensity value in } S_{xy} \\ z_{\text{med}} &= \text{median of the intensity values in } S_{xy} \\ z_{xy} &= \text{intensity value at coordinates } (x, y) \\ S_{\max} &= \text{maximum allowed size of } S_{xy} \end{aligned}$$

The adaptive median filtering algorithm uses two processing levels at each point (x, y), denoted level A and level B:

level A: If $z_{\min} < z_{\text{med}} < z_{\max}$, go to level B
Else, increase the size of S_{xy}
If $S_{xy} \leq S_{\max}$, repeat level A
Else, output z_{med}

level B: If $z_{\min} < z_{\text{med}} < z_{\max}$, output z_{xy}
Else, output z_{med}

where S_{xy} and $S_{\max}$ are odd, positive integers greater than 1. Another option in the last step of level A is to output z_{xy} instead of the median value z_{med}. This produces a slightly less blurred result, but can fail to detect salt (pepper) noise embedded in a constant background having the same value as pepper (salt) noise.

A custom function for adaptive median filtering that we call `adpmedian` is included in your Support Package. It has the syntax


```
f = adpmedian(g,Smax)
```

where `g` is the image to be filtered and, as defined above, `Smax` is an odd integer greater than 1 that specifies the maximum allowed (square) size of the adaptive filter window.

EXAMPLE 5.6: Adaptive median filtering.

Figure 5.6(a) shows the circuit board image corrupted by salt and pepper noise, both with probability 0.25:

```
>> f = imread('circuitboard.tif');
>> g = imnoise(f,'salt & pepper',0.25);
>> figure, imshow(g)
```

Figure 5.6(b) is the result of "standard" median filtering:

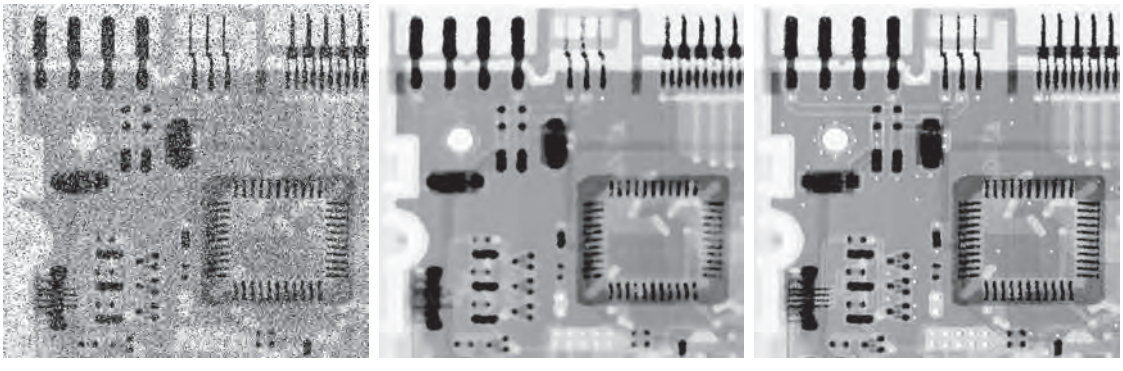
See Section 3.5 regarding function `medfilt2`.

```
>> f1 = medfilt2(g,[7 7],'symmetric');
```

This image is reasonably free of noise, but it is blurred and distorted as illustrated, for example, by the connector fingers in the top middle of the image. On the other hand, the command

```
>> f2 = adpmedian(g,7);
```

yielded the image in Fig. 5.6(c), which is also reasonably free of noise, but is less distorted and less blurred than Fig. 5.6(b). For instance, the connector fingers mentioned above are less distorted and the feed-through holes (small white circles) are much sharper and brighter in Fig. 5.6(c) .



a b c

FIGURE 5.6 (a) Image corrupted by salt-and-pepper noise with density 0.25. (b) Result obtained using a median filter of size 7×7. (c) Result obtained using adaptive median filtering with $S_{max} = 7$.

where `noise` is a random noise image of the same size as `g`, generated using one of the methods we discussed in Section 5.2.

EXAMPLE 5.7: Modeling a blurred noisy image.

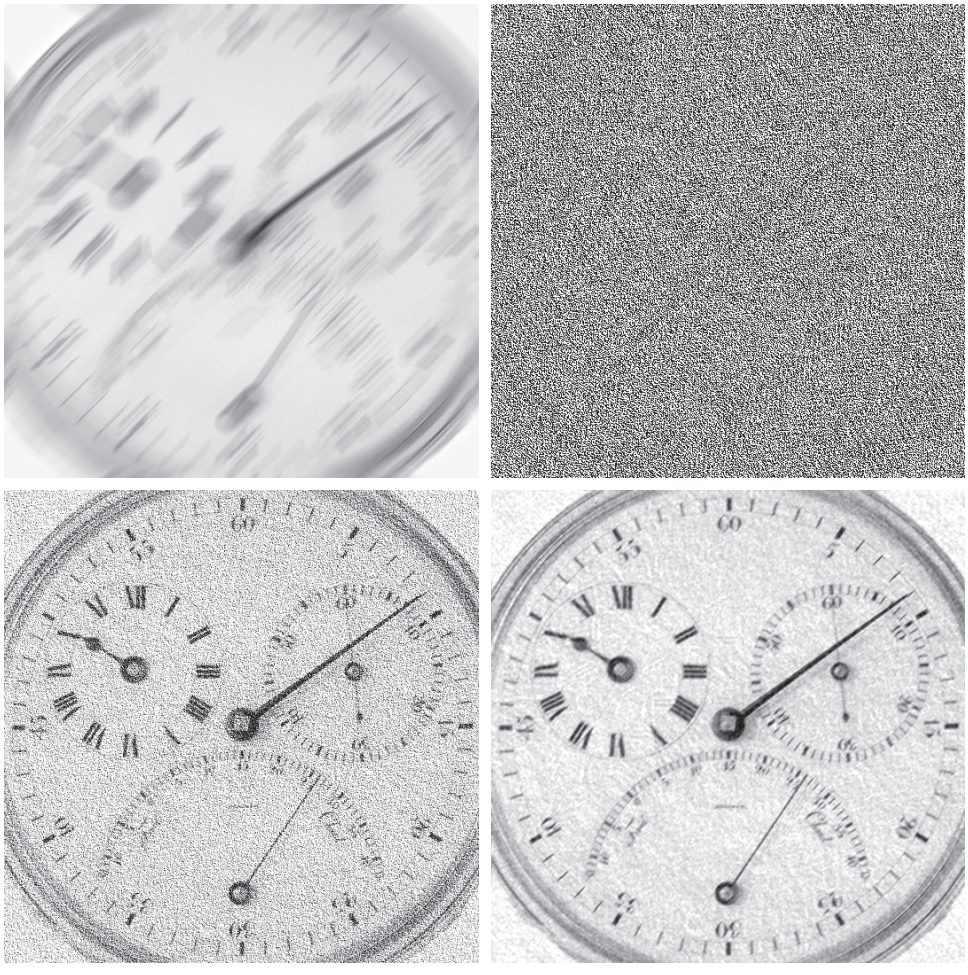
Figure 5.7(a) shows a grayscale image of size 534×535 pixels that we will first use to model a degradation caused by blurring and additive noise and then use in several of the following sections to illustrate various image restoration techniques.

Figures 5.7(b) resulted from using the following commands:

```
>> f = im2double(imread('chronometer-small.tif'));
>> figure, imshow(f) % Fig. 5.7(a).
>> % Generate an aggressive motion-blurring PSF.
```

a b
c d

FIGURE 5.7
(a) Original image of size 534×535 pixels. (b) Image blurred with a PSF obtained using `fspecial` with `len = 50` pixels and `theta = 45` degrees. (c) Gaussian noise pattern, scaled for display. (d) Blurred, noisy image.



a b
c d

FIGURE 5.8
(a) Blurred, noisy image.
(b) Result of inverse filtering.
(c) Result of Wiener filtering using a constant ratio.
(d) Result of Wiener filtering using autocorrelation functions.

is considerably better. The final result is not perfect, but considering the extensive degradation of the original image, the restored image contains all the principal details that were lost due to degradation.

5.7 CONSTRAINED LEAST SQUARES (REGULARIZED) FILTERING

Another approach to linear restoration is *constrained least squares filtering*, called *regularized filtering* in the Toolbox documentation. We know from Section 3.4 that the 2-D *discrete convolution* of two functions f and h is

Remember, convolution is commutative, so the order of f and h does not matter. The form shown here is better suited for a matrix formulation.

$$(f \star h)(x,y) = \sum_{m=0}^{M-1} \sum_{n=0}^{N-1} f(m,n)h(x-m,y-n) \tag{5-22}$$

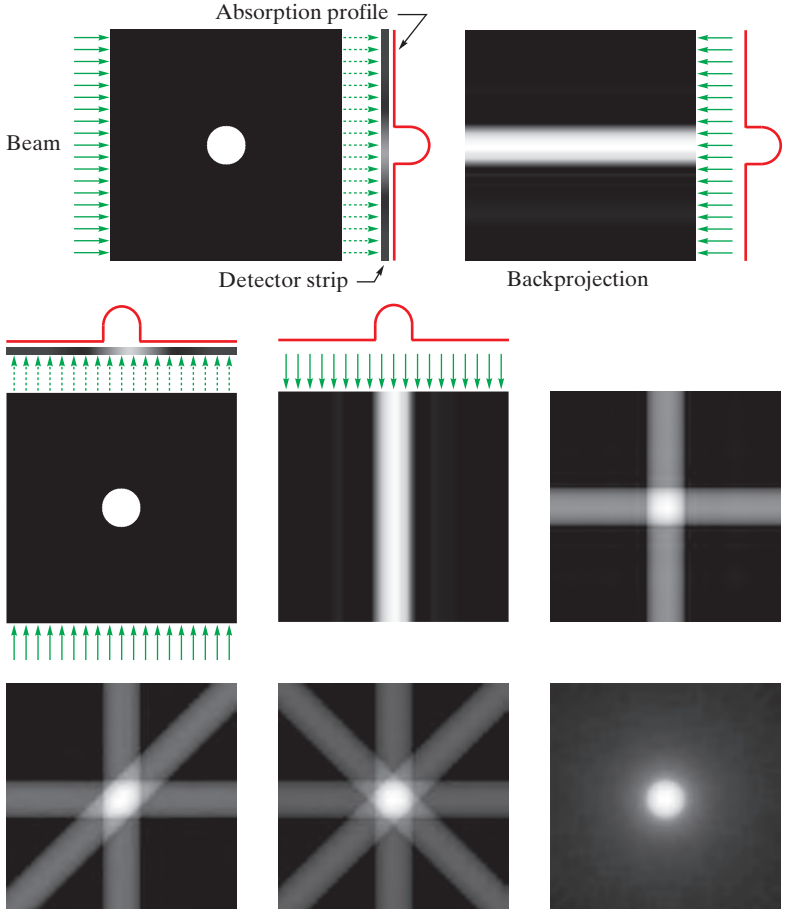
5.10 IMAGE RECONSTRUCTION FROM PROJECTIONS

Computerized Axial Tomography (CAT) is also used to refer to CT imaging.

Thus far in this chapter we have dealt with the problem of image restoration. In this section we focus attention on the topic of reconstructing an image from a series of 1-D projections. This area, called *computed tomography* (CT), is one of the most successful commercial applications of image processing, particularly in medicine.

BACKGROUND

The foundation of image reconstruction from projections is straightforward and can be explained intuitively. Consider the image in Fig. 5.14(a). To give physical meaning to the following discussion, assume that this image represents a "slice" through a section of a human body that contains a tumor (bright, circular region) embedded in a homogeneous area of tissue (black background). Such a slice might be obtained, for example, by passing a thin, flat beam of X-rays perpendicular to the body and recording at the opposite end measurements proportional to the absorption of the



a b
c d e
f g h

FIGURE 5.14
(a) Flat region, parallel beam detector strip and absorption profile. (b) Backprojection of absorption profile. (c) Beam and detector strip rotated 90°. (d) Backprojection of absorption profile. (e) Sum of (b) and (d). (f) Result of adding another backprojection at 45°. (g) Result of adding yet another backprojection at 135°. (h) Result of adding 32 backprojections 5.625° apart (observe that the result is blurred).

Summary

The material in this chapter is a good overview of how MATLAB and Image Processing Toolbox functions can be used for image restoration and how they can be used as the basis for generating models that help explain the degradation to which an image has been subjected. The capabilities of the Toolbox for noise generation were enhanced significantly by the development in this chapter of the functions `imnoise2` and `imnoise3`. Similarly, the spatial filters available in function `spfilt`, especially the nonlinear filters, are also a significant extension of Toolbox capabilities in this area. These functions are perfect examples of how relatively simple it is to incorporate MATLAB and Toolbox functions into new code to create applications that enhance the capabilities of an already large set of existing tools. Our treatment of image reconstruction from projections covers the principal functions available in the Toolbox for dealing with projection data.

MATLAB Projects

*Solutions to the projects marked with an asterisk * are in the DIPUM3E Student Support Package (consult the book web site). All your code must be documented so that typing `help` at the prompt, followed by the script or function name, gives enough detail for a user to be able to run it. Test the functionality of all your code thoroughly.*

5.1 Read the image `testpattern512.tif` and do the following:

(a)* Fix the mean at 0.25 and add four levels of Gaussian noise to the image by varying the standard deviation. The four levels should be such that the noise appears: (1) *mild* (you can tell the noise is there, but it is barely perceivable); (2) *intermediate* (the noise is definitely present, but all the image features are still clearly visible); (3) *heavy* (the noise is objectionable, causing some of the image features to be obscured by the noise); and (4) *extra heavy* (the noise dominates the image; most of the smaller and light features in the image are obscured by noise). For comparisons of your results to be meaningful, you should scale the image to the full range [0,1], using, for example, the custom function `intensityScaling`. Show all four results and list the values of standard deviation you used. Explain why image details begin to disappear as the noise level increases.

(b)* In (a), as the noise increases, the scaled images get darker. Explain the reason why this is so.

(c) Repeat the four levels of noise outlined in (a) using uniform noise [the parameters to specify are a and b in Eq. (5-13)]. Try to make your images appear as close as possible to their Gaussian counterparts in (a).

(d) The images in (c) will have higher contrast than their Gaussian counterparts in (a). Explain why.

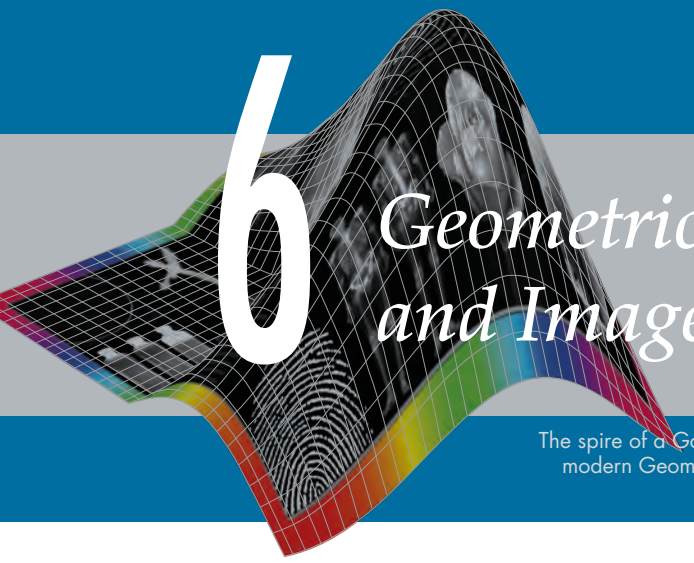
5.2 Read the image `sombrero-galaxy-noisy.tif`. You are told the noise that corrupted the image had zero mean.

(a)* Estimate the value of the noise standard deviation in the range [0,255]. (*Hint:* Consider using a region of interest that is in a mid-gray region with a low intensity gradient across it.)

(b) Determine which of the PDFs in Table 5.1 is closer to the PDF of our noisy image. Explain how you arrived at your conclusion.

5.3 In the following denoising experiments, use any spatial filtering method of your choice, *except* median filtering. The overall intensity of the restored image should be visually close to the original image, `polymercell.tif`, and the size of the filtering neighborhood should be as small as possible, but be capable of eliminating all the salt and/or pepper noise.

(a)* Read the image `polymercell-pepper.tif` and restore it using spatial filtering.



6 Geometric Transformations and Image Registration

The spire of a Gothic cathedral and the importance of the unbounded straight line in modern Geometry are both emblematic of the transformation of the modern world.
Alfred North Whitehead

Geometric transformations modify the spatial relationships between pixels in an image. The image can be made larger or smaller. It can be rotated, shifted, or otherwise stretched in a variety of ways. Geometric transformations are used to create thumbnail views, adapt digital video from one playback resolution to another, correct distortions caused by viewing geometry, and align multiple images of the same scene or object.

In this chapter we explore the central concepts behind the geometric transformation of images, including geometric coordinate mappings, image interpolation, and inverse mappings. We show how to apply these techniques using Image Processing Toolbox functions and we explain underlying Toolbox conventions. We conclude the chapter with a discussion of image registration, the process of aligning multiple images of the same scene or object for the purpose of visualization or quantitative comparisons.

Functions Developed in this Chapter:

- `geotrans` provides an easy way to make common transformations like rotation, scale, translation, reflection, and shear.
- `imwarp2` applies a 2-D geometric transformation to an image using a fixed output location.

6.1 TRANSFORMING POINTS

Understanding the geometric transformation of images begins naturally with a discussion of the transformation of points. Suppose that (w, z) and (x, y) are two spatial coordinate systems called the *input space* and *output space*, respectively. A geometric coordinate transformation can be defined that maps input space points to output space points:

$$(x, y) = T\{(w, z)\} \tag{6-1}$$

where $T\{\bullet\}$ is called a *forward transformation* or *forward mapping*. If the forward transformation has an inverse, then that inverse maps output space points to input space points:

$$(w, z) = T^{-1}\{(x, y)\} \tag{6-2}$$

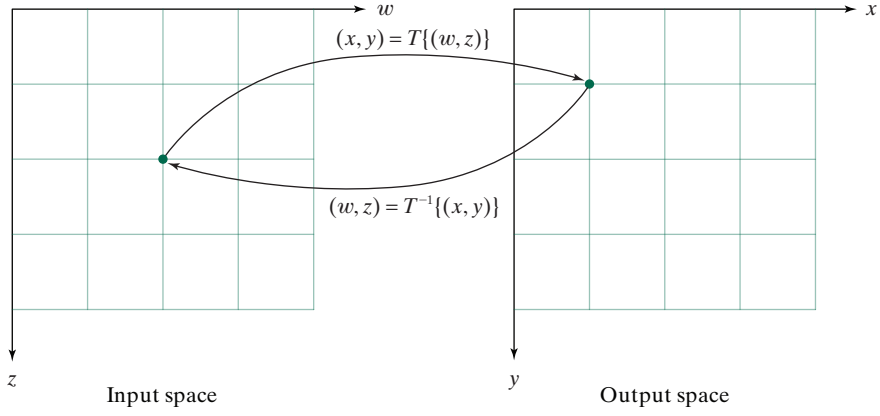
where $T^{-1}\{\bullet\}$ is called the *inverse transformation* or *inverse mapping*. Figure 6.1 illustrates the forward and inverse transformation for this simple example:

$$(x, y) = T\{(w, z)\} = (w/2\,, z/2) \tag{6-3}$$

$$(w, z) = T^{-1}\{(x, y)\} = (2x, 2y) \tag{6-4}$$

Geometric transformations of images are defined in terms of point transformations. Let $f(w, z)$ denote an image in the input space. We can define a transformed image in the output space, $g(x, y)$, in terms of $f(w, z)$ and $T^{-1}\{\bullet\}$ as follows:

$$g(x, y) = f(T^{-1}\{(x, y)\}) \tag{6-5}$$



a b

FIGURE 6.1 Forward and inverse transformation of a point for $T\{(w, z)\} = (w/2\,, z/2)$.

The input argument, `T`, is the 3×3 projective transformation matrix in Eq. (6-20). To transform points with a projective transformation, we use `tform` with the same two functions, `transformPointsForward` and `transformPointsInverse`, that we used for affine transformation objects. For example,

```
>> T = [ 1.8  -0.8   0.2
        -0.1   1.7  -0.2
         0.2   0.4   1.0];
>> tform = projective2d(T)

tform =

  projective2d with properties:

                T: [3×3 double]
    Dimensionality: 2

>> WZ = [-1 -1; 1 -1; 1 1; -1 1];
>> XY = transformPointsForward(tform, WZ)

XY =

   -1.5000   -0.5000
    1.5000   -1.5000
    1.9000    1.3000
   -2.8333    4.833
```

Figure 6.4 illustrates some of the geometric properties of projective transformations. The figure shows a projective transformation applied to a grid of points. As illustrated in the figure, sets of parallel lines transform to output-space lines that intersect at locations called *vanishing points*. All vanishing points for a projective transformation lie on a single line called the *horizon line*. Only input-space lines parallel to the horizon line remain parallel when transformed. All other sets of parallel lines transform to lines that intersect at a vanishing point on the horizon line.

CREATING COMMON TRANSFORMATIONS

The custom function `geotrans`, listed below, provides an easy way to make common transformations such as rotation, scale, translation, reflection, and shear. For example, `geotrans('scale',2,4)` creates a transformation that scales horizontally by 2 and vertically 4, while `geotrans('rotate',45)` creates a transformation that rotates counterclockwise by 45 degrees. The function can also compose affine and projective transformations together into a single projective transformation.


```
function tform = geotrans(varargin)
%GEOTRANS  Make affine and projective geometric transformations.
%   TFORM = GEOTRANS(TYPE,P1,P2,___) makes a geometric transformation
%   with the specified type and parameters.
%
%   Valid values for TYPE and P1, P2, ..., are:
%
```

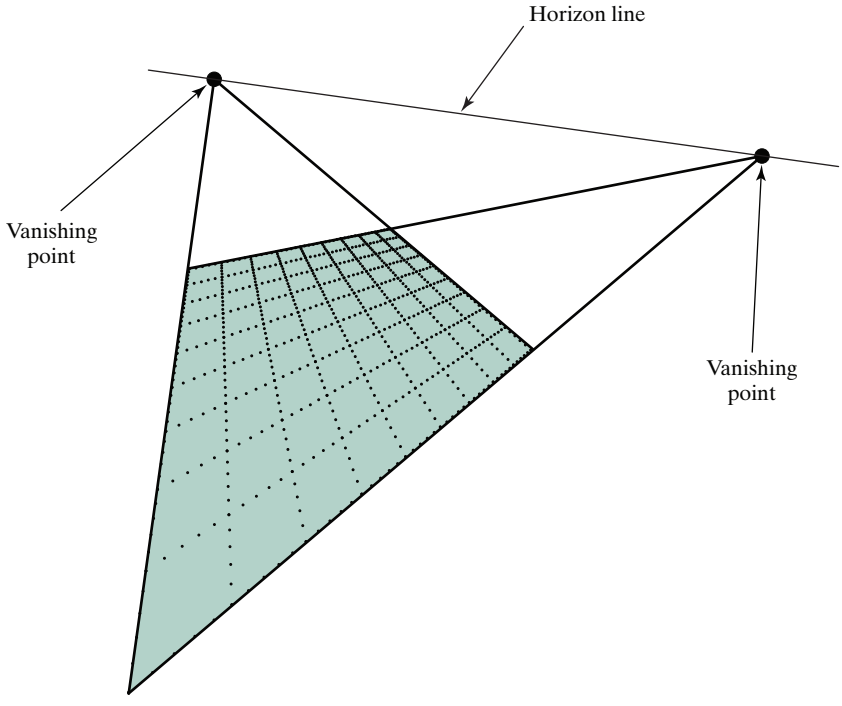


FIGURE 6.4
Vanishing points and the horizon line for a projective transformation.

```
%   'scale'     Scale. If only P1 is provided, it is the scale
%               factor in both directions. If P1 and P2 are provided,
%               then P1 is the horizontal scale factor and P2 is the
%               vertical scale factor.
%
%   'rotate'    Rotation. P1 is the rotation angle, measured in
%               degrees counterclockwise from the positive horizontal
%               axis.
%
%   'translate' Translation. P1 is the horizontal translation and P2
%               is the vertical translation.
%
%   'h-reflect' Horizontal reflection.
%
%   'v-reflect' Vertical reflection.
%
%   'h-shear'   Horizontal shear. P1 is the sheer angle in degrees,
%               measured counterclockwise from the downward-pointing
%               y-axis.
%
%   'v-shear'   Vertical shear. P1 is the sheer angle in degrees,
%               measured clockwise from the horizontal axis.
%
%
%   'compose'   Composition of transformations. P1, P2, ..., are
%               affine2d and projective2d transformations.
```

A *feature* in the present context is any portion of an image that can potentially be identified and located in multiple images. Features can be points, lines, or corners, for example. (See Chapter 13 regarding image features.) Once selected, features have to be matched. That is, for a feature in one image, one must determine the corresponding feature in another image or sequence of images. Feature-based registration methods can be manual or automatic, depending on whether feature detection and matching is human-assisted or performed using an automatic algorithm.

From the set of matched-feature pairs, a geometric transformation function is inferred that maps features in one image onto the locations of the matching features in another. Usually a particular parametric transformation model is chosen based on the particular image capture geometry. For example, if two images are taken with the same viewing angle but from a different position (possibly including a rotation about the optical axis) and if the scene objects are far enough from the camera to minimize perspective effects, then an affine transformation can be used (Brown [1992]).

MANUAL FEATURE DETECTION AND MATCHING

The Image Processing Toolbox provides an app called the **Control Point Selection Tool** for manually selecting and matching corresponding features, also called *control points*, in a pair of images to be registered. The tool is launched by passing the filenames of the images to be aligned as input arguments to the function `cpselect`. For example:

`cpselect`

```
>> cpselect('vector-gis-data.tif','aerial-photo-cropped.tif')
```

Alternatively, the images can be read into MATLAB variables first and then passed to `cpselect`:

```
>> f = imread('vector-gis-data.tif');
>> g = imread('aerial-photo-cropped.tif');
>> cpselect(f,g)
```

The tool helps navigate (zoom, pan, and scroll) in large images. Features can be selected and paired with each other by clicking on the images using the mouse.

Figure 6.22 shows the **Control Point Selection Tool** in action. Figure 6.22(a) is a binary image showing road, pond, stream, and power-line data. Figure 6.22(b) shows an aerial photograph covering the same region. The white rectangle in Fig. 6.22(b) shows the approximate location of the data in Fig. 6.22(a). Figure 6.22(c) is a screen shot of the **Control Point Selection Tool** showing six pairs of corresponding features selected at the intersections of several roadways.

USING fitgeotrans TO OBTAIN TRANSFORMATION PARAMETERS

Once feature pairs have been identified and matched, the next step in the registration process is to determine the geometric transformation function. The usual procedure is to choose a particular transformation model and then estimate the

(for example, 'affine') specifying the desired type of transformation. The output argument is a transformation object whose type depends on the specific transformation type chosen.

EXAMPLE 6.8: Registering images using manually selected and matched features.

In this example we use fitgeotrans, imwarp, and imshowpair to register and then visualize the alignment of the images in Fig. 6.22(a) and (b). The first two feature-based registration steps, detecting and matching corresponding features, were performed manually using the **Control Point Selection Tool** (cpselect) and saved to a MAT-file in a struct called cpstruct. The following code reads the image data and loads the previously saved feature matching results:

```
>> f_fixed = imread('aerial-photo.tif');
>> f_moving = imread('vector-gis-data.tif');
>> s = load('cpselect-results');
>> cpstruct = s.cpstruct;
```

To perform the third step, inferring the geometric transformation, we use fitgeotrans to obtain an affine transformation that aligns image f_moving with reference image f_fixed:

```
>> tform = fitgeotrans(cpstruct.inputPoints,cpstruct.basePoints,'affine');
```

Finally, we call imwarp to register the moving image with the fixed image. We use the optional second output argument to imwarp to capture the spatial reference information for aligned image f_reg. We also construct the default spatial reference information for the fixed image:

```
>> [f_reg,f_reg_ref] = imwarp(f_moving,tform);
>> f_fixed_ref = imref2d(size(f_fixed));
```

To inspect the result, we pass the fixed image, the aligned image, and the spatial reference information to imshowpair and we use the axis function to zoom into the area of interest:

```
>> imshowpair(f_fixed,f_fixed_ref,f_reg,f_reg_ref)
>> axis([1600 2650 1760 2700])
```

Figure 6.23 shows that the GIS and aerial images were registered successfully.

AUTOMATIC FEATURE DETECTION AND MATCHING

See Section 13.6 for additional details on feature generation and matching in the sense discussed here.

Advances in computer vision have helped popularize a number of methods for automatic feature detection and matching for growing applications such as automated driving. Here we will explore in more detail a feature detection method known as *SURF* (*Speeded-Up Robust Features*), as well as related computational and visualization functions in the Computer Vision Toolbox.

SURF is really two associated methods—one for detecting feature points and another for computing descriptors of the intensity variation in the neighborhood of feature points (Bay [2008]). Feature point detection is based on computing the Hessian matrix at every image location and at multiple scales. The *Hessian matrix*

FIGURE 6.23 Using the **Control Point Selection Tool** and functions `fitgeotrans` and `imwarp` to align a GIS image with an aerial photograph. The GIS image is characterized by roadways and bodies of water.

provides a measure of the local curvature of a function (see Gonzalez and Woods [2018]). Locations with a high Hessian matrix determinant are considered to be candidate feature points. At a given scale, σ, and location, (x, y), the Hessian matrix is given by:

$$\mathcal{H}(x,y,\sigma) = \begin{bmatrix} L_{xx}(x,y,\sigma) & L_{xy}(x,y,\sigma) \\ L_{xx}(x,y,\sigma) & L_{yy}(x,y,\sigma) \end{bmatrix} \tag{6-30}$$

$L_{xx}(x,y,\sigma)$ is the convolution at location (x, y) of image f and the second-order derivative of a Gaussian function with scale σ.

The SURF feature detection method can detect points at different scales (meaning high local curvature in regions of varying size) by computing $|\mathcal{H}|$ for several different values of σ. SURF achieves fast computational speed by approximating the computation of $\mathcal{H}$ using something called an *integral image* (see Gonzalez and Woods [2018]). Each pixel of an integral image, $f_{\sum}(x,y)$, is the sum of all the pixels of f in the rectangular region between the image origin and (x, y):

$$f_{\sum}(x,y) = \sum_{u=0}^{x}\sum_{v=0}^{y} f(u,v) \tag{6-31}$$

Integral images can be used to speed a variety of computations because of one key property: we can compute a sum of intensities of f over any axis-aligned rectangular region bounded at the origin by adding or subtracting just three values from the integral image (Viola [2001]).

For each feature point, SURF computes a dominant orientation and 64 descriptor values. The descriptor values are based on Haar wavelet responses in a $20\sigma \times 20\sigma$

region around the feature point. The dominant orientation is also computed from Haar wavelet responses, but from a $6\sigma \times 6\sigma$ region. (See Chapter 8 for a detailed discussion of wavelet analysis.)

The Computer Vision Toolbox has several functions related to SURF-based feature detection, description, matching, and geometric transformations. The function `detectSURFFeatures` takes a grayscale image as input and returns a set of feature points and the corresponding scale and dominant orientation values. The calling syntax is:

`detectSURFFe...`

```
points = detectSURFFeatures(f)
```

After the initial set of candidate points is computed using this function, we use the `extractFeatures` function to extract feature descriptors for each feature point that is sufficiently far from an image boundary for the computation to be reliable. The feature points (`valid_points`) and their descriptors (`features`) are both returned as output arguments:

The Computer Vision Toolbox uses the terms features, descriptors, and feature descriptors interchangeably.

`extractFeatures`

```
[features,valid_points] = extractFeatures(points)
```

After computing feature points and descriptors for a pair of images, pairs of likely corresponding features are determined using the `matchFeatures` function, whose basic syntax is

`matchFeatures`

```
indexPairs = matchFeatures(features1,features2)
```

The output, `indexPairs`, is a $P \times 2$ matrix indicating the corresponding feature pairs from the two sets. If `[a b]` is a row of `indexPairs`, that means that `valid_points1(a)`, from the first image, is a likely match for `valid_points2(b)`, from the second image. Finally, given pairs of likely feature point correspondences, the function `estimateGeometricTransform` can compute a transformation that will register one image to the other. The syntax is:

`estimateGeom...`

```
tform = estimateGeometricTransform(matched_points1,...
                                   matched_points2,type)
```

where `type` is the geometric transformation type, such as `'affine'`.

EXAMPLE 6.9: Panorama stitching using SURF feature detection and matching.

This example illustrates the use of SURF-based feature detection and matching to stitch together two images to form a *panorama*. We will work with the two images shown in Figs. 6.24(a) and (b).

```
>> f1_rgb = imread('ah-quad-b.jpg');
>> f2_rgb = imread('ah-quad-c.jpg');
>> imshow(f1_rgb)
>> figure, imshow(f2_rgb)
```

Next, we convert the images to grayscale and call `detectSURFFeatures`:

a b
c d
e

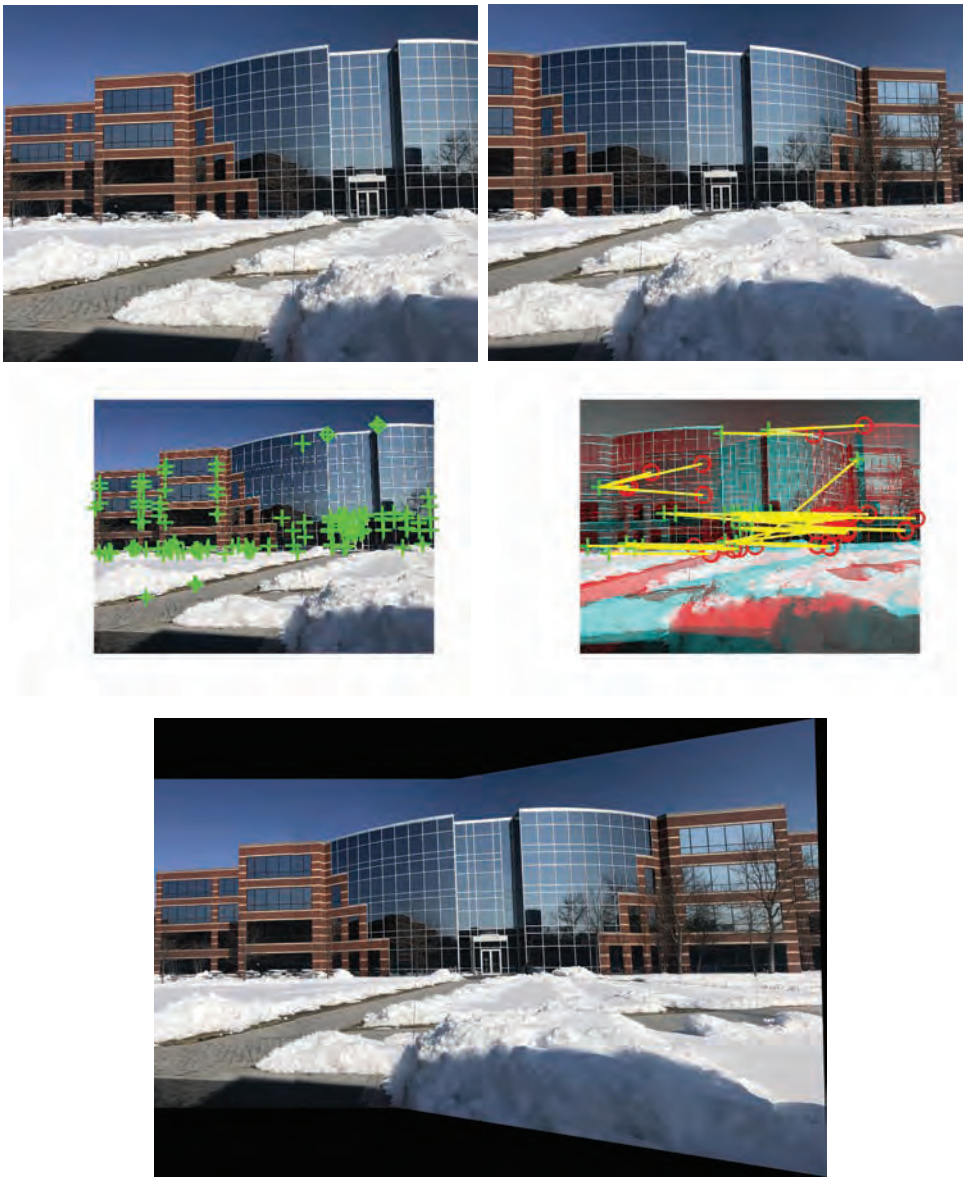
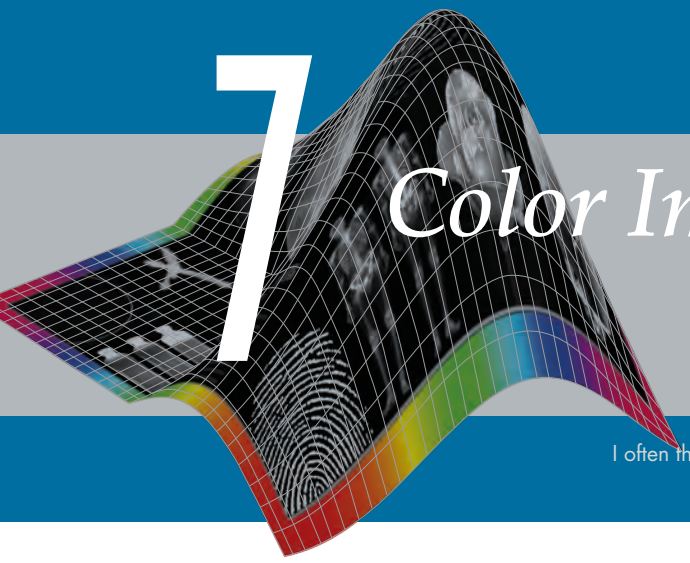


FIGURE 6.24
Automatic feature-based registration using SURF features. (a) and (b) Two different views of the same structure. (c) Strongest feature points detected in the left image. (d) Forty potential feature point matches. (e) Registration result and image blending used to form the panorama image.

```
>> f1_gray = rgb2gray(f1_rgb);
>> f2_gray = rgb2gray(f2_rgb);
>> points1 = detectSURFFeatures(f1_gray);
>> points2 = detectSURFFeatures(f2_gray);
```

The function `extractFeatures` takes our candidate feature points, selects the ones that are not too close to an image boundary, and computes feature descriptors for each one:



7 Color Image Processing

I often think that the night is more alive and more richly colored than the day.
Vincent Van Gogh

The use of color in image processing is motivated by two principal factors. First, color is a powerful descriptor that often simplifies object identification and extraction from a scene. Second, humans can discern over a million shades of color, compared to only a few dozen shades of gray. The latter factor is particularly important in manual image analysis. In this chapter we discuss fundamentals of color image processing using the Image Processing Toolbox and extend some of its functionality by developing additional color generation and transformation functions. The discussion in this chapter assumes familiarity on the part of the reader with the principles and terminology of the physics of color at an introductory level.

Functions Developed in this Chapter:

- `colormatchingFunctions` returns a table defining the CIE 1931 Standard Observer.
- `lambda2xyz` converts spectral wavelengths to tristimulus values.
- `rspd2xyz` converts a relative power spectral density function to CIE 1931 tristimulus values.
- `chromaticityDiagram` plots a chromaticity diagram.
- `xyz2xyy` converts XYZ tristimulus values to chromaticity coordinates.
- `xyy2xyz` converts chromaticity coordinates to XYZ tristimulus values.
- `colorSwatches` displays a set of colors as square regions.
- `rgb2hsi` converts an RGB image to HSI.
- `hsi2rgb` converts an HSI image to RGB.
- `rgbcube` displays an RGB cube.
- `ice` implements an *Interactive Color Editor* for RGB and other color model images.
- `colorgrad` computes color gradients of RGB images.
- `colorseg` segments an RGB image.
- `spectrumBar` adds a visible light spectrum bar to a line plot.
- `spectrumColors` generates RGB colors in the visible light spectrum.

7.1 COLOR FUNDAMENTALS

The color we perceive an object to be is the result of complex interactions between the light of an illuminating source, what happens to that light when it hits and is reflected from the object, and the human visual system. The illuminating source, or *illuminant*, might be the noon sun, a cloudy sky, or an LED light bulb. The object may reflect, absorb, or transmit some of the light. When reflected light from the object arrives at the eyes, it stimulates light-sensitive retinal cells that then transmit signals to the visual processing centers of the brain. To at least some degree, these interactions can be modeled and analyzed to understand and predict the human response to color.

LIGHT

Sir Isaac Newton was the first to observe (in 1666) that when a beam of sunlight passes through a glass prism, the emerging light, rather than being white, consists instead of a continuous spectrum of colors ranging from violet at one end to red at the other. The perceived colors in this spectrum include violet, blue, green, yellow, orange, and red. As Fig. 7.1 shows, no color in the spectrum ends abruptly; rather, each color blends smoothly into the next. This *visible light spectrum* is *electromagnetic radiation* in a specific range of wavelengths: Blue and violet hues are at wavelengths below 480 nm; green hues range from 480 to 560 nm; yellow hues are between 560 and 590 nm; orange from 590 to 630; and red hues are located above 630 nm (Berns [2000]). *Ultraviolet light*, which is filtered out by some sunglasses to protect your eyes, is in the range just below 400 nm. *Infrared light*, which is detected by night-vision equipment, is in the range just above 700 nm.

You can generate Fig. 1 using the custom functions

```
[rgb,lambda] = spectrumColors
```

and `colorSwatches`. The function listing for `spectrumColors` is in your Support Package. We will discuss `colorSwatches` later, in the "Standard RGB Model" section.

Different kinds of light sources contain different mixes of visible electromagnetic radiation. The particular mix of a specific light source is characterized by the *spectral power distribution curve* of the source. Figure 7.2(a) shows the *relative* spectral power distribution curve for average midday open-air light. By convention, relative spectral power distribution curves for illuminants are normalized so that the spectral power at 560 nm is 1.0 (Berns [2000]). This *average daylight curve*, called *D65*, is a standard

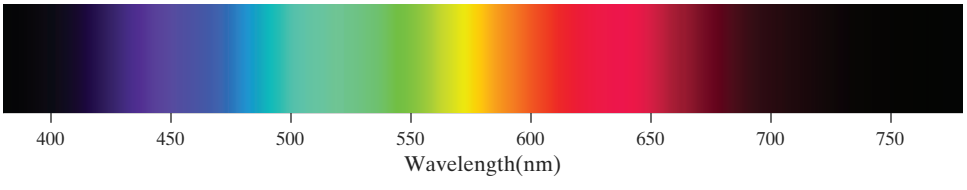


FIGURE 7.1 Visible light spectrum.

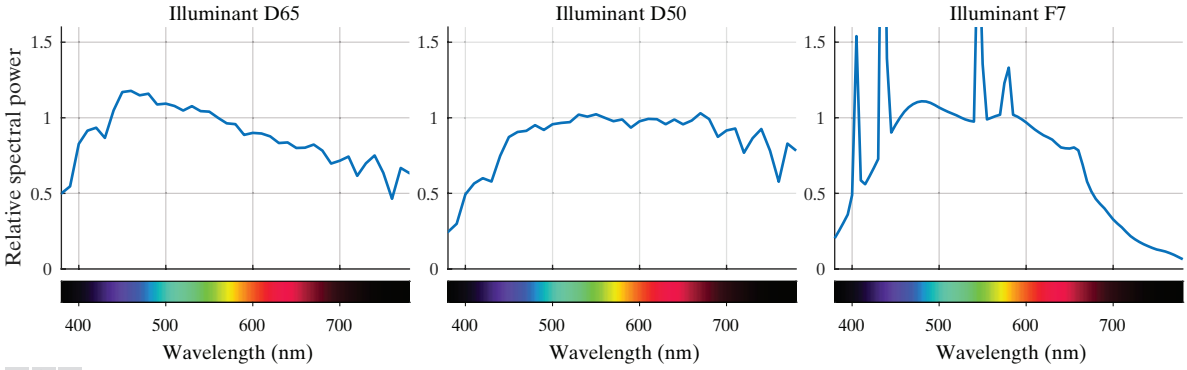


FIGURE 7.2 Relative spectral power densities for several standard illuminants. (a) D65—average mid-daylight. (b) D50—average sunrise or sunset light. (c) F7—a fluorescent light used to simulate D65.

reference curve created by the *International Commission on Illumination*, usually written as *CIE* (for the French name, Commission Internationale de l'Éclairage). Figures 7.2(b) and (c) show the curves for two other illuminants. *D50* is the average daylight for sunrise or sunset and *F7* is a fluorescent light often used to simulate D65 (CIE [2004], Berns [2000]). The plots with spectrum bars appearing in Fig. 7.2 and other figures in this chapter were created using the custom function


```
cb_out = spectrumBar(ax)
```

This function adds a visible spectrum light bar on the specified axis; the listing is in your Support Package.

OBJECT REFLECTANCE

Light that strikes an object may be reflected, absorbed, or transmitted. In general, the relative amount of light reflected by an object varies with the wavelength of the light. It is this wavelength-dependent variation, interacting with the illumination and with the human visual system, that gives rise to a specific color being perceived.

Figure 7.3(a) shows the colors of three different materials. The leftmost square is the color of a sample of azurite, a copper carbonate mineral. The center square is the color of a green roofing material made of fiberglass. And the rightmost square is manganese violet, a paint pigment. Figure 7.3(b) shows the relative reflectance curves for each of these three materials (Kokaly et al. [2017]). The azurite reflectance curve peaks at about 453 nm, which is in the blue region of the visible light spectrum. The fiberglass roofing tiles reflect the most light at about 505 nm, in the green region of the spectrum. Unlike the other two materials, the reflectance curve for the paint pigment has two distinct peaks, one in the blue part of the spectrum and the other in the red. The resulting purple hue is not one of the colors of the visible light spectrum; it is made by mixing blue and red hues.

a
b

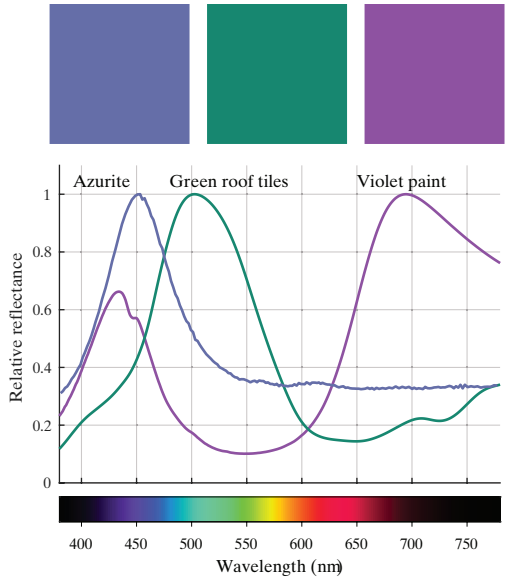


FIGURE 7.3 Colors and reflectance curves. (a) Colors of three different materials: azurite (a mineral), green fiberglass roof tiles, and manganese violet paint. (b) Relative reflectance curves of the three materials.

HOW THE EYE SENSES COLOR

When light from one or more sources is reflected from an object, arrives at the eye, and is focused by the lens onto the retina, it stimulates several types of light-sensitive receptor cells in the retina. The cells known as *rods* are active only in low-light situations and their contribution to color perception is insignificant. The cells known as *cones*, which activate at higher light levels, send color information to the brain.

There are three types of cones, distinguished primarily by how they respond to light at different wavelengths in the visible light spectrum. The cone types are labeled L, M, and S, based on whether they respond best to long, medium, or short wavelengths, respectively. Figure 7.4 shows the *relative sensitivity curves* for each of the three cone types (Stockman and Sharpe [2000]). These curves represent average responses for humans with normal color vision. Individual responses will vary and the responses of people with color-vision deficiencies can vary significantly from these curves.

Figure 7.5 illustrates how illumination, object reflectance, and cone sensitivity interact to form a color signal sent to the brain. The curve in Fig. 7.5(a) is the product of the D65 relative spectral power distribution curve in Fig. 7.2(a) with the relative reflectance curve of manganese violet in Fig. 7.3(d). This product curve represents the mix of light transmitted to the three types of retinal cones. Figures 7.5(b)–(d) show the product of the Fig. 7.5(a) curve with the relative sensitivity curves for the L, M, and S cone types. The strength of the signal that each cone type sends to the brain is then proportional to the area under the corresponding curve.

7.2 COLOR-SPACE MODELS

This section discusses several different ways to represent colors as points in a multidimensional space (typically, a three-dimensional space). Representing colors in this way, as opposed to spectral density or reflectance curves, facilitates the design and operation of most color display and sensing devices, as well as the application of image processing operations to color imagery as described later in this chapter. Of the large number of existing color-space models, we will focus our attention on those created for the following applications:

1. Perceptual color matching (the *CIE standard observer*)
2. Color displays (RGB models)
3. Color printing (CMY and CMYK models)
4. Brightness-color separation (HSV, HSI, and L*a*b* models)

THE CIE COLOR MATCHING MODEL

The CIE system of measuring and representing color is based on perceptual color matching experiments performed independently by Wright [1929], [1930] and by Guild and Petavel [1931]. In these experiments, three independent light sources (called *primaries*) were added together in varying ratios until the result matched a test stimulus perceptually.

Figure 7.6 illustrates the experiment. A test stimulus, with a known spectral power density curve $P(\lambda)$, is projected onto Area 1. Area 2 displays the combined projections of the three *monochromatic primaries*, $\hat{R}$, $\hat{G}$, and $\hat{B}$. The wavelengths of these primaries are 700 nm (red), 546.1 nm (green), and 435.8 nm (blue). The

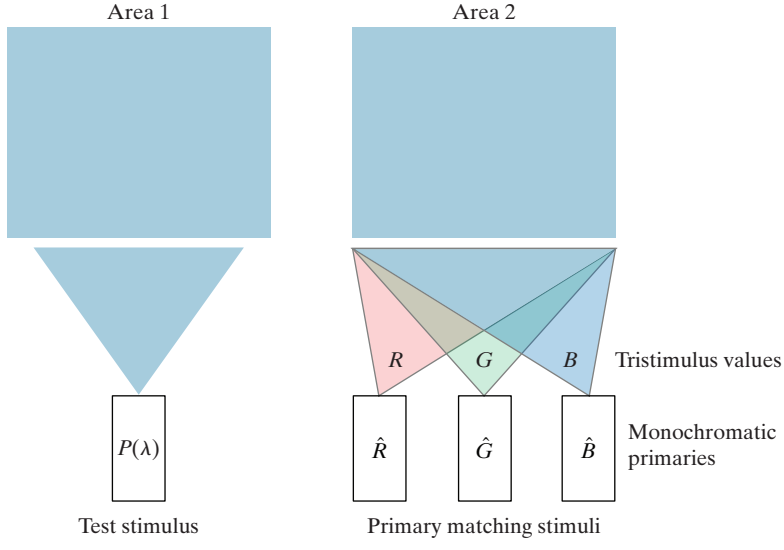


FIGURE 7.6 CIE color matching experiment. The objective is for an observer to adjust the monochromatic primaries until Area 2 is judged by the observer to be visually the same as Area 1.

For example,

```
>> XYZ = [0.9505 1.0000 1.0888];
>> xyy = xyz2xyy(XYZ)

xyy =
    0.3127    0.3290    1.0000

>> xyy2xyz(xyy)
ans =
    0.9505    1.0000    1.0888
```

THE STANDARD RGB MODEL

In the RGB color model, a color is described by three numbers, R, G, and B. These values indicate the intensities of red, green, and blue light sources, respectively. The light sources are called *RGB primaries* and it is a convention in color-space formulas and computations to express them using values in the range $[0, 1]$. When all three primaries are at full strength, the resulting stimulus is perceived as white.

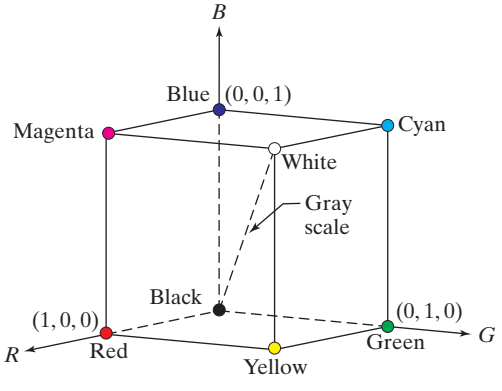
The RGB color space is frequently shown graphically as the RGB color cube in Fig. 7.9. The vertices of the cube are the *primary* (red, green, and blue) and *secondary* (cyan, magenta, and yellow) *colors of light*.

To view the color cube from any perspective, we use custom function `rgbcube`. Typing `rgbcube(vx,vy,vz)` at the prompt produces an RGB cube viewed from point `(vx,vy,vz)` on the MATLAB desktop. The resulting image can be saved to disk using function `print`, discussed in Section 2.4. The code for function `rgbcube` is:


```
function rgbcube(vx,vy,vz)
%RGBCUBE Displays an RGB cube on the MATLAB desktop.
%   RGBCUBE(VX,VY,VZ) displays an RGB color cube, viewed from point
%   (VX,VY,VZ). With no input arguments, RGBCUBE uses (10,10,4) as the
%   default viewing coordinates. To view individual color planes, use
```

a b

FIGURE 7.9
(a) Schematic of the RGB color cube showing the primary and secondary colors of light at the vertices. Points along the main diagonal have gray values from black at the origin to white at point (1, 1, 1). (b) The RGB color cube.



DEVICE MODELS AND ICC COLOR PROFILES

ICC Color Profiles

Document colors can have one appearance on a computer monitor and quite a different appearance when printed. Or the colors in a document may appear different when printed on different printers. In order to obtain high-quality, consistent color reproduction between different display and printing devices, it is necessary to create a transform to map colors from one device to another. In general, a separate color transform would be needed between every pair of devices. Other transforms would be needed to account for factors such as different printing conditions and device quality settings. Each of the many needed transforms would have to be developed using carefully-controlled experiments. Clearly such an approach would prove impractical for all but the most expensive, high-end systems.

The *International Color Consortium* (ICC), an industry group founded in 1993, has standardized a different approach: Each device has just two transforms associated with it, regardless of the number of other devices that may be present in the system. One of the transforms converts device colors to a standard, device-independent color space called the *profile connection space* (PCS). The other transform is the inverse of the first; it converts PCS colors back to device colors. (The PCS can be either XYZ or L*a*b*.) Together, the two transforms make up the *ICC color profile* for the device.

We discuss the L*a*b* color space later in this section.

One of the primary goals of the ICC has been to create, standardize, maintain, and promote the ICC color profile standard (ICC [2004]). The Image Processing Toolbox function `iccread` reads profile files. Its syntax is:

`iccread`

```
p = iccread(filename)
```

The output, `p`, is a structure containing file header information and the numerical coefficients and tables necessary to compute the color space conversions between device and PCS colors.

Converting colors using ICC profiles is done using the Toolbox functions `makecform` and `applycform`. The syntax of both functions is explained below. The ICC color profile standard includes mechanisms for handling a critical color conversion step called *gamut mapping*. A *color gamut* is a volume in color space that defines the range of colors that a device can reproduce (CIE [2004]). Color gamuts differ from device to device. For example, a typical monitor can display some colors that cannot be reproduced using a printer. Therefore, it is necessary to take different gamuts into account when mapping colors from one device to another. Gamut mapping is the process of compensating for differences between source and destination gamuts is called (ISO [2004]).

There are many different methods used for gamut mapping (Morovic [2008]), some better suited for specific purposes than others. The ICC color profile standard defines four "purposes" (called *rendering intents*) for gamut mapping. These rendering intents are listed in Table 7.2. The `makecform` syntax for specifying rendering intents is:

Hue describes (i.e., gives a name to) a pure color, such as red or blue. *Saturation* is a measure of color purity—to how much gray there is in a color. *Value* is a measure of how light or dark a color is. *Intensity* refers to the brightness of a color; in the HSI model it is defined as the average of the R, G, and B values. *Lightness* is sometimes defined as $[\max(R,G,B) + \min(R,G,B)]/2$, but this is only one of the many definitions found in the color sciences.

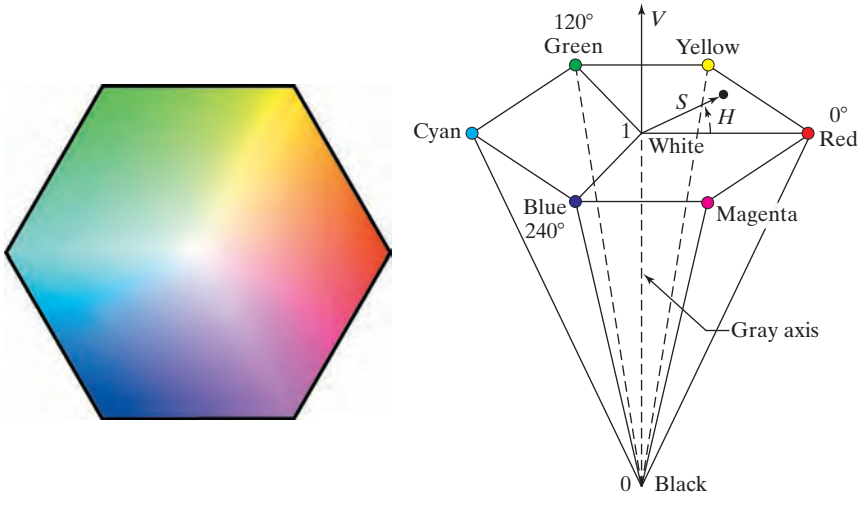
The HSV color space is formulated by looking at the RGB color cube along its gray axis (the axis joining the black and white vertices). The result is the hexagonally shaped color palette shown in Fig. 7.13(a). As we move along the gray axis in Fig. 7.13(b), the size of the hexagonal plane that is perpendicular to the axis changes, yielding the volume depicted in the figure. Hue is expressed as an angle around a color hexagon, typically using the red axis as the reference (0°) axis. The value component is measured along the gray axis of the cone. The $V = 0$ end of the axis is black. The $V = 1$ end of the axis is white, which lies in the center of the full color hexagon in Fig. 7.13(a). Thus, this axis represents all shades of gray. Saturation (purity of the color) is measured as the distance from the V axis, with maximum saturation being achieved at the maximum value of V, which represents white. Because of its geometry, this model is referred to as a *single hexagon* model.

Converting from RGB to HSV entails developing the equations to map RGB values (which are in Cartesian coordinates) to cylindrical coordinates. This topic is treated in detail in most texts on computer graphics (e.g., see Rogers [1997]), so we do not develop the equations here. In the following section we do develop the equations of the HSI model, which are similar.

The Toolbox function for converting from RGB to HSV is `rgb2hsv`, whose syntax is

`rgb2hsv`

```
hsv_image = rgb2hsv(rgb_image)
```



a b

FIGURE 7.13
(a) The HSV color hexagon. (b) The HSV single hexagonal cone.

```
% Implement the conversion equations.
R = zeros(size(hsi,1),size(hsi,2));
G = zeros(size(hsi,1),size(hsi,2));
B = zeros(size(hsi,1),size(hsi,2));

% RG sector (0 <= H < 2*pi/3).
idx = find( (0 <= H) & (H < 2*pi/3));
B(idx) = I(idx).*(1 - S(idx));
R(idx) = I(idx).*(1 + S(idx).*cos(H(idx))./...
                                        cos(pi/3 - H(idx)));
G(idx) = 3*I(idx) - (R(idx) + B(idx));

% BG sector (2*pi/3 <= H < 4*pi/3).
idx = find( (2*pi/3 <= H) & (H < 4*pi/3) );
R(idx) = I(idx).*(1 - S(idx));
G(idx) = I(idx).*(1 + S(idx).*cos(H(idx) - 2*pi/3)./...
                    cos(pi - H(idx)));
B(idx) = 3*I(idx) - (R(idx) + G(idx));

% BR sector.
idx = find( (4*pi/3 <= H) & (H <= 2*pi));
G(idx) = I(idx).*(1 - S(idx));
B(idx) = I(idx).*(1 + S(idx).*cos(H(idx) - 4*pi/3)./...
                                        cos(5*pi/3 - H(idx)));
R(idx) = 3*I(idx) - (G(idx) + B(idx));

% Combine all three results into an RGB image. Clip to [0,1] to
% compensate for floating-point arithmetic rounding effects.
rgb = cat(3,R,G,B);
rgb = max(min(rgb,1),0);
```

EXAMPLE 7.3: Converting from RGB to HSI.

Figure 7.17(a) shows an RGB image and Figs. 7.17(b)–(d) are its hue, saturation, and intensity components. They were obtained using the following commands:

```
>> f = imread('firebreather-midres.tif');
>> figure, imshow(f) % Fig. 7.17(a).
>> g = rgb2hsi(f);
>> figure, imshow(g(:,:,1)) % Fig. 7.17(b).
>> figure, imshow(g(:,:,2)) % Fig. 7.17(c).
>> figure, imshow(g(:,:,3)) % Fig. 7.17(d).
```

The hue image in Fig. 7.17(b) shows dark values in the region of the flames. The values of this image are angles measured with respect to the red axis. The flames have reddish yellow tones which are low angle values when measured with respect to this axis. In contrast, the flame colors are highly saturated, so the saturation image in Fig. 7.17(c) shows high values in that region. The intensity image looks like you would expect. It is a grayscale image containing the gray tones of the intensity component of the HSI image. The key importance of the intensity image in this example is that it shows the intensity completely decoupled from the color content of the image. Thus, we can use any of the many grayscale processing methods from earlier chapters without affecting the color tonality of the image.

a b c
d e f

FIGURE 7.17 (a) RGB image. (b) Hue image. (c) Saturation image. (d) Intensity image. All three images were obtained using function `rgb2hsi`. (e) Histogram-equalized intensity image. (f) RGB image obtained using function `hsi2rgb`. (Original image credit: Luc Viatour, **https://lucnix.be/**.)

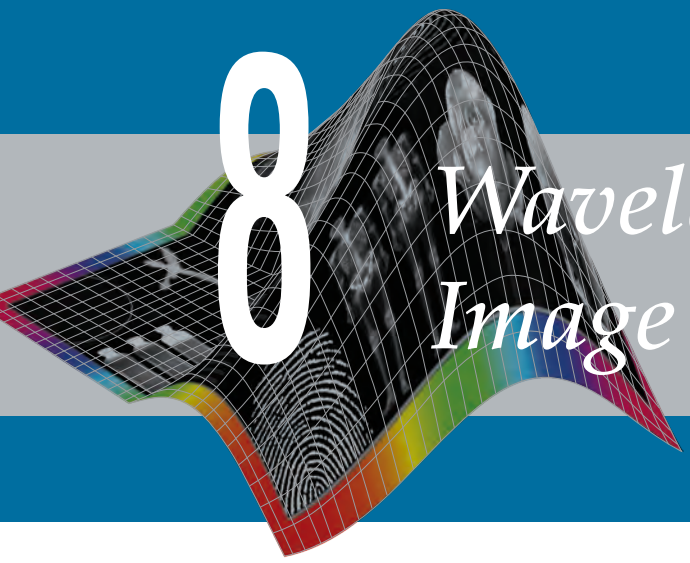
Figure 7.17(e) shows the result of histogram-equalizing the intensity image:

```
>> grayeq = histeq(g(:,:,3));
>> figure, imshow(grayeq) % Fig. 7.17(e).
```

We construct the enhanced RGB image using the histogram-equalized image in place of the intensity image in function `hsi2rgb`:

```
>> hsi = cat(3,g(:,:,1),g(:,:,2),grayeq);
>> rgbeq = hsi2rgb(hsi);
>> figure, imshow(rgbeq) % Fig. 7.17(f).
```

The histogram equalized RGB image shown in Fig. 7.17(f) retained the original colors, but they are considerably more vibrant. The difference in visible detail between the original and enhanced color images is significant. For example on the far right you see spectators' faces much more clearly. You can even see green trees past the right corner of the building. These details were barely visible or not visible at all in Fig. 7.17(a). You can also see additional details, such as the clothing adornments and the flowers on the left window, in the enhanced image. The equalized color image has a slight overexposed appearance, a condition caused by histogram equalization spreading the intensities to the full scale. A method like histogram matching would be capable of producing a more balanced intensity appearance.



8 Wavelet and Other Image Transforms

I take pleasure in my transformations. I look quiet and consistent, but few know how many women there are inside of me.
Anais Nin

The discrete Fourier transform is a member of an important class of linear transforms that decompose functions into weighted sums of orthogonal basis functions. It and many other transforms, including the discrete cosine, Walsh-Hadamard, and Haar transforms, can be studied using the tools of linear algebra and implemented via matrix operations, which are ideally suited for MATLAB. In this chapter, we present a general framework for both the computation and use of orthogonal image transforms and devote particular attention to the discrete wavelet transform, which places constraints beyond orthogonality on the decomposition functions employed. We introduce the Wavelet Toolbox, a collection of MathWorks functions designed for wavelet analysis but not included in the Image Processing Toolbox, and develop a compatible set of routines that allow wavelet-based processing using the Image Processing Toolbox alone. These custom functions, in combination with Image Processing Toolbox functions, provide the tools needed to implement the concepts discussed in Chapter 6 of *Digital Image Processing* by Gonzalez and Woods [2018]. They are applied in much the same way—and provide a similar range of capabilities—as MATLAB functions `fft2` and `ifft2`.

Functions Developed in this Chapter:

- `basisImage` displays the basis images of a 2D matrix-based transform.
- `whtmtx` generates the sequency-ordered transformation matrix of a matrix-based Walsh-Hadamard transform.
- `wavefilter` returns the wavelet decomposition and reconstruction filters used in a fast wavelet transform.
- `wavefast` computes the fast wavelet transform of a matrix using the decomposition filters provided by function `wavefilter`. The output of `wavefast` is a wavelet decomposition structure.
- Functions `wavework`, `wavecut`, `wavecopy`, and `wavepaste` are used to modify wavelet decomposition structures.
- `wavedisplay` displays the coefficients of a wavelet decomposition structure as they are commonly encountered in the literature.
- `waveback` computes the inverse FWT of a wavelet decomposition structure.
- `wavezero` zeroes the detail coefficients of a wavelet decomposition structure at a specified decomposition level.

8.1 MATRIX-BASED ORTHOGONAL TRANSFORMS

For a more detailed explanation of the concepts presented in this section, see Chapter 6 of Gonzalez and Woods [2018].

Consider a discrete function $f(x)$ of spatial variable $x = 0,1,\ldots,N-1$ with generalized discrete *forward transform*

$$T(u) = \sum_{x=0}^{N-1} f(x)r(x,u) \tag{8-1}$$

where *transform domain variable* $u = 0,1,\ldots,N-1$ and $r(x,u)$ is called a *forward transformation kernel*. Function $f(x)$ can be recovered from $T(u)$ using the *inverse transform*

$$f(x) = \sum_{u=0}^{N-1} T(u)s(x,u) \tag{8-2}$$

where $x = 0,1,\ldots,N-1$ and $s(x,u)$ is the transform's *inverse transformation kernel*. Equations (8-1) and (8-2) form a 1-D *transform pair* whose nature, computational complexity, and usefulness depend on the properties of $r(x,u)$ and $s(x,u)$. Their 2-D equivalent for square images $f(x, y)$ of size $N \times N$ can be similarly defined as

$$T(u,v) = \sum_{x=0}^{N-1}\sum_{y=0}^{N-1} f(x,y)r(x,y,u,v) \tag{8-3}$$

and

$$f(x,y) = \sum_{u=0}^{N-1}\sum_{v=0}^{N-1} T(u,v)s(x,y,u,v) \tag{8-4}$$

where both u and v are transform domain variables and $r(x,y,u,v)$ and $s(x,y,u,v)$ are tranformation kernels. If the transformation kernels are *separable* and *symmetric*,

$$\begin{aligned} r(x,y,u,v) &= r(x,u)r(y,v) \\ s(x,y,u,v) &= s(x,u)s(y,v) \end{aligned} \tag{8-5}$$

then the corresponding 2-D transforms [i.e., Eqs. (8-3) and (8-4)] can be computed by applying their 1-D counterparts [i.e., Eqs. (8-1) and (8-2)] in a row and column manner.

The right side of Eq. (8-2) can be viewed as a *series expansion* of $f(x)$ around a set of *expansion functions*, $s(x,u)$ for $u = 0,1,\ldots,N-1$, and *expansion coefficients* $T(u)$. If the expansion functions are represented as N-dimensional column vectors

$$\mathbf{s}_u = \begin{bmatrix} s(0,u) \\ s(1,u) \\ \vdots \\ s(N-1,u) \end{bmatrix} \quad \text{for } u = 0,1,\ldots,N-1 \tag{8-6}$$

```
            basisImages(x:x+N-1, y:y+N-1) = S;
         end
      end
      basisImages = mat2gray(basisImages);

      % Add borders and space between basis images.
      for i = 1:1:N
         for j = 1:1:N
            display(C*(i-1)+1:C*(i-1)+N+2, C*(j-1)+1:C*(j-1)+N+2) ...
               = border;
            display(C*(i-1)+2:C*(i-1)+N+1, C*(j-1)+2:C*(j-1)+N+1) ...
               = basisImages(N*i-N+1:N*i, N*j-N+1:N*j);
         end
      end

      imshow(display,[ ]);
   end
```

EXAMPLE 8.2: Basis images and correlation.

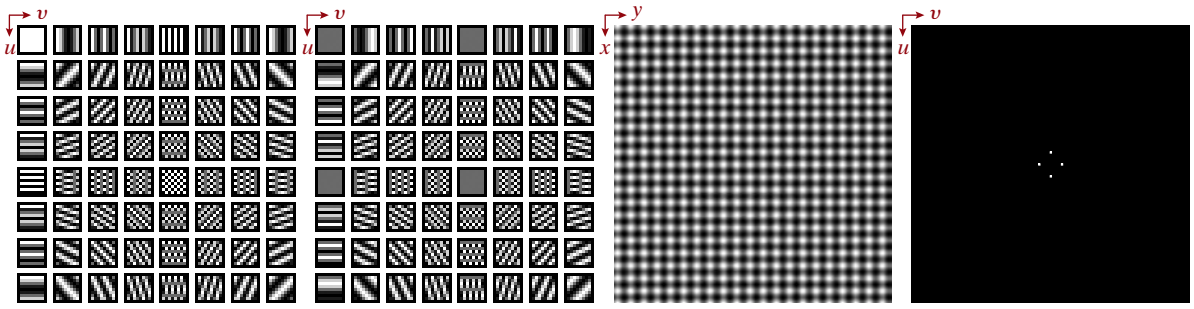
Custom function `basisImages` was used to generate the two 8×8 arrays of 8×8 DFT basis images in Figs. 8.3(a) and (b). To separate the basis images from one another and display them with a black border, we let

```
>> A = (1/sqrt(8))*dftmtx(8);
>> basisImages(A,0,2);
```

where parameters `gray` and `space` of function `basisImage` were set to `0` and `2`, respectively. Because the basis images of the DFT are complex-valued functions, the real and imaginary parts are displayed separately; a single DFT basis image is formed from one subimage of Fig. 8.3(a) and the corresponding subimage of Fig. 8.3(b). Thus, for example, the complex 8×8 basis subimage in the first row and and second column of Figs. 8.3(a) and (b) are the real and imaginary components of the DFT basis image with horizontal frequency and vertical frequencies 0 and $2\pi(1/8) = \pi/4$ radians/s, respectively. Row and column indices u and v determine the horizontal (i.e., along a row) and vertical (i.e., along a column) frequencies of each basis subimage. Note that the basis image of maximum frequency occurs when $u = v = 4$. As u and v increase above 4 (to 5, 6, and 7), the effective frequency decreases due to aliasing and produces conjugate symmetry.

Figure 8.3(c) shows a 512×512 image that combines a horizontal and vertical sinusoid of differing frequencies. Because DFT coefficients measure the similarity of their single-frequency, complex exponential basis functions and the image being transformed, they can be used to determine the frequencies of the image's constituent sinusoids:

```
>> A = (1/sqrt(512))*dftmtx(512);
>> F = load('sinusoids.mat'); F = F.F; imshow(F);
>> T = A*F*transpose(A);
>> figure; imshow(im2uint8(mat2gray(log(1 + abs(512*fftshift(T))))),[]);
>> [row, col] = find(abs(imag(T))>1e-9 | abs(real(T))>1e-9)

row =

    23
```



a b c d

FIGURE 8.3 (a) The real part of the DFT basis images for $N = 8$; (b) the imaginary part of the basis images; (c) an image composed of two sinusoids; and (d) the log scaled and centered spectrum of the image. The white pixels in (d) were enlarged to make them visible in print.

```
   491
     1
     1

col =

     1
     1
    22
   492
```

Note that the `find` function was used to locate the four nonzero values that are clearly present in Fig. 8.3(d). We simply search for transform coefficients with magnitudes greater than 10^{-9}. They occur at `row` and `col` coordinates (23, 1), (491, 1), (1, 22), and (1, 492), which correspond to frequency indices [i.e., (u,v) pairs] (22, 0), (490, 0), (0, 21), and (0, 491). The image in Fig. 8.3(c) is therefore composed of a vertical sinusoid of angular frequency $2(22)\pi/512$ rad/s, which corresponds to indices (22, 0) and (490, 0), and a horizontal sinusoid of frequency $2(21)\pi/512$ rad/s, which corresponds to indices (0, 21) and (0, 492). Note that half the coordinate pairs that are returned by the `find` function are a consequence of the DFT's conjugate symmetry.

TRANSFORMATION MATRICES

MATLAB provides several built-in functions for the computation of orthogonal transformation matrices. As noted in Example 8.1, for example, `dftmtx(N)` generates a scaled DFT transformation matrix that can be multiplied by $1/\sqrt{N}$ for use with the matrix equations in row 2 of Table 8.1. Of the remaining transforms in Fig. 8.2, MATLAB provides built-in functions for two: (1) the discrete cosine transform (DCT) and (2) the Walsh-Hadamard transform (WHT). DCT transformation matrices are computed using

`dctmtx`

```
A = dctmtx(N)
```

a
b

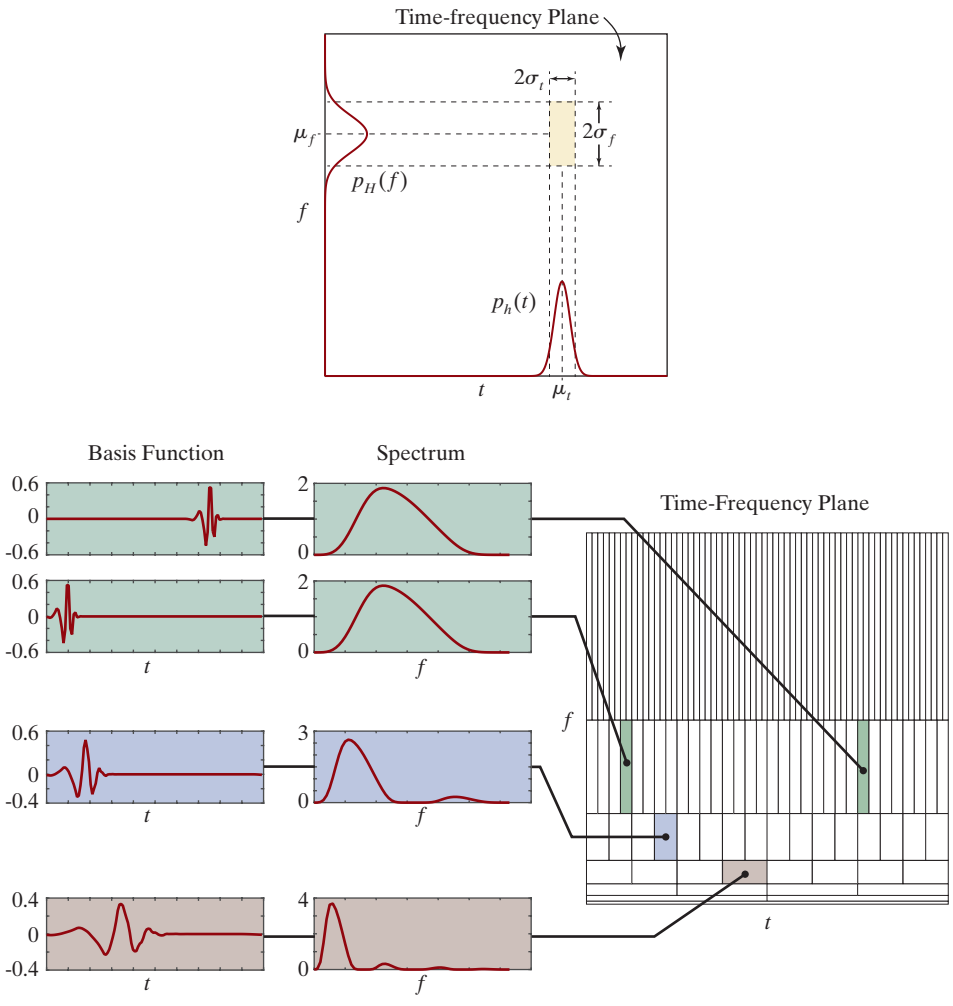


FIGURE 8.4
(a) Basis function localization in the time-frequency plane and (b) the time and frequency localization of 128-point Daubechies basis functions.

where f denotes frequency and $H(f)$ is the Fourier transform of $h(t)$. Then the energy[†] of basis function h, as illustrated in Fig. 8.4(a), is concentrated at (μ_t, μ_f) on the time-frequency plane. The majority of the energy falls in a rectangular region, called a *Heisenberg box* or *cell*, of area $4\sigma_t\sigma_f$ such that

$$\sigma_t^2\sigma_f^2 \geq \frac{1}{16\pi^2} \tag{8-25}$$

The constant on the right side of Eq. (8-25) is ¼ if stated in terms of angular frequency ω. Equality is possible, but only with a Gaussian basis function, whose transform is also a Gaussian function.

Since the *support* of a function can be defined as the set of points where the function is nonzero, *Heisenberg's uncertainty principle* tells us that it is impossible for a function to have finite support in both time and frequency. Equation (8-25), called the *Heisenberg-Gabor inequaltiy*, places a lower bound on the area of the Heisenberg

[†]The energy of continuous function $h(t)$ is $\int_{-\infty}^{\infty} |h(t)|^2\, dt$.

controls their height or amplitude. Note that the associated expansion functions are binary scalings and integer translates of *mother* wavelet $\psi(x) = \psi_{0,0}(x)$ and scaling function $\varphi(x) = \varphi_{0,0}(x)$.

2. *Multiresolution Compatibility*. The 1-D scaling function just introduced satisfies the following requirements of *multiresolution* analysis:
   - **(a)** $\varphi_{j,k}$ is orthogonal to its integer translates.
   - **(b)** The set of functions that can be represented as a series expansion of $\varphi_{j,k}$ at low scales or resolutions (i.e., small j) is contained within those that can be represented at higher scales.
   - **(c)** The only function that can be represented at every scale is $f(x) = 0$.
   - **(d)** Any function can be represented with arbitrary precision as $j \to \infty$.

   When these conditions are met, there is a companion wavelet $\psi_{j,k}$ that, together with its integer translates and binary scalings, spans—that is, can represent—the difference between any two sets of $\varphi_{j,k}$-representable functions at adjacent scales.
3. *Orthogonality*. The expansion functions [i.e., $\{\varphi_{j,k}(x)\}$] form an orthonormal or biorthogonal basis for the set of 1-D measurable, square-integrable functions. For a biorthogonal wavelet transform with scaling and wavelet functions $\varphi_{j,k}(x)$ and $\psi_{j,k}(x)$, the duals are denoted $\tilde{\varphi}_{j,k}(x)$ and $\tilde{\psi}_{j,k}(x)$ respectively.

8.4 THE FAST WAVELET TRANSFORM

An important consequence of the above properties is that both $\varphi(x)$ and $\psi(x)$ can be expressed as linear combinations of double-resolution copies of themselves—that is, via the series expansions

$$\varphi(x) = \sum_n h_\varphi(n)\sqrt{2}\varphi(2x - n) \tag{8-33}$$

$$\psi(x) = \sum_n h_\psi(n)\sqrt{2}\varphi(2x - n) \tag{8-34}$$

where h_φ and h_ψ—the expansion coefficients—are called *scaling* and *wavelet vectors*, respectively. They are the filter coefficients of the *fast wavelet transform* (FWT), an iterative computational approach to the DWT shown in Fig. 8.5. The $W_\varphi(j, m, n)$ and $\{W_\psi^i(j, m, n)$ for $i = H, V, D\}$ outputs in this figure are the DWT coefficients at scale j. Blocks containing time-reversed scaling and wavelet vectors—the $h_\varphi(-n)$ and $h_\psi(-m)$—are *lowpass* and *highpass decomposition filters*, respectively. Finally, blocks containing a 2 and a down arrow represent *downsampling*—extracting every other point from a sequence of points. Mathematically, the series of filtering and downsampling operations used to compute $W_\psi^H(j, m, n)$ in Fig. 8.5 is, for example,

$$W_\psi^H(j, m, n) = h_\psi(-m) \star \left[h_\varphi(-n) \star W_\varphi(j + 1, m, n)\big|_{n=2k,\, k\geq 0}\right]\big|_{m=2k,\, k\geq 0} \tag{8-35}$$

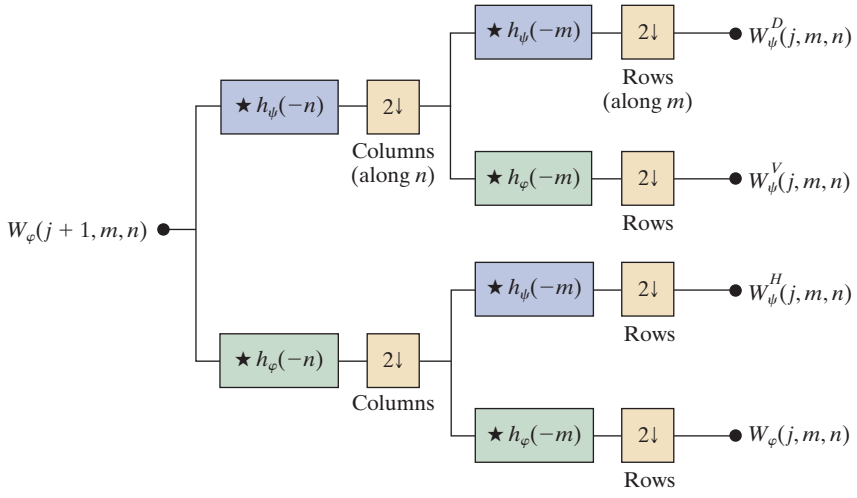


FIGURE 8.5
The 2-D fast wavelet transform (FWT) filter bank. Each pass generates one DWT scale. In the first iteration, $W_\varphi(j+1,m,n) = f(x,y)$.

where ★ denotes convolution. Evaluating convolutions at nonnegative, even indices is equivalent to filtering and downsampling by 2.

The input to the filter bank in Fig. 8.5 is decomposed into four lower resolution (or lower scale) components. The W_φ coefficients are created via two lowpass filters (i.e, h_φ-based) and are thus called *approximation coefficients*; $\{W_\psi^i \text{ for } i = H, V, D\}$ are *horizontal*, *vertical*, and *diagonal detail coefficients*, respectively. Output $W_\varphi(j,m,n)$ can be used as a subsequent input, $W_\varphi(j+1,m,n)$, to the block diagram for creating even lower resolution components; $f(x,y)$ is the highest resolution representation available and serves as the input for the first iteration. Note that the operations in Fig. 8.5 use neither wavelets nor scaling functions—only their associated wavelet and scaling vectors are used. In addition, three transform domain variables are involved—scale j and horizontal and vertical translation, n and m. These variables correspond to $u, v, \ldots$ in the first two equations of Section 8.1.

FAST WAVELET TRANSFORMS USING THE WAVELET TOOLBOX

In this section, we use the Wavelet Toolbox to compute the FWT of a 4×4 test image. In the next section, we will develop custom functions to do this without the Wavelet Toolbox (i.e., using the Image Processing Toolbox alone). The material here lays the groundwork for their development.

The Wavelet Toolbox provides decomposition filters for a wide variety of fast wavelet transforms. The filters associated with a specific transform are accessed via the function `wfilters`, which has the following general syntax:

`wfilters`
We denote MATLAB toolbox functions that are not part of the Image Processing Toolbox in brown.

```
[Lo_D,Hi_D,Lo_R,Hi_R] = wfilters(wname)
```

Here, input parameter `wname` determines the returned filter coefficients in accordance with Table 8.3; outputs `Lo_D`, `Hi_D`, `Lo_R`, and `Hi_R` are row vectors that contain the lowpass decomposition, highpass decomposition, lowpass reconstruction, and

The compositing just described takes place within the only `for` loop in `wavedisplay`. After checking the inputs for consistency, `wavecut` is called to remove the approximation coefficients from decomposition vector `c`. These coefficients are then scaled for later display using `mat2gray`. Modified decomposition vector `cd` (i.e., `c` without the approximation coefficients) is then similarly scaled. For positive values of input `scale`, the detail coefficients are scaled so that a coefficient value of 0 appears as middle gray; all necessary padding is performed with a `fill` value of 0.5 (mid-gray). If `scale` is negative, the absolute values of the detail coefficients are displayed with a value of 0 corresponding to black and the pad `fill` value is set to 0. After the approximation and detail coefficients have been scaled for display, the first iteration of the `for` loop extracts the last decomposition step's detail coefficients from `cd` and appends them to `w` (after padding to make the dimensions of the four subimages match and insertion of a one-pixel white border) via the `w = [w h; v d]` statement. This process is then repeated for each scale in `c`. Note the use of `wavecopy` to extract the various detail coefficients needed to form `w`.

EXAMPLE 8.8: Transform coefficient display using `wavedisplay`.

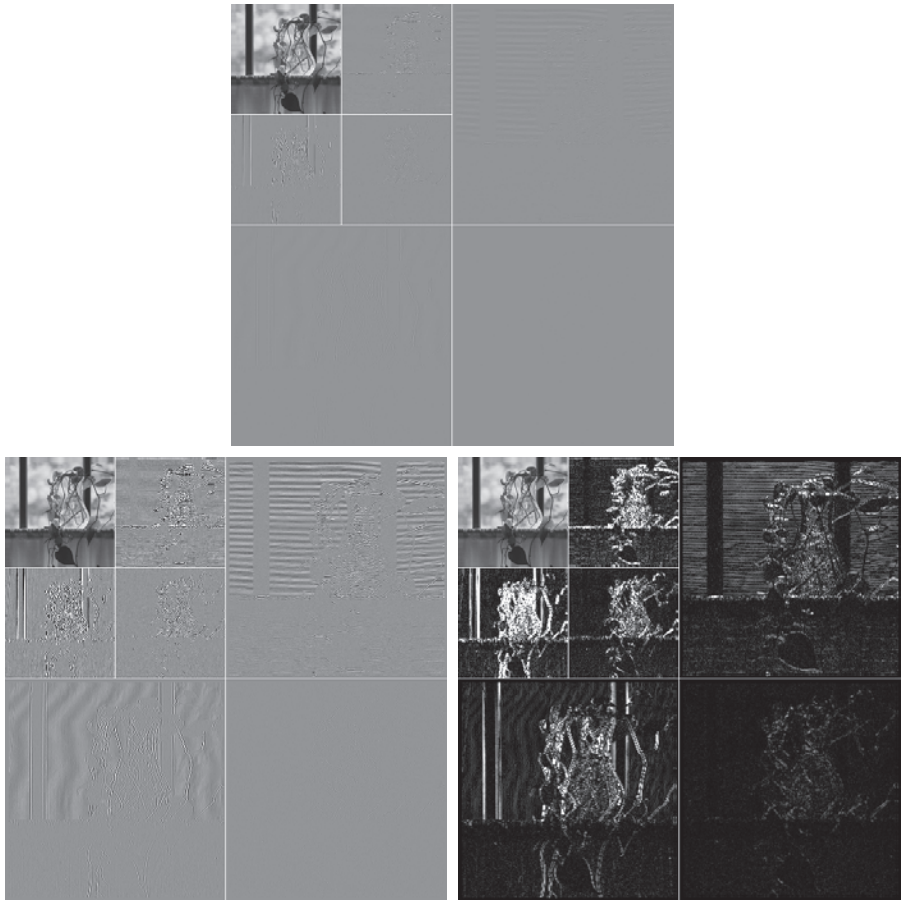
The following sequence of commands computes the two-scale DWT of the image in Fig. 8.7 with respect to fourth-order Daubechies' wavelets and displays the resulting coefficients:

```
>> f = imread('vase.tif');
>> [c,s] = wavefast(f,2,'db4');
>> wavedisplay(c,s);
>> figure; wavedisplay(c,s,8);
>> figure; wavedisplay(c,s,-8);
```

The images generated by the final three commands are shown in Figs. 8.8(a) through (c), respectively. Without additional scaling, the detail coefficient differences in Fig. 8.8(a) are barely visible. In Fig. 8.8(b), the differences are accentuated by multiplying the coefficients by 8. Note the mid-gray padding along the borders of the level 1 coefficient subimages; it was inserted to reconcile dimensional variations between transform coefficient subimages. Figure 8.8(c) shows the effect of taking the absolute values of the details. Here, all padding is done in black.

8.6 THE INVERSE FAST WAVELET TRANSFORM

Like its forward counterpart, the inverse fast wavelet transform can be computed iteratively using digital filters. Figure 8.9 shows the required synthesis or reconstruction filter bank, which reverses the process of the analysis or decomposition filter bank of Fig. 8.5. At each iteration, four scale j approximation and detail subimages are upsampled (by inserting zeroes between their elements) and convolved with two one-dimension filters—one operating on the subimages' columns and the other on its rows. Adding the results yields the scale $j+1$ approximation, and the process is repeated until the original image is reconstructed. The filters used in the convolutions are a function of the wavelets employed in the forward transform. Recall that



a
b c

FIGURE 8.8
Displaying a two-scale wavelet transform of the image in Fig. 8.7: (a) Automatic scaling; (b) additional scaling by 8; and (c) absolute values scaled by 8.

they can be obtained from the `wfilters` and `wavefilter` functions of Section 8.4 with input parameter `type` set to `'r'` for "reconstruction."

When using the Wavelet Toolbox, function `waverec2` is employed to compute the inverse FWT of wavelet decomposition structure `[C,S]`. It is invoked using

`waverec2`

```
g = waverec2(C,S,wname)
```

where `g` is the resulting reconstructed two-dimensional image (of class `double`). The required reconstruction filters can be alternately supplied using the syntax

```
g = waverec2(C,S,Lo_R,Hi_R)
```

The following custom function, which we call `waveback`, can be used when the Wavelet Toolbox is unavailable. It is the final function needed to complete our wavelet-based package for processing images in conjunction with the Image Processing Toolbox (and without the Wavelet Toolbox).

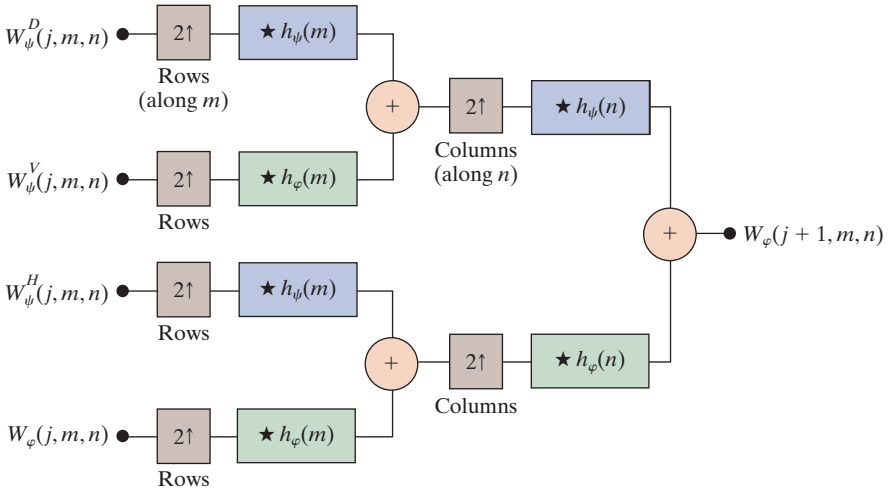


FIGURE 8.9
The 2-D FWT^{-1} filter bank. The boxes with the up arrow represent upsampling by inserting zeroes between every element.


```
function [varargout] = waveback(c,s,varargin)
%WAVEBACK Computes inverse FWTs for multi-level decomposition [C,S].
%   [VARARGOUT] = WAVEBACK(C,S,VARARGIN) performs a 2D N-level partial
%   or complete wavelet reconstruction of decomposition structure [C,S].
%
%   SYNTAX:
%   Y = WAVEBACK(C,S,'WNAME','EXTMODE');  Output inverse FWT matrix Y
%   Y = WAVEBACK(C,S,LR,HR,'EXTMODE');    using lowpass and highpass
%                                         reconstruction filters (LR and
%                                         HR) or filters obtained by
%                                         calling WAVEFILTER with 'WNAME'.
%
%   [NC,NS] = WAVEBACK(C,S,'WNAME','EXTMODE',N);  Output the wavelet
%   [NC,NS] = WAVEBACK(C,S,LR,HR,'EXTMODE',N);    decomposition structure
%                                                 {NC, NS] after N step
%                                                 reconstruction.
%
%   See also WAVEFAST and WAVEFILTER.

% Check the input and output arguments for reasonableness.
if (~ismatrix(c)) || (size(c,1) ~= 1)
   error('C must be a row vector.');
end

if (~ismatrix(s)) || ~isreal(s) || ~isnumeric(s) || ...
        ((size(s, 2) ~= 2) && (size(s,2) ~= 3))
   error('S must be a real, numeric two- or three-column array.');
end

elements = prod(s,2);
if (length(c) < elements(end)) || ...
      ~(elements(1) + 3 * sum(elements(2:end - 1)) >= elements(end))
   error(['[C S] must be a standard wavelet ' ...
          'decomposition structure.']);
```

3. Compute the inverse transform.

Because scale in the wavelet domain is analogous to frequency in the Fourier domain, most of the Fourier-based filtering techniques of Chapter 4 have an equivalent "wavelet domain" counterpart. In this section, we use the preceding three-step procedure to give several examples of the use of wavelets in image processing. Attention is restricted to the functions developed earlier in the chapter; the Wavelet Toolbox is not needed to implement the examples given here—nor the examples in Chapter 6 of *Digital Image Processing* (Gonzalez and Woods [2018]).

EXAMPLE 8.10: Wavelet directionality and edge detection.

Consider the 512×512 test image in Fig. 8.10(a). We used this image in Chapter 4 to illustrate smoothing and sharpening in the frequency domain. Here, we use it to demonstrate the directional sensitivity of the 2-D wavelet transform and its usefulness in edge detection:

```
>> f = imread('testpattern512.tif');
>> imshow(f);
>> [c,s] = wavefast(f,1,'sym4');
>> figure; wavedisplay(c,s,-6);
>> [nc,y] = wavecut('a',c,s);
>> figure; wavedisplay(nc,s,-6);
>> edges = abs(waveback(nc,s,'sym4'));
>> figure; imshow(mat2gray(edges));
```

The horizontal, vertical, and diagonal directionality of the single-scale wavelet transform of Fig. 8.10(a) with respect to `'sym4'` wavelets is clearly visible in Fig. 8.10(b). Note, for example, that the horizontal edges of the original image are present in the horizontal detail coefficients of the upper-right quadrant of Fig. 8.10(b). The vertical edges of the image can be similarly identified in the vertical detail coefficients of the lower-left quadrant. To combine this information into a single edge image, we simply zero the approximation coefficients of the generated transform, compute its inverse, and take the absolute value. The modified transform and resulting edge image are shown in Figs. 8.10(c) and (d), respectively. A similar procedure can be used to isolate the vertical or horizontal edges alone.

EXAMPLE 8.11: Wavelet-based image smoothing or blurring.

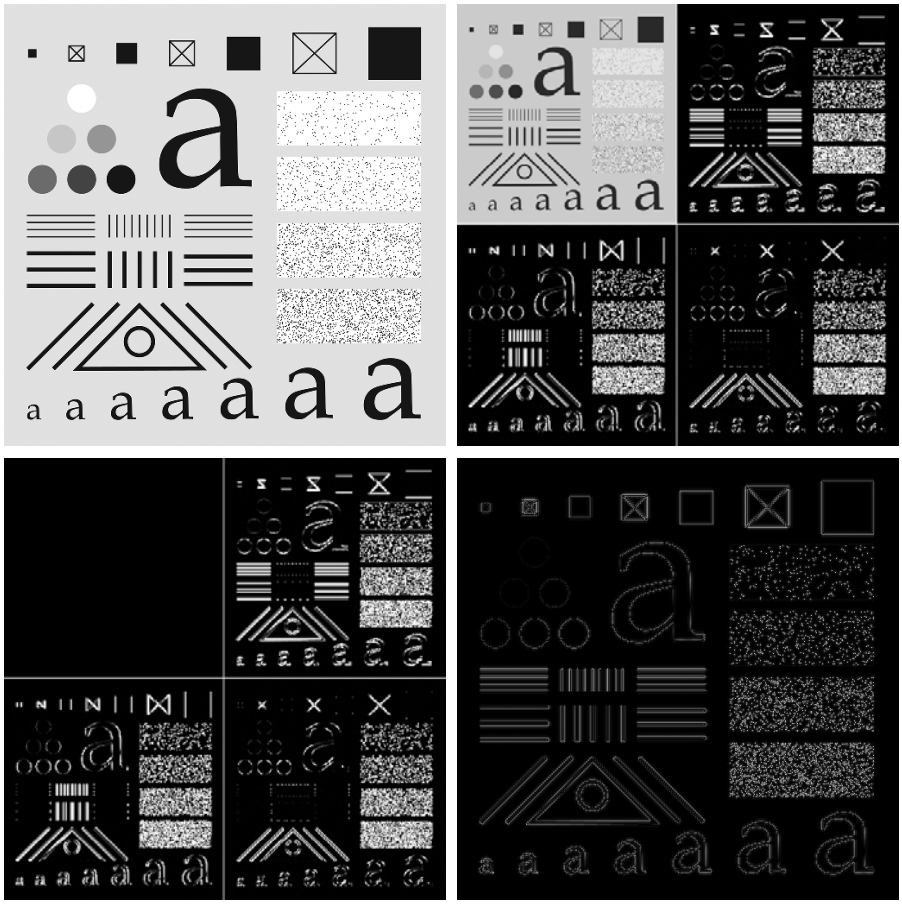
Symlets, short for "symmetrical wavelets", have minimal assymetry for a given compact support.

Wavelets, like their Fourier counterparts, are effective instruments for smoothing or blurring images. Consider again the test image of Fig. 8.10(a), which is repeated in Fig. 8.11(a). Its wavelet transform with respect to fourth-order symlets is shown in Fig. 8.11(b), where it is clear that a four-scale decomposition has been performed. To streamline the smoothing process, we employ the following utility function:


```
function [nc,g8] = wavezero(c,s,l,wname)
%WAVEZERO Zeroes wavelet transform detail coefficients.
%   [NC,G8] = WAVEZERO(C,S,L,WNAME) zeroes the level L detail
%   coefficients in wavelet decomposition structure [C,S] and computes
%   the resulting inverse transform with respect to WNAME wavelets.
```

a b
c d

FIGURE 8.10
Wavelets in edge detection: (a) A simple test image; (b) its single-scale wavelet transform; (c) the transform modified by zeroing all approximation coefficients; and (d) the edge image resulting from computing the absolute value of the inverse transform.

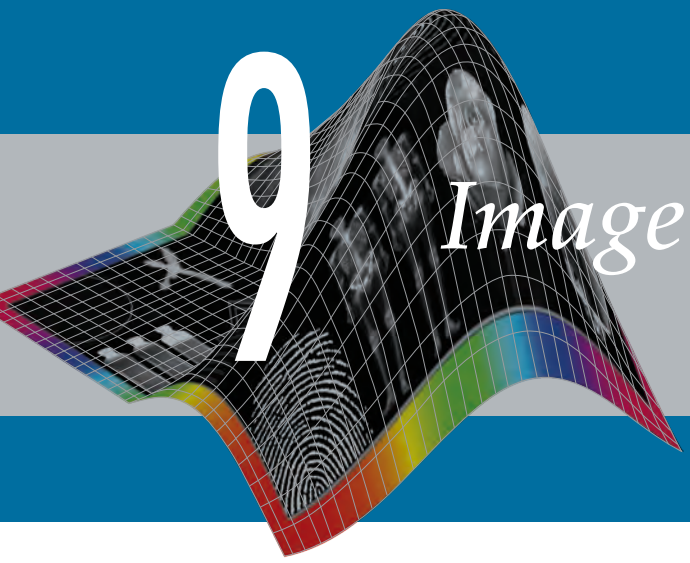


```
[nc,~] = wavecut('h',c,s,1);
[nc,~] = wavecut('v',nc,s,1);
[nc,~] = wavecut('d',nc,s,1);
i = waveback(nc,s,wname);
g8 = im2uint8(mat2gray(i));
figure; imshow(g8);
```

A series of increasingly smoothed versions of Fig. 8.11(a) can be generated using function `wavezero`:

```
>> f = imread('testpattern512.tif');
>> [c,s] = wavefast(f,4,'sym4');
>> wavedisplay(c,s,20);
>> [c,g8] = wavezero(c,s,1,'sym4');
>> [c,g8] = wavezero(c,s,2,'sym4');
>> [c,g8] = wavezero(c,s,3,'sym4');
>> [c,g8] = wavezero(c,s,4,'sym4');
```

Note that the smoothed image in Fig. 8.11(c) is only slightly blurred, as it was



9 Image Compression

The greatest artist is the simplifier.
Donald M. Murray

Image compression addresses the problem of reducing the amount of data required to represent a digital image. Because the Image Processing Toolbox does not include functions for image compression, the goal of this chapter is to provide practical ways of exploring compression techniques within the context of MATLAB. For instance, we develop a MATLAB callable C function that illustrates how to manipulate variable-length data representations at the bit level. Variable-length coding is a mainstay of image compression, but MATLAB is best at processing matrices of uniform (i.e., fixed length) data. During the development of the function, we assume that the reader has a working knowledge of the C language and focus our discussion on how to make MATLAB interact with programs (both C and Fortran) external to the MATLAB environment. This is an important skill when there is a need to interface MATLAB functions to preexisting C or Fortran programs, and when vectorized functions need to be speeded up (e.g., when a `for` loop can not be adequately vectorized). The range of compression functions developed in this chapter, together with MATLAB's ability to treat C and Fortran programs as though they were conventional MATLAB files or built-in functions, is another illustration of the fact mentioned in Chapter 1 that MATLAB is an effective tool for prototyping image compression systems and algorithms.

Functions Developed in this Chapter:

- `imratio` computes the ratio of the number of bytes in two images.
- `compare` computes the error between two matrices and optionally displays it.
- `ntrop` computes the entropy of a matrix.
- `huffman` computes a Huffman code.
- `mat2huff` and `huff2mat` perform Huffman encoding and decoding, respectively. C function `unravel` is used by `huff2mat`.
- `mat2lpc` and `lpc2mat` compresses a matrix using linear predictive coding and decodes an encoded matrix, respectively.
- `quantize` quantizes a matrix.
- `im2jpeg` and `jpeg2im` compress and decompress a matrix using an approximation of the JPEG compression standard.
- `im2jpeg2k` and `jpeg2k2im` compress and decompress a matrix using an approximation of the JPEG2000 compression standard.
- `tifs2cv` and `cv2tifs` compress and decompress a multi-frame TIFF image sequence.
- `showmo` displays the motion vectors for selected frames of a `tif2cv` compressed image sequence.

9.1 BACKGROUND

In video compression systems, $f(x, y)$ would be replaced by $f(x, y, t)$ and frames would be sequentially fed into the block diagram of Fig. 9.1.

As Fig. 9.1 shows, image compression systems are composed of two distinct structural blocks: an *encoder* and a *decoder*. Image $f(x, y)$ is fed into the encoder, which creates a set of symbols from the input data and uses them to represent the image. If we let n_1 and n_2 denote the number of information carrying units (usually bits) in the original and encoded images, respectively, the compression that is achieved can be quantified numerically via the *compression ratio*

$$C_R = \frac{n_1}{n_2} \tag{9-1}$$

A compression ratio like 10 (or 10 : 1) indicates that the original image has 10 information carrying units (e.g., bits) for every 1 unit in the compressed data set. In MATLAB, the ratio of the number of bits used in the representation of two image files and/or variables can be computed using the following custom function:


```
function cr = imratio(f1,f2)
%IMRATIO Computes the ratio of the bytes in two images/variables.
%   CR = IMRATIO(F1,F2) returns the ratio of the number of bytes in
%   variables/files F1 and F2. If F1 and F2 are an original and
%   compressed image, respectively, CR is the compression ratio.

narginchk(2,2);                         % Check input arguments
cr = bytes(f1)/bytes(f2);               % Compute the ratio

%-------------------------------------------------------------------------%
function b = bytes(f)
% Return the number of bytes in input f. If f is a string, assume
% that it is an image filename; if not, it is an image variable.

if ischar(f)
   info = dir(f);        b = info.bytes;
elseif isstruct(f)
   % MATLAB's whos function reports an extra 124 bytes of memory per
   % structure field because of the way MATLAB stores structures in
```

dir

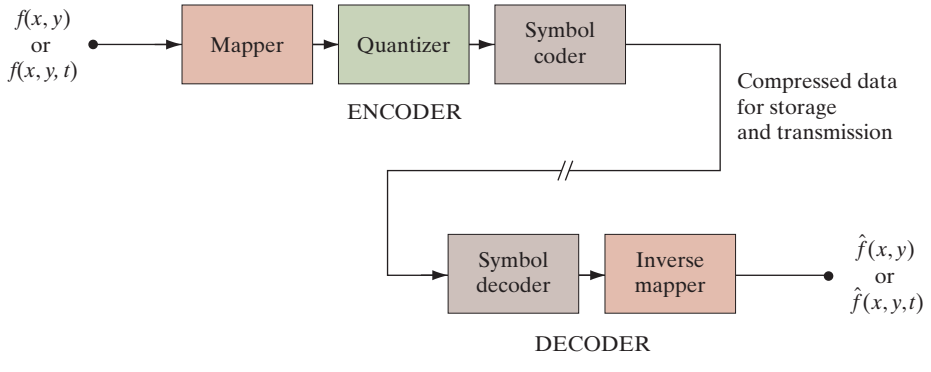


FIGURE 9.1 A general image compression system block diagram.

```
        % memory. Don't count this extra memory; instead, add up the memory
        % associated with each field.
        b = 0;
        fields = fieldnames(f);
        for k = 1:numel(f)
           for q = 1:length(fields)
               b = b + bytes(f(k).(fields{q}));
           end
        end
     else
        info = whos('f');     b = info.bytes;
     end
```

fieldnames

For example, the compression of the JPEG encoded image in Fig. 2.5(c) of Chapter 2 can be computed via

```
>> r = imratio(imread('bubbles25.jpg'),'bubbles25.jpg')
r =
    37.0945
```

Note that in function `imratio`, local function `b = bytes(f)` is designed to return the number of bytes in (1) a file, (2) a structure variable, and/or (3) a nonstructure variable. If `f` is a nonstructure variable, function `whos`, introduced in Section 2.2, is used to get its size in bytes. If `f` is a file name, function `dir` performs a similar service. In the syntax employed, `dir` returns a structure (see Section 2.10 for more on structures) with fields `name`, `date`, `bytes`, and `isdir`. They contain the file's name, modification date, size in bytes, and whether or not it is a directory (`isdir` is 1 if it is and is 0 otherwise), respectively. Finally, if `f` is a structure, `bytes` calls itself recursively to sum the number of bytes allocated to each field of the structure. This eliminates the overhead associated with the structure variable itself (124 bytes per field), returning only the number of bytes needed for the data in the fields. Function `fieldnames` is used to retrieve a list of the fields in `f`, and the statements

```
for k = 1:length(fields)
    b = b + bytes(f(k).(fields{q}));
```

perform the recursions. Note the use of *dynamic structure fieldnames* in the recursive calls to `bytes`. If `S` is a structure and `F` is a string variable containing a field name, the statements

```
S.(F) = foo;
field = S.(F);
```

employ the dynamic structure fieldname syntax to set and/or get the contents of structure field `F`, respectively.

To view and/or use a compressed (i.e., encoded) image, it must be fed into a decoder (see Fig. 9.1), where a reconstructed output image, $\hat{f}(x, y)$ is generated. In general, $\hat{f}(x, y)$ may or may not be an exact representation of $f(x, y)$. If it is, the system is called *error free*, *information preserving*, or *lossless*; if not, some level of

EXAMPLE 9.4: Decoding with `huff2mat`.

The Huffman encoded image of Example 9.3 can be decoded using the following sequence of commands:

load

Function `load` reads MATLAB variables from a file and loads them into the workspace. The variable names are maintained through a `save/load` sequence.

```
>> load squeeze-tracy;
>> g = huff2mat(c);
>> f = imread('tracy.tif');
>> rmse = compare(f, g)
rmse =
     0
```

Note that the overall encoding-decoding process is information preserving; the root-mean-square error between the original and decompressed images is 0. Because such a large part of the decoding job is done in C MEX-file `unravel`, `huff2mat` is slightly faster than its encoding counterpart, `mat2huff`. Note the use of the `load` function to retrieve the MAT-file encoded output from Example 9.2.

9.3 SPATIAL REDUNDANCY

Consider the images shown in Figs. 9.7(a) and (c). As Figs. 9.7(b) and (d) show, they have virtually identical histograms. Note also that the histograms are trimodal, indicating the presence of three dominant ranges of gray-level values. Because the gray levels of the images are not equally probable, variable-length coding can be used to reduce the coding redundancy that would result from a natural binary coding of the pixel values:

```
>> f1 = imread('matches-random.tif');
>> c1 = mat2huff(f1);
>> ntrop(f1)
ans =
    7.4253
>> imratio(f1, c1)
ans =
    1.0704
>> f2 = imread('matches-ordered.tif');
>> c2 = mat2huff(f2);
>> ntrop(f2)
ans =
    7.3505
>> imratio(f2, c2)
ans =
    1.0821
```

Note that the first-order entropy estimates of the two images are about the same (7.4253 and 7.3505 bits/pixel); they are compressed similarly by `mat2huff` (with compression ratios of 1.0704 versus 1.0821). These observations highlight the fact that variable-length coding is not designed to take advantage of

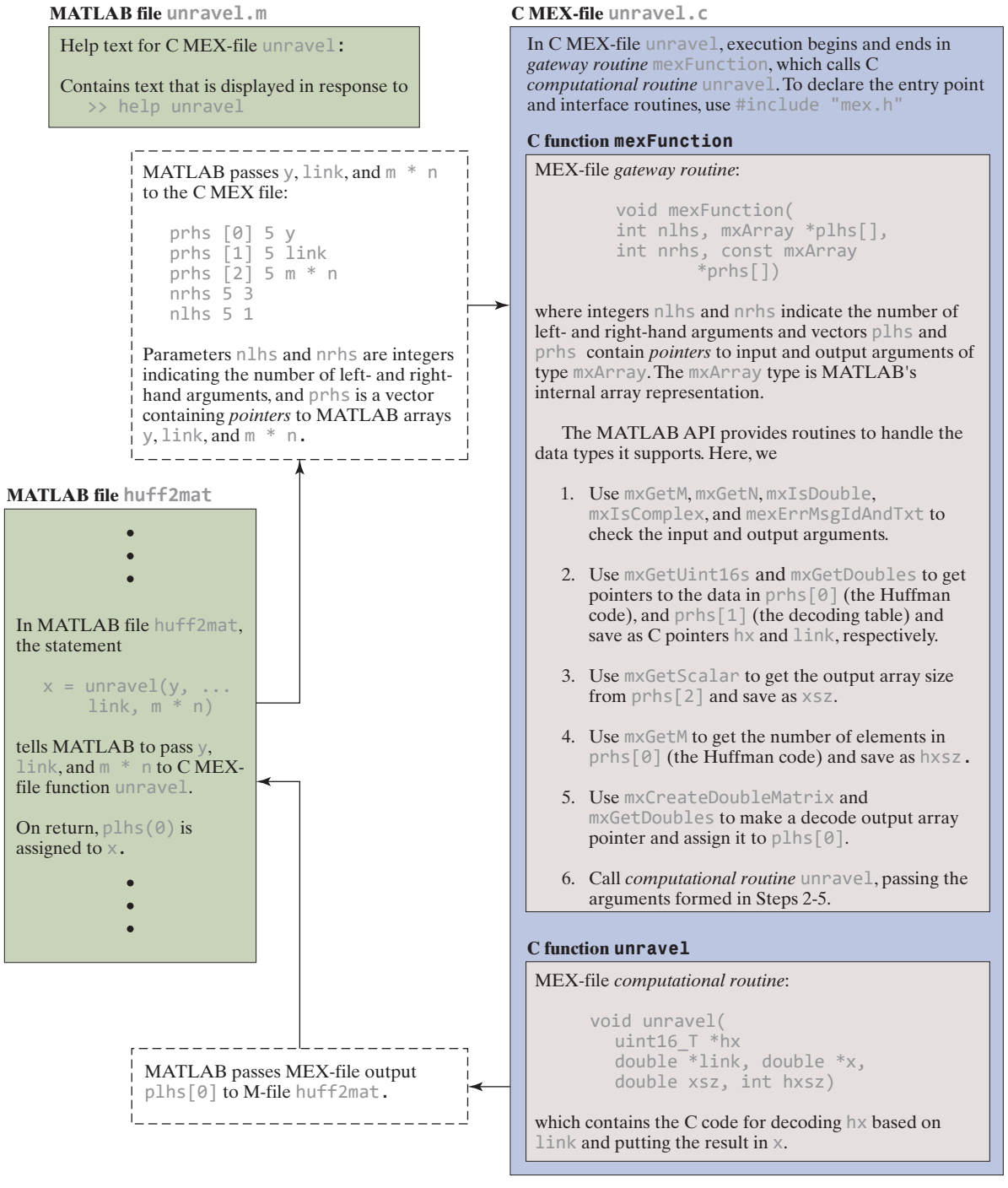
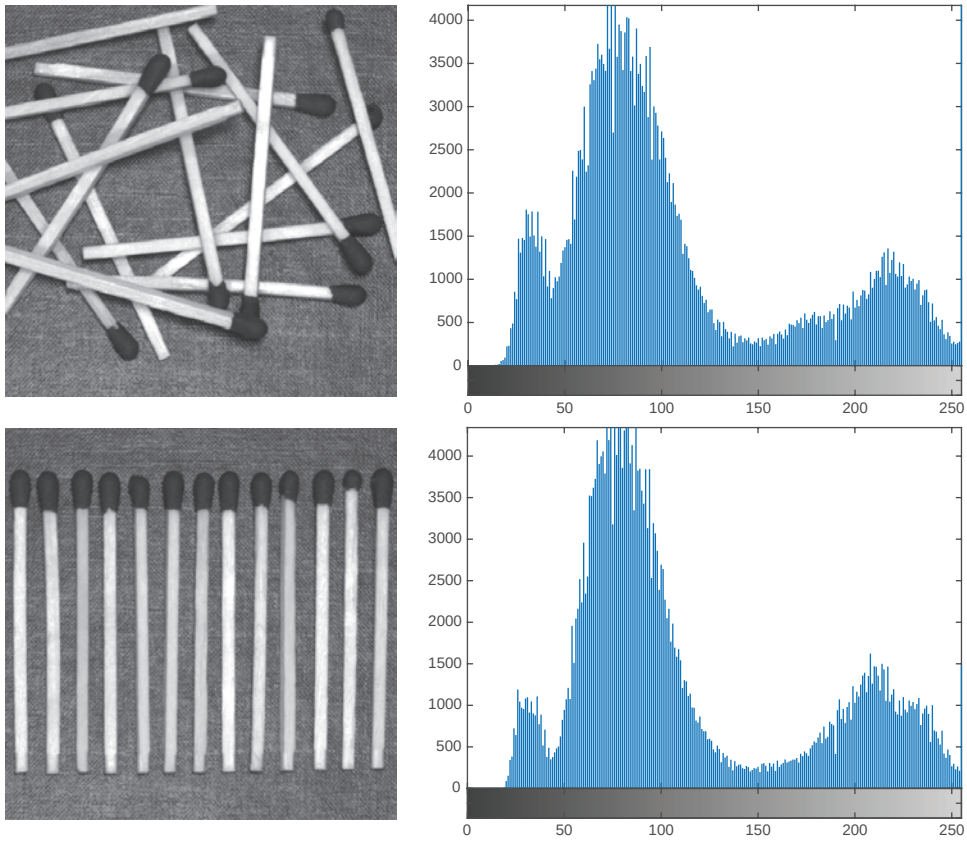


FIGURE 9.6 The interaction of function `huff2mat` and MATLAB callable C function `unravel`. Note that MEX-file `unravel` contains two functions: gateway routine `mexFunction` and computational routine `unravel`. Help text for MEX-file `unravel` is contained in the separate MATLAB file, also named `unravel`.



a b
c d

FIGURE 9.7
Two images and their gray-level histograms.

the obvious structural relationships between the aligned matches in Fig. 9.7(c). Although the pixel-to-pixel correlations are more evident in that image, they are present also in Fig. 9.7(a). Because the values of the pixels in either image can be reasonably predicted from the values of their neighbors, the information carried by individual pixels is relatively small. Much of the visual contribution of a single pixel to an image is redundant; it could have been guessed on the basis of the values of its neighbors. These correlations are the underlying basis of interpixel redundancy.

In order to reduce *interpixel redundancies*, the 2-D pixel array normally used for human viewing and interpretation must be transformed into a more efficient (but normally "nonvisual") format. For example, the differences between adjacent pixels can be used to represent an image. Transformations of this type (that is, those that remove interpixel redundancy) are referred to as *mappings*. They are called *reversible mappings* if the original image elements can be reconstructed from the transformed data set.

A simple mapping procedure is illustrated in Fig. 9.8. The approach, called *lossless predictive coding*, eliminates the interpixel redundancies of closely spaced pixels by extracting and coding only the new information in each pixel. The *new information*

sequence of commands combines IGS quantization, lossless predictive coding, and Huffman coding to compress the image of Fig. 9.10(a) to less than a quarter of its original size:

```
>> f = imread('brushes.tif');
>> q = quantize(f,4,'igs');
>> qs = double(q)/16;
>> e = mat2lpc(qs);
>> c = mat2huff(e);
>> imratio(f,c)
ans =
    4.1420
```

Encoded result `c` can be decompressed by the inverse sequence of operations (without "inverse quantization"):

```
>> ne = huff2mat(c);
>> nqs = lpc2mat(ne);
>> nq = 16*nqs;
>> compare(q,nq)
ans =
    0
>> compare(f,nq)
ans =
    6.8382
```

Note that the root-mean-square error of the decompressed image is about 7 gray levels—and that this error results from the quantization step alone.

9.5 JPEG COMPRESSION

The techniques of the previous sections operate directly on the pixels of an image and thus are *spatial domain methods*. In this section, we consider a family of popular compression standards that are based on modifying the transform of an image. Our objectives in this section are to introduce the use of 2-D transforms in image compression, to provide additional examples of how to reduce the image redundancies discussed in Sections 9.2 through 9.4, and to give the reader a feel for the state of the art in image compression. The standards presented (although we consider only approximations of them) are designed to handle a wide range of image types and compression requirements.

In *transform coding*, a reversible, linear transform like the DFT of Chapter 4 or the *discrete cosine transform* (DCT) of Chapter 8 is used to map an image into a set of transform coefficients, which are then quantized and coded. For most natural images, a significant number of the coefficients have small magnitudes and can be coarsely quantized (or discarded entirely) with little loss of visual image quality.

JPEG

One of the most popular and comprehensive continuous tone, still frame compression standards is the *JPEG (for Joint Photographic Experts Group) standard*. In the JPEG *baseline coding standard*, which is based on the discrete cosine transform and is adequate for most compression applications, the input and output images are limited to 8 bits, while the quantized DCT coefficient values are restricted to 11 bits.

```
m = double(y.quality)/100*m;          % Get encoding quality.
xb = double(y.numblocks);             % Get x blocks.
sz = double(y.size);
xn = sz(2);                           % Get x columns.
xm = sz(1);                           % Get x rows.
x = huff2mat(y.huffman);              % Huffman decode.
eob = max(x(:));                      % Get end-of-block symbol

z = zeros(64,xb);   k = 1;            % Form block columns by copying
for j = 1:xb                          % successive values from x into
   for i = 1:64                       % columns of z, while changing
      if x(k) == eob                  % to the next column whenever
         k = k + 1;   break;          % an EOB symbol is found.
      else
         z(i,j) = x(k);
         k = k + 1;
      end
   end
end

z = z(rev,:);                                   % Restore order
x = col2im(z,[8 8],[xm xn],'distinct');         % Form matrix blocks

fun = @(block_struct)denorm(block_struct.data,m);
x = blockproc(x,[8 8],fun);                     % Denormalize DCT
t = dctmtx(8);                                  % Get 8 x 8 DCT matrix
fun = @(block_struct)iblkdct(block_struct.data,t);
x = blockproc(x,[8 8],fun);                     % Compute block DCT-1
x = x + double(2^(y.bits - 1));                 % Level shift
if y.bits <= 8
    x = uint8(x);
else
    x = uint16(x);
end

% Inverse DCT matrix multiplications
function y = iblkdct(x,a)
y = a'*x*a;

% Denormalize DCT
function y = denorm(x,m)
y = x.*m;
```

EXAMPLE 9.8: JPEG compression.

Figures 9.13(a) and (b) show two JPEG coded and decoded approximations of the monochrome image in Fig. 9.4(a). The first result, which has a compression ratio of about 18 to 1, was obtained by direct application of the normalization array in Fig. 9.12(a). The second, which compresses the original image by a ratio of 42 to 1, was generated by multiplying (scaling) the normalization array by 4.

The differences between the original image of Fig. 9.4(a) and the reconstructed images of Figs. 9.13(a) and (b) are shown in Figs. 9.13(c) and (d) respectively. Both images were scaled to make the errors more visible. The corresponding rms errors are 2.4 and 4.4 gray levels. The effect of these errors on picture

a b
c d
e f

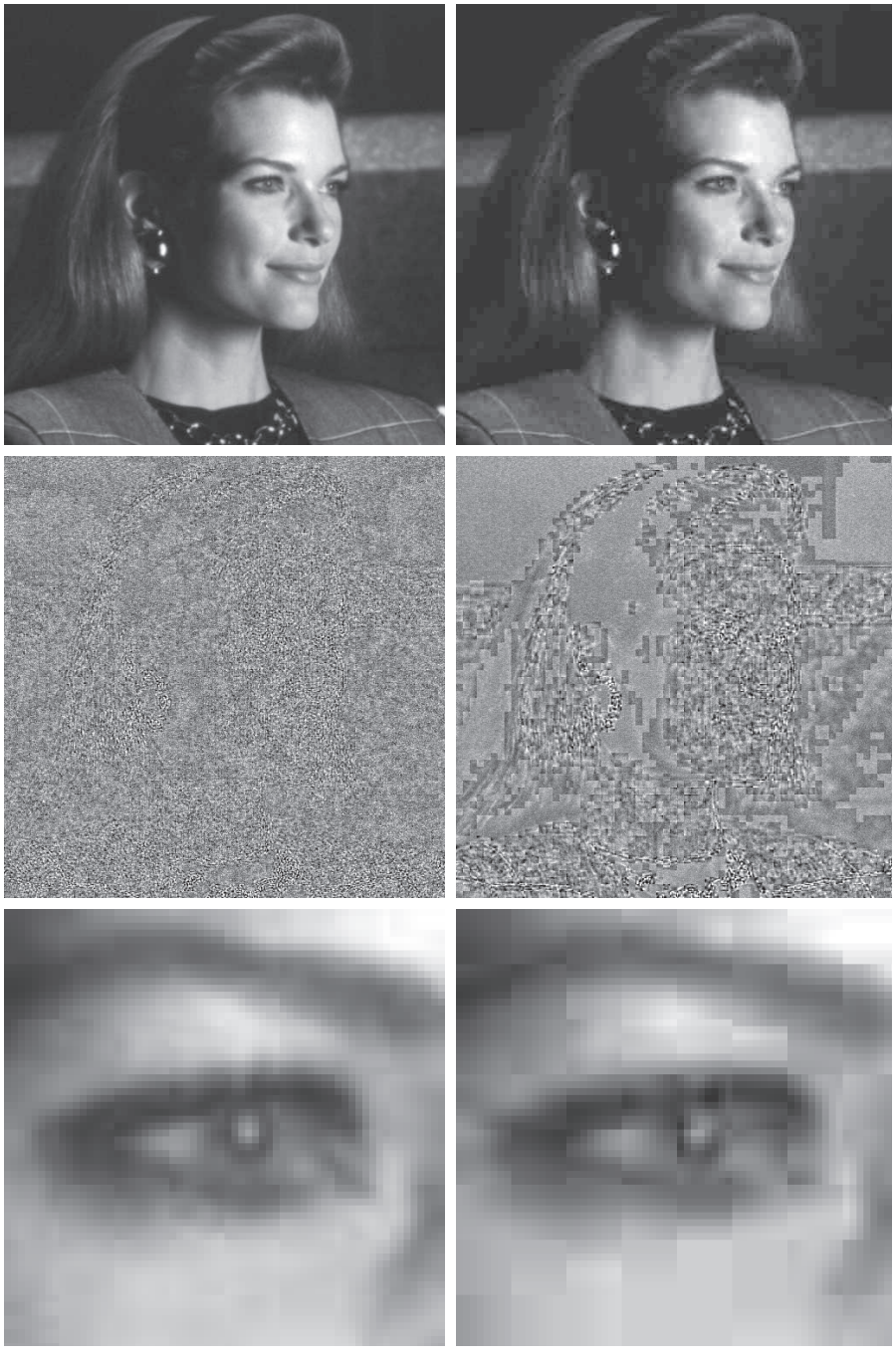


FIGURE 9.13
Left column: Approximations of Fig. 9.4 using the DCT and normalization array of Fig. 9.12(a). Right column: Similar results with the normalization array scaled by a factor of 4.

quality is more visible in the zoomed images of Figs. 9.13(e) and (f). These images show a magnified section of Figs. 9.13(a) and (b), respectively, and allow a better assessment of the subtle differences between the reconstructed images. [Figure 9.4(b) shows the zoomed original.] Note the *blocking artifact* that is present in both zoomed approximations.

We generated the images in Fig. 9.13 and the numerical results just discussed were generated with the following sequence of commands:

```
>> f = imread('tracy.tif');
>> c1 = im2jpeg(f);
>> f1 = jpeg2im(c1);
>> imshow(f1);
>> imratio(f,c1)
ans =
    18.4116
>> compare(f,f1,3)
ans =
    2.4329
>> c4 = im2jpeg(f,4);
>> f4 = jpeg2im(c4);
>> figure; imshow(f4);
>> imratio(f,c4)
ans =
    43.3153
>> compare(f,f4,3)
ans =
    4.4052
```

These results differ from those that would be obtained in a real JPEG baseline coding environment because `im2jpeg` approximates the JPEG standard's Huffman encoding process. Two principal differences are noteworthy: (1) In the standard, all runs of coefficient zeros are Huffman coded, while `im2jpeg` only encodes the terminating run of each block; and (2) the encoder and decoder of the standard are based on a known (default) Huffman code, while `im2jpeg` carries the information needed to reconstruct the encoding Huffman code words on an image to image basis. Using the standard, the compressions ratios noted above would be approximately doubled.

JPEG 2000

Like the initial JPEG release of the previous section, JPEG 2000 is based on the idea that the coefficients of a transform that decorrelates the pixels of an image can be coded more efficiently than the original pixels themselves. If the transform's basis functions—wavelets in the JPEG 2000 case—pack most of the important visual information into a small number of coefficients, the remaining coefficients can be quantized coarsely or truncated to zero with little loss of image quality.

Figure 9.14 shows a simplified JPEG 2000 coding system (absent several optional operations). The first step of the encoding process, as in the original JPEG standard, is to level shift the pixels of the image by subtracting 2^{m-1}, where 2^m is the number of gray levels in the image. The one-dimensional discrete wavelet transform of the rows and the columns of the image are then be computed. For error-free compression, the transform used is biorthogonal, with a 5-3 coefficient scaling and wavelet vector.

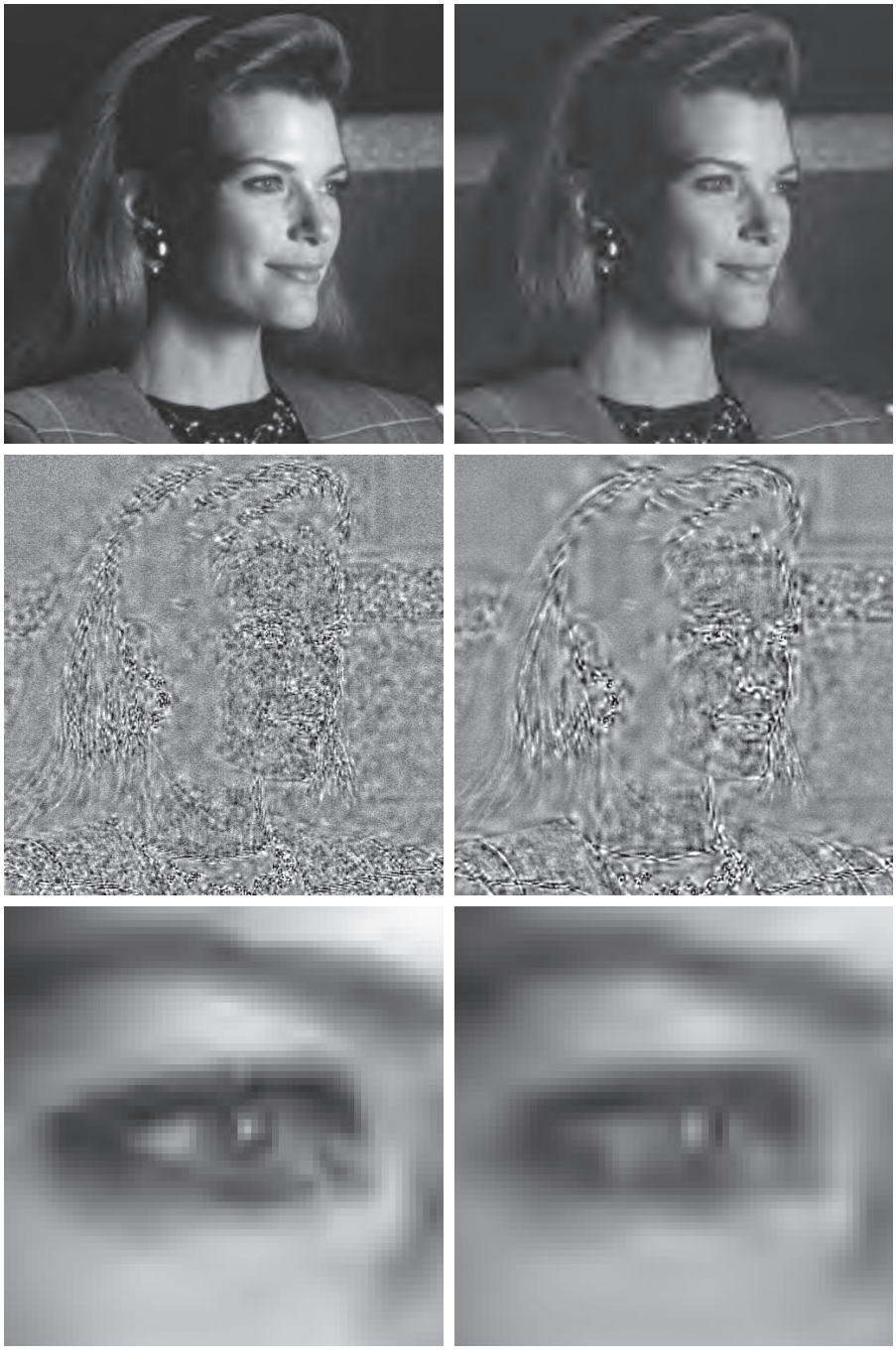
```
for j = 1:length(zi)
   c = [c r(i:zi(j) - 1) zeros(1,runs(r(zi(j) + 1)))];
   i = zi(j) + 2;
end

zi = find(r == eoc);                % Undo terminating zero run
if length(zi) == 1                  % or last non-zero run.
   c = [c r(i:zi - 1)];
   c = [c zeros(1,cl - length(c))];
else
   c = [c r(i:end)];
end

% Denormalize the coefficients.
c = c + (c > 0) - (c < 0);
for k = 1:n
   qi = 3*k - 2;
   c = wavepaste('h',c,s,k,wavecopy('h',c,s,k)*q(qi));
   c = wavepaste('v',c,s,k,wavecopy('v',c,s,k)*q(qi + 1));
   c = wavepaste('d',c,s,k,wavecopy('d',c,s,k)*q(qi + 2));
end
c = wavepaste('a',c,s,k,wavecopy('a',c,s,k)*q(qi + 3));

% Compute the inverse wavelet transform and level shift.
x = waveback(c,s,'jpeg9.7',n);
x = uint8(x + 128);
```

The principal difference between the wavelet-based JPEG 2000 system of Fig. 9.14 and the DCT-based JPEG system of Fig. 9.11 is the omission of the latter's subimage processing stages. Because wavelet transforms are both computationally efficient and inherently local (i.e., their basis functions are limited in duration), subdivision of the image into blocks is unnecessary. As you will see in the following example, the removal of the subdivision step eliminates the blocking artifact that characterizes DCT-based approximations at high compression ratios.

EXAMPLE 9.9: JPEG 2000 compression.

Figure 9.16 shows two JPEG 2000 approximations of the monochrome image in Figure 9.4(a). Figure 9.16(a) was reconstructed from an encoding that compressed the original image by 42 : 1. Fig. 9.16(b) was generated from an 88 : 1 encoding. The two results were obtained using a five-scale transform and implicit quantization with $\mu_0 = 8$ and $\varepsilon_0 = 8.5$ and 7, respectively. Because `im2jpeg2k` only approximates the JPEG 2000's bit-plane–oriented arithmetic coding, the compression rates just noted differ from those that would be obtained by a true JPEG 2000 encoder. In fact, the actual rates would increase by approximately a factor of 2.

Because the 42 : 1 compression of the results in the left column of Fig. 9.16 is identical to the compression achieved for the images in the right column of Fig. 9.13 (Example 9.8), Figs. 9.16(a), (c), and (e) can be compared—both qualitatively and quantitatively—to the JPEG results of Figs. 9.13(b), (d), and (f). A visual comparison reveals a noticeable decrease of error in the wavelet-based JPEG 2000 images. In fact, the rms error of the JPEG 2000–based result in Fig. 9.16(a) is 3.6 gray levels, as opposed to 4.4 gray levels for the corresponding transform-based JPEG result in Fig. 9.13(b). Besides decreasing reconstruc-



a b
c d
e f

FIGURE 9.16
Left column: JPEG 2000 approximations of Fig. 9.4 using five scales and implicit quantization with $\mu_0 = 8$ and $\varepsilon_0 = 8.5$. Right column: Similar results with $\varepsilon_0 = 7$.

9.6 VIDEO COMPRESSION

A *video* is a sequence of images, called *video frames*, in which each frame is a monochrome or full-color image. As might be expected, the redundancies introduced in Sections 9.2 though 9.4 are present in most video frames—and the compression methods previously examined, as well as the compression standards presented in Section 9.5, can be used to process the frames independently. In this section, we introduce a redundancy that can be exploited to increase the compression that individual frame processing would yield. Called *temporal redundancy*, it is caused by the correlations between pixels in adjacent frames.

In the material that follows, we present both the fundamentals of video compression and the principal Image Processing Toolbox functions that are used for the processing of image sequences—whether the sequences are time-based video sequences or spatial-based sequences like those generated in magnetic resonance imaging. Before continuing, however, we note that the uncompressed video sequences that are used in our examples are stored in *multiframe TIFF* files. A multiframe TIFF can hold a sequence of images that may be read *one at a time* using the following imread syntax

```
imread('filename.tif',idx)
```

where `idx` is the integer index of the frame in the sequence to be read. To write uncompressed frames to a multiframe TIFF file, the corresponding `imwrite` syntax is

```
imwrite(f,'filename','Compression','none','WriteMode',mode)
```

where `mode` is set to `'overwrite'` when writing the initial frame and to `'append'` when writing all other frames. Note that unlike `imread`, `imwrite` does not provide random access to the frames in a multiframe TIFF; frames must be written in the time order in which they occur.

REPRESENTING VIDEO IN MATLAB

There are two standard ways to represent video data in the MATLAB workspace. In the first, which is also the simplest, each frame of video is concatenated along the fourth dimension of a four dimensional array. The resulting array is called a MATLAB *image sequence* and its first two dimensions are the row and column dimensions of the concatenated frames. The third dimension is 1 for monochrone (or indexed) images and 3 for full-color images; the fourth dimension is the number of frames in the image sequence. Thus, the following commands read the first and last frames of the 16-frame multiframe TIFF, `'shuttle.tif'`, and build a simple two-frame $256 \times 480 \times 1 \times 2$ monochrome image sequence `s1`:

```
>> i = imread('shuttle.tif',1);
>> frames = size(imfinfo('shuttle.tif'),1);
>> s1 = uint8(zeros([size(i) 1 2]));
```


```
seq2tifs(s,'filename.tif')
```

where `s` is a MATLAB image sequence and `'filename.tif'` is a multiframe TIFF file. To perform similar conversions with MATLAB movies, use


```
m = tifs2movie('filename.tif')
```

and


```
movie2tifs(m,'filename.tif')
```

where `m` is MATLAB movie. Finally, to convert a multiframe TIFF to an *Advanced Video Interleave* (AVI) file for use with the *Windows Media Player* app, use `tifs2movie` in conjunction with MATLAB's `VideoWriter` function:

`VideoWriter`

For more information on the parameters used in `VideoWriter`, type `>> help VideoWriter`.

```
v = VideoWriter('filename.avi');
open(v);
writeVideo(v,tifs2movie('filename.tif'));
close(v);
```

where `'filename.tif'` is a multiframe TIFF and `'filename.avi'` is the name of the generated AVI file. To view a multiframe TIFF on the toolbox movie player, combine `tifs2movie` with function `implay`:

```
implay(tifs2movie('filename.tif'))
```

TEMPORAL REDUNDANCY AND MOTION COMPENSATION

Like spatial redundancies, which result from correlations between pixels that are near to one another in space, temporal redundancies are the result of correlations between pixels that are close to one another in time. As you will see in the following example, which parallels Example 9.5 of Section 9.3, both redundancies are addressed in much the same way.

EXAMPLE 9.10: Temporal redundancy.

Figure 9.19(a) shows the second frame of the multiframe TIFF whose first and last frames are depicted in Fig. 9.18. As was noted in Sections 9.2 and 9.3, the spatial and coding redundancies that are present in a conventional 8-bit representation of the frame can be removed through the use of Huffman and linear predictive coding:

```
>> f2 = imread('shuttle.tif',2);
>> ntrop(f2)
```

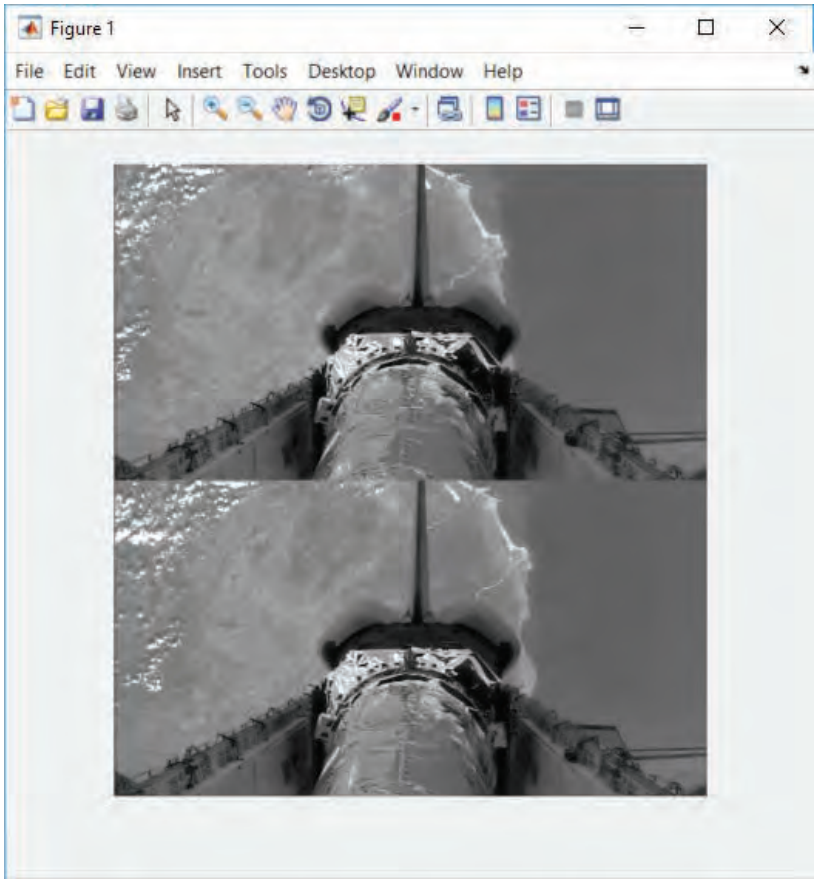
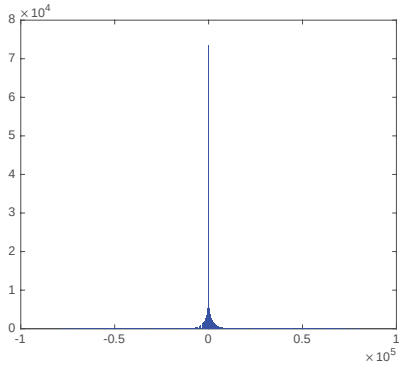


FIGURE 9.18
A montage of two video frames. (Original images courtesy of NASA.)



a b

FIGURE 9.19 (a) The second frame of a 16-frame video of the space shuttle in orbit around the Earth. The first and last frames are shown in Fig. 9.18. (b) The histogram of the prediction error resulting from the previous frame prediction in Example 9.9. (Original image courtesy of NASA.)

```
ans =
    6.3368
>> compare(imread('shuttle.tif',8),imread('ss2.tif',8))
ans =
   11.8650
>> compare(imread('shuttle.tif',16),imread('ss2.tif',16))
ans =
   14.2251
```

Note that `cv2tifs` (the decompression function) is almost 9 times faster than `tifs2cv` (the compression function)—only 0.6175 seconds as opposed to 4.8599 seconds. This is as should be expected, because the encoder not only performs an exhaustive search for the best motion vectors, (the decoder merely uses those vectors to generate predictions), but decodes the encoded prediction residuals as well. Note also that the rms errors of the reconstructed frames increase from only 6 gray levels for the first frame to almost 15 gray levels for the final frame. Figures 9.22(b) and (c) show an original and reconstructed frame in the middle of the video (i.e., at frame 8). With an rms error of about 12 gray levels, the loss of detail—particularly in the clouds in the upper left and the rivers on the right side of the landmass,—is clearly evident. Finally, we note that with a compression of 16.67 : 1, the motion compensated video uses only 6% of the memory required to store the original uncompressed multiframe TIFF.

Summary

The material in this chapter introduces the fundamentals of digital image compression through the removal of coding redundancy, spatial redundancy, temporal redundancy, and irrelevant information. MATLAB routines that attack each of these redundancies—and extend the Image Processing Toolbox—are developed. Both still frame and video coding considered. Finally, an overview of the popular JPEG and JPEG 2000 image compression standards is given. For additional information on the removal of image redundancies—both techniques that are not covered here and standards that address specific image subsets (like binary images)—see Chapter 8 of the fourth edition of *Digital Image Processing* by Gonzalez and Woods [2018].

MATLAB Projects

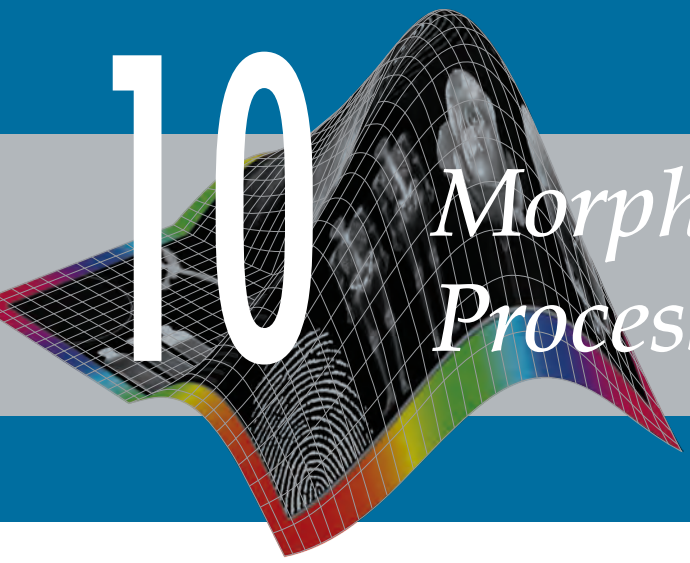
*Solutions to the projects marked with an asterisk * are in the DIPUM3E Student Support Package (consult the book web site). All your code must be documented so that typing `help` at the prompt, followed by the script or function name, gives enough detail for a user to be able to run it. Test the functionality of all your code thoroughly.*

9.1 If n_1 and n_2 denote the number of bits in two representations of the same image, the *relative data redundancy*, R, of the representation with n_1 bits is

$$R = \frac{n_1 - n_2}{n_1}$$

and can be computed using the compression ratio defined by Eq. (9-1) and returned by function `imratio` of Section 9.1. In the context of digital image compression, n_1 is normally the number of bits needed to represent an image as a 2-D array of intensity values.

(a) Write a function `r=redundancy(i1,i2)` that calls function `cr=imratio(i1,i2)` and uses the returned value, `cr`, to compute the relative data redundancy, `r`, of `i1` with respect to `i2`.



10 Morphological Image Processing

Morphology: relating to or concerned with form and structure.
Merriam-Webster Dictionary

The word *morphology* commonly denotes a branch of biology that deals with the form and structure of animals and plants. We use the same word here in the context of *mathematical morphology* as a tool for extracting image components that are useful in the representation and description of region shape, such as boundaries, skeletons, and the convex hull of a region. We are interested also in morphological techniques for pre- or postprocessing, such as morphological filtering, thinning, and pruning.

The material in this chapter begins a transition from image processing methods, whose inputs and outputs are images, to image analysis methods, whose outputs attempt to describe the contents of an input image. Morphology is a cornerstone of the mathematical set of tools underlying the development of techniques that extract "meaning" from an image. Other approaches are developed and applied in the remaining chapters of the book.

Function Developed in this Chapter:

- `conwaylaws` applies Conway's genetic laws to a single pixel and its 3×3 neighborhood. The function is based on Conway's Game of Life.

10.1 PRELIMINARIES

In this section we introduce some basic concepts from set theory and discuss the application of MATLAB's logical operators to binary images.

SOME BASIC CONCEPTS FROM SET THEORY

Let Z be the set of real integers. The sampling process used to generate digital images may be viewed as partitioning the xy-plane into a grid with the coordinates of the center of each grid being a pair of elements from the Cartesian product, Z^2.[†] In the terminology of set theory, a function $f(x, y)$ is said to be a *digital image* if the image coordinates, (x, y), are integers from Z^2 and f is a mapping that assigns an intensity value (that is, a real number from the set of real numbers, R) to each distinct pair of coordinates. If the elements of R are integers also (as is usually the case in this book), a digital image then becomes a two-dimensional function whose coordinates and amplitude (i.e., intensity) values are integers.

Let A be a set in Z^2, the elements of which are pixel coordinates (x, y). We denote the condition that $w = (x, y)$ is an element of A using the notation

$$w \in A \tag{10-1}$$

Similarly, if w is not an element of A, we write

$$w \notin A \tag{10-2}$$

A set B of pixel coordinates that satisfies a particular condition is written as

$$B = \{w \,|\, \text{condition}\} \tag{10-3}$$

For example, the set of all pixel coordinates that do not belong to set A, denoted A^c, is given by

$$A^c = \{w \,|\, w \notin A\} \tag{10-4}$$

This set is called the *complement* of A (see Fig. 10.1).

The *union* of two sets, A and B, denoted by

$$C = A \cup B \tag{10-5}$$

is the set of all elements that belong to A, to B, or to both. Similarly, the *intersection* of sets A and B, denoted by

$$C = A \cap B \tag{10-6}$$

is the set of all elements that belong to both A and B.

[†] The Cartesian product of a set of integers, Z, is the set of all ordered pairs of elements (z_i, z_j) with z_i and z_j being integers from Z. It is customary to denote the Cartesian product by Z^2.

a b c
d e f

FIGURE 10.3 (a) Binary image A. (b) Binary image B. (c) Complement A^c. (d) Union $A \cup B$. (e) Intersection $A \cap B$. (f) Set difference $A - B$.

10.2 DILATION AND EROSION

The operations of *dilation* and *erosion* are fundamental in morphological image processing. Many of the algorithms presented later in this chapter are based on these two operations.

DILATION

Dilation is an operation that "grows" or "thickens" objects in an image. The specific manner and extent of this thickening is controlled by a shape referred to as a *structuring element (SE)*. Figure 10.4 illustrates how dilation works. Figure 10.4(a) shows a binary image containing a rectangular object. Figure 10.4(b) is a structuring element—a five-pixel-long diagonal line in this case. Graphically, SEs can be represented either by a matrix of 0s and 1s or as a set of foreground (1-valued) pixels, as in Fig. 10.4(b). We use both representations interchangeably in this chapter. Regardless of the representation, the origin of the structuring element must be clearly identified.

a b
c
d

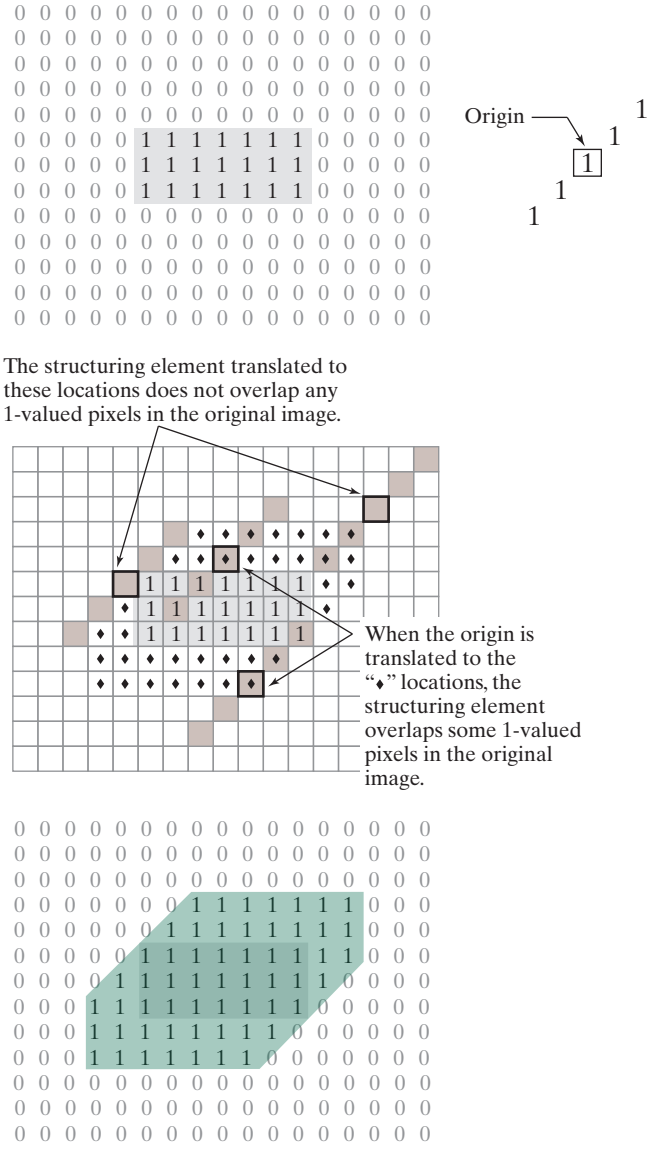


FIGURE 10.4 Illustration of dilation. (a) Original image with rectangular object. (b) Structuring element with five pixels arranged in a diagonal line. The origin, or center, of the structuring element is shown with a dark border. (c) Structuring element translated to several locations in the image. (d) Output image. The green region shows the 1-valued elements of the resulting dilation. The gray region shows the location of 1s in the original image.

The location of the origin in a structuring element is important. Changing the location of the defined origin generally changes the result of a morphological operation.

Figure 10.4(b) indicates the origin of the structuring element using a black box. Figure 10.4(c) depicts dilation as a process that translates the origin of the structuring element throughout the domain of the image and checks to see where the element overlaps 1-valued pixels. The output image [Fig. 10.4(d)] is 1 at each *location* of the *origin* of the structuring element such that the structuring element overlaps at least one 1-valued pixel in the input image.

imerode

```
>> E15 = imerode(A,se);
>> imshow(E15)
```

As Fig. 10.8(b) shows, these commands successfully removed the thin wires in the mask. Figure 10.8(c) shows what happens if we choose a structuring element that is too small:

```
>> se = strel('disk',5);
>> E5 = imerode(A,se);
>> imshow(E5)
```

Some of the wire leads were not removed in this case. Figure 10.8(d) shows what happens if we choose a structuring element that is too large:

```
>> E35 = imerode(A,strel('disk',35));
>> imshow(E35)
```

The wire leads were removed, but so were the border leads.

10.3 COMBINING DILATION AND EROSION

In image-processing applications, dilation and erosion are used most often in various combinations. In this section we consider three of the most common: opening, closing, and the hit-or-miss transformation. We also introduce lookup table operations.

OPENING AND CLOSING

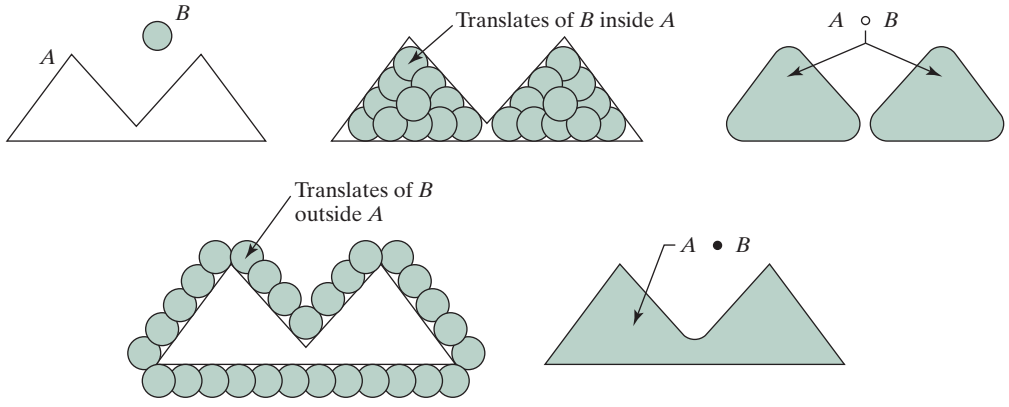
The *morphological opening* of A by B, denoted $A \circ B$, is defined as the erosion of A by B, followed by the dilation of the result by B:

$$A \circ B = (A \ominus B) \oplus B \tag{10-18}$$

An equivalent formulation of opening is

$$A \circ B = \cup\{(B)_z \mid (B)_z \subseteq A\} \tag{10-19}$$

where $\cup\{\cdot\}$ denotes the union of all sets inside the braces. This formulation has a simple geometric interpretation: $A \circ B$ is the union of all translations of B that fit *entirely* within A. Figure 10.9 illustrates this interpretation. Figure 10.9(a) shows a set A and disk-shaped structuring element B. Figure 10.9(b) shows some of the translations of B that fit entirely within A. The union of all such translations results in the two shaded regions in Fig. 10.9(c); these two regions are the complete opening. The white regions in this figure are areas where the structuring element could not fit completely within A and are therefore not part of the opening. Morphological



a b c
d e

FIGURE 10.9 Opening and closing as unions of translated structuring elements. (a) Set A and structuring element B. (b) Translations of B that fit entirely within set A. (c) The complete opening (shaded). (d) Translations of B outside the border of A. (e) The complete closing (shaded).

opening removes completely regions of an object that cannot contain the structuring element, smooths object contours, breaks thin connections [as in Fig. 10.9(c)], and removes thin protrusions.

The *morphological closing* of A by B, denoted $A \bullet B$, is a dilation followed by an erosion:

$$A \bullet B = (A \oplus B) \ominus B \tag{10-20}$$

Geometrically, $A \bullet B$ is the complement of the union of all translations of B that do not overlap A. Figure 10.9(d) illustrates several translations of B that do not overlap A. By taking the complement of the union of all such translations, we obtain the shaded region if Fig. 10.9(e), which is the complete closing. Like opening, morphological closing tends to smooth the contours of objects. Unlike opening, however, closing generally joins narrow breaks, fills long thin gulfs, and fills holes smaller than the structuring element.

Opening and closing are implemented by Toolbox functions `imopen` and `imclose`. These functions have the syntax forms

`imopen`

```
C = imopen(A,B)
```

and

`imclose`

```
C = imclose(A,B)
```

where, for now, A is a binary image and B is a matrix of 0s and 1s that specifies the structuring element. A `strel` object from Table 10.2 can be used instead of B.

Finally, we use logical indexing to eliminate the dilated pixels not belonging to the original binary image:

```
>> L3 = L2;
>> L3(~f) = 0;
>> g3 = label2rgb(L3,'jet',[.7 .7 .7],'shuffle');
>> figure, imshow(g3)
```

In Figure 10.19(e), the three object clusters have been identified and shown in three distinct colors.

10.5 MORPHOLOGICAL RECONSTRUCTION

Reconstruction is a morphological transformation involving two images and a structuring element, instead of a single image and structuring element. One image, the *marker*, is the starting point for the transformation. The other image, the *mask*, constrains the transformation. The SE used defines connectivity. In this section we use 8-connectivity (the default), which implies that B in the following discussion is a 3×3 matrix of 1s with the center defined at coordinates $(2, 2)$. In this section we work with binary images and wait until Section 10.6 to discuss grayscale reconstruction.

If G is a mask and F is a marker, the reconstruction of G from F, denoted $R_G(F)$, is defined by the following iterative procedure:

This definition of reconstruction is based on dilation. It is possible to define a similar operation using erosion. The results are duals of each other with respect to set complementation. These concepts are developed in detail in Gonzalez and Woods [2018].

1. Initialize h_1 to be the marker image, F.
2. Create a structuring element using `B = ones(3)`.
3. Repeat
$$h_{k+1} = (h_k \oplus B) \cap G \tag{10-22}$$
until $h_{k+1} = h_k$.
4. Let $R_G(F) = h_{k+1}$.

Marker F must be a subset of G—that is,

$$F \subseteq G$$

Figure 10.20 illustrates the preceding iterative procedure. Although this iterative formulation is useful conceptually, much faster computational algorithms exist. Toolbox function `imreconstruct` uses the "fast hybrid reconstruction" algorithm described in Vincent [1993]. The calling syntax for `imreconstruct` is

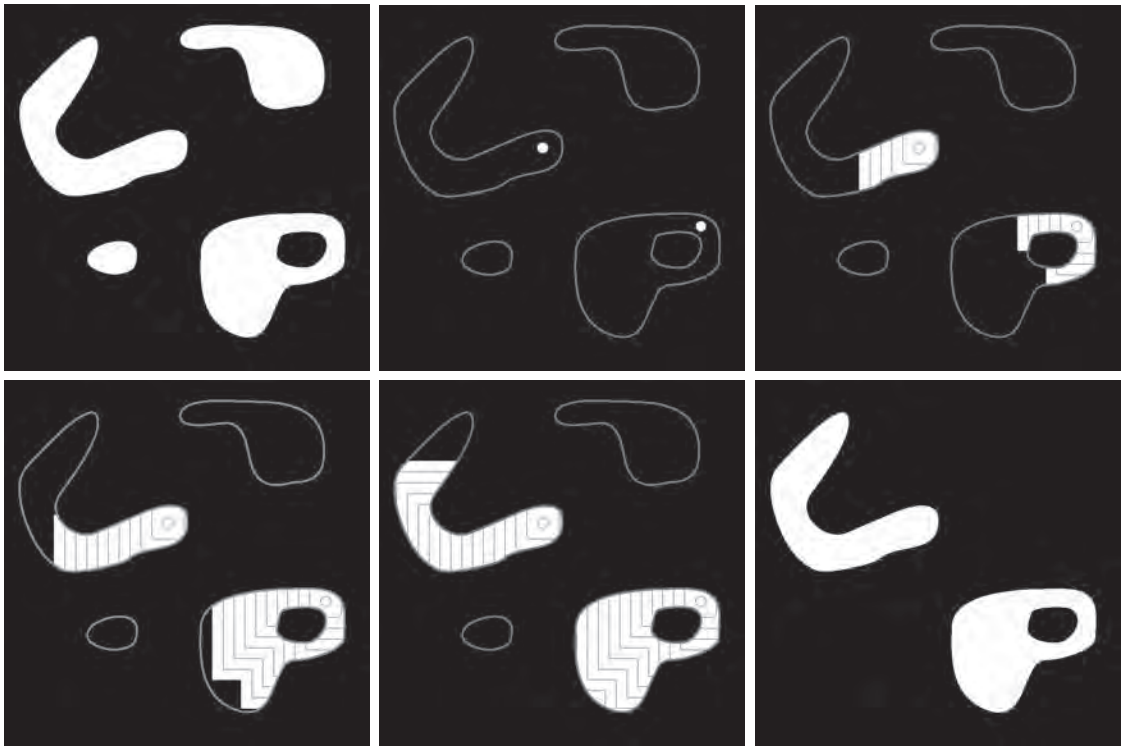
`imreconstruct`

```
out = imreconstruct(marker,mask,conn)
```

where `marker` and `mask` are as defined above and `conn` specifies the connectivity, which, in this book, is either 4 or 8 (the default) .

OPENING BY RECONSTRUCTION

In morphological opening, erosion typically removes small objects and the subsequent dilation tends to restore the shape of the objects that remain. However, the



a b c
d e f

FIGURE 10.20 Morphological reconstruction. (a) Original image (the mask). (b) Marker image. (c)–(e) Intermediate results after 100, 200, and 300 iterations, respectively. (f) Final result. (The outlines of the objects in the mask image are superimposed on (b)–(e) as visual references.)

accuracy of this restoration depends on the similarity between the shapes and the structuring element. The method discussed in this section, *opening by reconstruction*, restores the original shapes of the objects that remain after erosion. The opening by reconstruction of an image G using structuring element B is defined as $R_G(G \ominus B)$. In words, we see that opening by reconstruction uses the original image as a mask and the eroded image as a marker. Thus, for an image G with white objects on a black background, we perform opening by reconstruction with function `imreconstruct` using the command

```
>> or = imreconstruct(imerode(G,B),G)
```

where the erosion is carried out until only the white regions of interest remain.

The *closing by reconstruction* of an image G using structuring element B is the dual of opening by reconstruction in the sense that we perform the same operations as we do for opening by reconstruction (using the *complement* of G) and then

a b

FIGURE 10.28
(a) Opening-by-reconstruction. (b) Opening-by-reconstruction followed by closing-by-reconstruction.

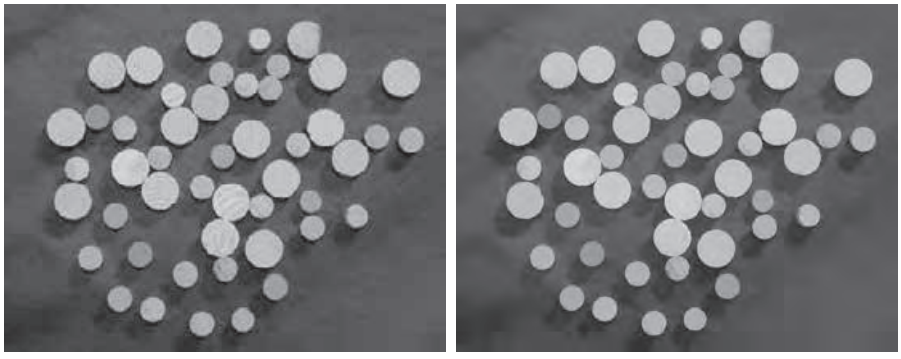


image in a reconstruction. The original image is used as the mask. Figure 10.28(a) shows an example of opening-by-reconstruction, obtained using the commands

```
>> f = imread('dowels.tif');
>> imshow(f)
>> se = strel('disk',5);
>> fe = imerode(f,se);
>> fobr = imreconstruct(fe,f);
```

Reconstruction can be used to clean up an image further by applying to it a *closing-by-reconstruction*. The technique is implemented by complementing an image, computing its opening-by-reconstruction, and then complementing the result. For example,

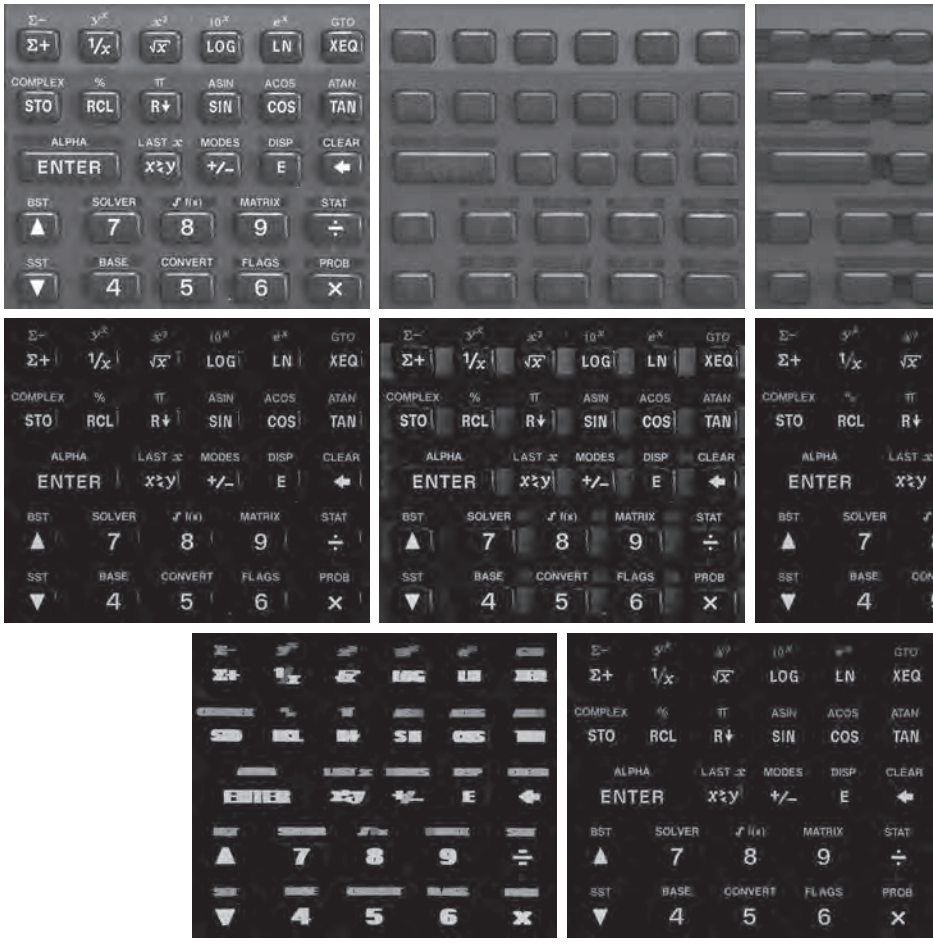
```
>> fobrc = imcomplement(fobr);
>> fobrce = imerode(fobrc,se);
>> fobrcbr = imcomplement(imreconstruct(fobrce, fobrc));
>> figure, imshow(fobrcbr)
```

Figure 10.28(b) shows the result of opening-by-reconstruction followed by closing-by-reconstruction. Compare it with the open-close filter and alternating sequential filter results in Fig. 10.24.

EXAMPLE 10.12: Using grayscale reconstruction to remove a complex background.

Our concluding example uses grayscale reconstruction in several steps. The objective is to isolate the text out of the image of calculator keys shown in Fig. 10.29(a). The first step is to suppress the horizontal reflections on the top of each key. To accomplish this, we use the fact that these reflections are wider than any single text character in the image. We perform opening-by-reconstruction using a structuring element that is a long horizontal line:

```
>> f = imread('calculator.tif');
>> f_obr = imreconstruct(imerode(f, ones(1,71)),f);
>> f_o = imopen(f,ones(1,71)); % For comparison.
```



a b c
d e f
g h

FIGURE 10.29 An application of grayscale reconstruction. (a) Original image. (b) Opening-by-reconstruction. (c) Opening. (d) Tophat-by-reconstruction. (e) Tophat. (f) Opening-by-reconstruction of (d) using a horizontal line. (g) Dilation of (f) using a horizontal line. (h) Final reconstruction result.

The opening-by-reconstruction (`f_obr`) is shown in Fig. 10.29(b). For comparison, Fig. 10.29(c) shows the standard opening (`f_o`). Opening-by-reconstruction did a better job of extracting the background between horizontally adjacent keys. Subtracting the opening-by-reconstruction from the original image is called *tophat-by-reconstruction* and is shown in Fig. 10.29(d):

```
>> f_thr = f - f_obr;
>> f_th = f - f_o;          % Or imtophat(f,ones(1,71))
```

Figure 10.29(e) shows the standard tophat computation (i.e., `f_th`).

Next, we suppress the vertical reflections on the right edges of the keys in Fig. 10.29(d). This is done by performing opening-by-reconstruction with a small horizontal line:

```
>> g_obr = imreconstruct(imerode(f_thr,ones(1,11)),f_thr);
```

In the result in Fig. 10.29(f), the vertical reflections are gone, but so are the thin, vertical-stroke characters, such as the slash on the percent symbol and the "I" in ASIN. We make use of the fact that the characters that have been suppressed in error are very close spatially to other characters still present by first performing a dilation [Fig. 10.29(g)],

```
>> g_obrd = imdilate(g_obr,ones(1,21));
```

followed by a final reconstruction with `f_thr` as the mask and `min(g_obrd,f_thr)` as the marker:

```
>> f2 = imreconstruct(min(g_obrd,f_thr),f_thr);
```

Figure 10.29(h) shows the final result. Note that the shading and reflections on the background and keys were removed successfully.

Summary

The morphological concepts and techniques introduced in this chapter constitute a powerful set of tools for extracting features from an image. The basic operators of erosion, dilation, and reconstruction—defined for both binary and grayscale image processing—can be used in combination to perform a wide variety of tasks. As shown in the following chapter, morphological techniques can be used for image segmentation. Moreover, they play an important role in algorithms for image description, as discussed in Chapter 13.

MATLAB Projects

*Solutions to the projects marked with an asterisk * are in the DIPUM3E Student Support Package (consult the book web site). All your code must be documented so that typing `help` at the prompt, followed by the script or function name, gives enough detail for a user to be able to run it. Test the functionality of all your code thoroughly.*

10.1 You are asked to design a system that performs erosion of binary images using rectangular structuring elements composed of all 1s. You are told that the system is to be implemented in a specialized processor whose only fast sliding neighborhood capability is ultra-fast 2D convolutions. The system can also perform other arithmetic and logical operations, but not on sliding neighborhoods.

(a) Use MATLAB's function `conv2` to write a custom function, `imerodeconv`, that simulates the system by using convolution to perform binary erosion. The inputs to your function are a binary image and an odd-sized rectangular structuring element of all 1s. The output should be the eroded image. For simplicity in your simulation, you may assume that the input image contains a single object. You may also ignore border effects.

(b)* Read image `letterA.tif` and use a 9×9 structuring element of 1s to erode it both with your function and with Toolbox function `imerode`. Compare the two results visually and numerically. They should be identical. (*Hint:* Note that the object in the image `letterA.tif` is black, whereas we expect it to be white for our definition of erosion to apply.)

11 Image Segmentation I

Edge Detection, Thresholding, and Region Detection

Divide and conquer.
Julius Caesar

The material in the previous chapter began a transition from image processing methods whose inputs and outputs are images, to methods in which the inputs are images but the outputs are attributes extracted from those images. Segmentation, the subdivision of images into regions, is the next step in that direction. Most of the segmentation algorithms in this chapter are based on one of two basic properties of image intensity values: discontinuity and similarity. In the first category, the approach is to partition an image into regions based on abrupt changes in intensity, such as edges. Approaches in the second category are based on partitioning an image into regions that are similar according to a set of predefined criteria. Thresholding, region growing, and region splitting and merging are examples of methods in this category. We show that improvements in segmentation performance can be achieved by combining methods from distinct categories, such as techniques in which edge detection is combined with thresholding. We discuss also image region segmentation using clustering and superpixels. We also introduce graph cuts, an approach ideally suited for extracting the principal regions of an image. This is followed by a discussion of image segmentation based on morphological watersheds, an approach that combines several of the attributes of segmentation based on the techniques presented in the first part of the chapter.

Functions Developed in this Chapter:

- `otsuthresh` computes Otsu's optimum threshold directly from an image histogram.
- `percentile2i` computes an intensity value given a percentile.
- `i2percentile` computes a percentile given an intensity value.
- `regiongrow` segments an image using region growing.
- `splitmerge` segments an image using a split-and-merge algorithm.
- `kmeansClustering` implements the "standard" *k*-means algorithm.
- `imcircle` creates a binary image of a circle.
- `imcolorcode` converts specified intensity values in a grayscale image to RGB colors.

11.1 BACKGROUND

Segmentation subdivides an image into disjoint regions. The level to which the subdivision is carried depends on the application. For example, in the automated inspection of electronic assemblies interest lies in analyzing images of the assemblies with the objective of determining the presence or absence of specific anomalies such as missing components or broken connection paths. There is no reason to carry segmentation past the level of detail required to identify these anomalies.

Segmentation of nontrivial images is one of the most difficult tasks in image processing. Segmentation accuracy determines the eventual success or failure of computerized image analysis procedures. For this reason, considerable care should be taken to improve the probability of an accurate segmentation. In some applications such as industrial inspection, at least some measure of control over the environment is possible at times. In others, as in remote sensing, user control over image acquisition is limited principally to the choice of imaging sensors.

The segmentation methods discussed in this chapter are based on one of two concepts: discontinuity and similarity. In the first category, the approach is to partition an image based on abrupt local changes in intensity values such as edges, that define boundaries between regions. The principal approaches in the second category are based on partitioning an image directly into regions whose pixels are similar according to a predefined set of criteria.

Let R represent the entire spatial region occupied by an image. We can express segmentation mathematically as a process that partitions R into n subregions that satisfy the following properties:

(a) $\bigcup_{i=1}^{n} R_i = R.$

(b) R_i is a connected set for $i = 0, 1, 2, \ldots, n.$

(c) $R_i \cap R_j = \varnothing$ for all valid values of i and j, $i \neq j.$

(d) $Q(R_i) = \text{TRUE}$ for $i = 0, 1, 2, \ldots, n.$

(e) $Q(R_i \cup R_j) = \text{FALSE}$ for all adjacent regions R_i and R_j.

where $Q(R_k)$ is a *logical predicate* defined over the points in set R_k and $\varnothing$ is the *null*, or *empty*, *set*. The symbols $\cup$ and $\cap$ denote *set union* and *intersection*, respectively. Two regions R_i and R_j are said to be *adjacent* if their union forms a connected set. If the set formed by the union of two regions is not connected, the regions are said to be *disjoint*.

Condition (a) indicates that the segmentation must be *complete*, in the sense that every pixel must be in a region. Condition (b) requires that points in a region be connected in some predefined sense (e.g., the points must be 8-connected). Condition (c) says that the regions must be *disjoint*. Condition (d) deals with the properties that must be satisfied by the pixels in a segmented region—for example, $Q(R_i) = \text{TRUE}$ if all pixels in R_i have the same intensity. Finally, condition (e) indicates that two adjacent regions R_i and R_j must be different in the sense of predicate Q. In general, Q can be a *compound predicate* such as: $Q(R_i) = \text{TRUE}$ if the average intensity of the pixels in region R_i is less than m_i AND if the standard

deviation of the intensity values in the region is greater than σ_i, where m_i and σ_i are specified constants.

In the sections that follow, we cover a number of segmentation approaches that are different in "philosophy," but which result in segmented images that satisfy the conditions just stated. For example, edge detection is based on intensity discontinuities, but its ultimate objective is to find region boundaries. We can think of connected edge pixels as forming disjoint regions and non-edge pixels as forming connected background regions. Ultimately, we can view segmentation as a *pixel-labeling* problem in which pixels are assigned to labeled classes, the union of which constitutes an image. Although segmentation is customarily treated as a separate topic, viewing it as a pixel classification problem casts it as a special case of the more general object recognition problem we will discuss in Chapter 14.

11.2 EDGE DETECTION

The central idea of edge detection for segmentation is that objects (regions) can be differentiated from the background by detecting their boundaries. Edge detection is an appropriate method to use when the principal features distinguishing these boundaries from the background are intensity discontinuities. Such discontinuities are detected using first- and second-order derivatives. The first-order derivative of choice in image processing is the gradient, defined in Section 3.4. We repeat the pertinent equations here for convenience. The *gradient* of a 2-D function, $f(x, y)$, is defined as the vector

We will discuss shortly how to compute g_x and g_y.

$$\nabla \mathbf{f} = \begin{bmatrix} g_x \\ g_y \end{bmatrix} = \begin{bmatrix} \dfrac{\partial f}{\partial x} \\ \dfrac{\partial f}{\partial y} \end{bmatrix} \tag{11-1}$$

The magnitude of this vector is

$$\begin{aligned} \nabla f &= \text{mag}(\nabla \mathbf{f}) = \left[g_x^2 + g_y^2 \right]^{\frac{1}{2}} \\ &= \left[(\partial f/\partial x)^2 + (\partial f/\partial y)^2 \right]^{\frac{1}{2}} \end{aligned} \tag{11-2}$$

This quantity is approximated sometimes by omitting the square-root operation:

$$\nabla f \approx g_x^2 + g_y^2 \tag{11-3}$$

or by using absolute values:

$$\nabla f \approx |g_x| + |g_y| \tag{11-4}$$

These approximations still behave as derivatives—they are zero in areas of constant intensity and their values are proportional to the degree of intensity change in areas

SOBEL EDGE DETECTOR

First-order derivatives are approximated digitally by differences. The *Sobel edge detector* computes the gradient by using the following discrete differences between rows and columns of a 3×3 neighborhood [see Fig. Fig. 11.2(a)]:

$$\begin{aligned}\nabla f &= \left[g_x^2 + g_y^2\right]^{\frac{1}{2}} \\ &= \left\{\left[(z_1 + 2z_2 + z_3) - (z_7 + 2z_8 + z_9)\right]^2 \right. \\ &\quad \left. + \left[(z_1 + 2z_4 + z_7) - (z_3 + 2z_6 + z_9)\right]^2\right\}^{\frac{1}{2}}\end{aligned} \tag{11-7}$$

See Gonzalez and Woods [2018] for an explanation of how the center weight of 2 provides image smoothing.

where the center pixel in each row or column is weighted by 2 to provide smoothing and the zs are intensities. We say that a pixel at location (x, y) is an *edge pixel* if $\nabla f \geq T$ at that location, where T is a specified threshold.

From the discussion in Section 3.4, we know that Sobel edge detection can be implemented by using function `imfilter` to filter an image `f` with one of the kernels in Fig. 11.2(b), filtering `f` again with the other kernel, squaring the values of each filtered image, adding the two results, and computing their square root. Similar comments apply to the second and third entries in Table 11.1. Function `edge` simply packages the preceding operations into one function call and adds other features such as accepting a threshold value or determining a threshold automatically. In addition, `edge` contains edge detection techniques that cannot be implemented directly using `imfilter`.

a b
c d

FIGURE 11.2 Some of the edge kernels used in function `edge`. The negative of these kernels are often used in practice. Either approach is correct, provided that it is used and interpreted consistently. See Project 11.3 for a set of kernels, called the *Kirsch compass kernels*, that can detect additional edge directions.

Image neighborhood

z_1	z_2	z_3
z_4	z_5	z_6
z_7	z_8	z_9

Sobel

1	2	1
0	0	0
−1	−2	−1

$g_x = (z_1 + 2z_2 + z_3) - (z_7 + 2z_8 + z_9)$

1	0	−1
2	0	−2
1	0	−1

$g_y = (z_1 + 2z_4 + z_7) - (z_3 + 2z_6 + z_9)$

Prewitt

1	1	1
0	0	0
−1	−1	−1

$g_x = (z_1 + z_2 + z_3) - (z_7 + z_8 + z_9)$

1	0	−1
1	0	−1
1	0	−1

$g_y = (z_1 + z_4 + z_7) - (z_3 + z_6 + z_9)$

Roberts

1	0
0	−1

$g_x = z_5 - z_9$

0	1
−1	0

$g_y = z_6 - z_8$

others in Fig. 11.2 due in part to its limited functionality (e.g., it is not symmetric and cannot be generalized to detect edges that are multiples of 45°). However, it still is used in hardware implementations where simplicity and speed are dominant factors. To prevent default edge thinning, include `'nothinning'` in the function call.

LAPLACIAN OF A GAUSSIAN (LOG) DETECTOR

Consider the Gaussian function

$$G(x, y) = e^{-\frac{x^2 + y^2}{2\sigma^2}}$$

where σ is the standard deviation. As you know from Chapter 3, this is a smoothing function that blurs an image when convolved with it. The degree of blurring is determined by the value of σ.

The Laplacian of this function (see Gonzalez and Woods [2018]) is

$$\begin{aligned} \nabla^2 G(x, y) &= \frac{\partial^2 G(x, x)}{\partial x^2} + \frac{\partial^2 G(x, x)}{\partial y^2} \\ &= \left[\frac{x^2 + y^2 - 2\sigma^2}{\sigma^4}\right] e^{-\frac{x^2 + y^2}{2\sigma^2}} \end{aligned} \tag{11-8}$$

For obvious reasons this function is called the *Laplacian of a Gaussian* (LoG). Because the second derivative is a linear operation, convolving (filtering) an image with $\nabla^2 G(x, y)$ is the same as convolving the image with the smoothing function first and then computing the Laplacian of the result. This is the key concept underlying the LoG detector. We convolve the image with $\nabla^2 G(x, y)$ knowing that it has two effects: It smooths the image (thus reducing noise) and it computes its Laplacian, which yields a double-edge image. Locating edges then consists of finding the zero crossings between the double edges.

The general calling syntax for the LoG detector is

```
[g,t] = edge(f,'log',T,sigma)
```

where `sigma` is the standard deviation and the other parameters are as we explained previously. The default value for `sigma` is 2. As before, function `edge` ignores any edges weaker than `T`. If `T` is not provided (or is empty, `[]`), `edge` chooses the threshold automatically. Setting `T` to 0 produces edges that are *closed contours*, a familiar characteristic of the LoG method.

ZERO-CROSSINGS DETECTOR

This detector is based on the same concept as the LoG method, but the convolution is carried out using a specified kernel, `h`. The calling syntax is

```
[g,t] = edge(f,'zerocross',T,h)
```

The other parameters are as explained for the LoG detector.

CANNY EDGE DETECTOR

The Canny detector (Canny [1986]) is the most powerful edge detector in function `edge`. The method can be summarized as follows:

1. The image is smoothed using a Gaussian filter with a specified standard deviation, σ, to reduce noise.
2. The local gradient $[g_x^2 + g_y^2]^{\frac{1}{2}}$ and edge direction $\tan^{-1}(g_y/g_x)$ are computed at each point. Any of the first three techniques in Table 11.1 can be used to compute the derivatives. An *edge point* is defined to be a point whose strength is locally maximum in the direction of the gradient.
3. The edge points determined in (2) produce *ridges* in the gradient magnitude image. The algorithm then tracks along the top of the ridges and sets to zero all pixels that are not actually on the ridge top, giving a thin line in the output (this process is called *nonmaximal suppression*). The ridge pixels are then thresholded by *hysteresis thresholding*, which is based on using two thresholds, T_1 and T_2, with $T_1 < T_2$. Ridge pixels with values greater than T_2 are said to be "strong" edge pixels; ridge pixels with values between T_1 and T_2 are said to be "weak" edge pixels; and ridge pixels with values less than T_1 are not considered to be edge pixels.
4. Finally, the algorithm performs edge linking by incorporating into an edge the weak pixels that are 8-connected to the strong pixels on that edge.

The syntax for the Canny edge detector is

```
[g,t] = edge(f,'canny',T,sigma)
```

where `T = [T1,T2]` is a vector containing the threshold values explained in step 3. Both values must be in the range [0,1]. If a scalar is specified, it is used as the high threshold and the low threshold is computed as `0.4*high_threshold`. Finally, `sigma` is the standard deviation of the smoothing filter; it defaults to `sqrt(2)`. If `t` is included in the output argument, it is as a two-element vector containing the two threshold values used by the algorithm. The rest of the syntax is as explained for the other methods, including the automatic computation of thresholds if `T` is not supplied.

APPROXIMATE CANNY EDGE DETECTOR

This is an approximate version of the Canny edge detection method that provides faster execution time at the expense of less precise detection. The syntax is

```
g = edge(f,'approxcanny',T)
```

where `T` is as explained for the Canny detector. Unlike all syntax forms discussed thus far, this syntax for `edge` does not output the threshold `t`.

```
>> t1
t1 =
    0.0373
```

We had to increase the threshold to 0.10 to get a result in which the strong vertical edges are predominant. We obtained Fig. 11.4(c) using the following commands:

```
>> gv2 = edge(f,'sobel',0.10,'vertical','nothinning');
>> figure, imshow(gv2) % Fig. 11.4(c).
```

Using the same value of `T` in the commands

```
>> gboth = edge(f,'sobel',0.10,,'nothinning');
>> figure, imshow(gboth) % Fig. 11.4(d).
```

resulted in Fig. 11.4(d), which shows predominantly vertical and horizontal edges.

Function `edge` does not compute Sobel edges at ±45°. To compute such edges we use the kernels in Fig. 11.3 and function `imfilter`. For example, we generated Fig. 11.4(e) using the commands

```
>> wpos45 = [0 1 2; -1 0 1; -2 -1 0]; % From Fig. 11.3(a).
>> gpos45 = imfilter(f,wpos45,'replicate');
>> gpos45 = gpos45 >= 0.4*max(abs(gpos45(:)));
>> figure, imshow(gpos45) % Fig. 11.4(e)
```

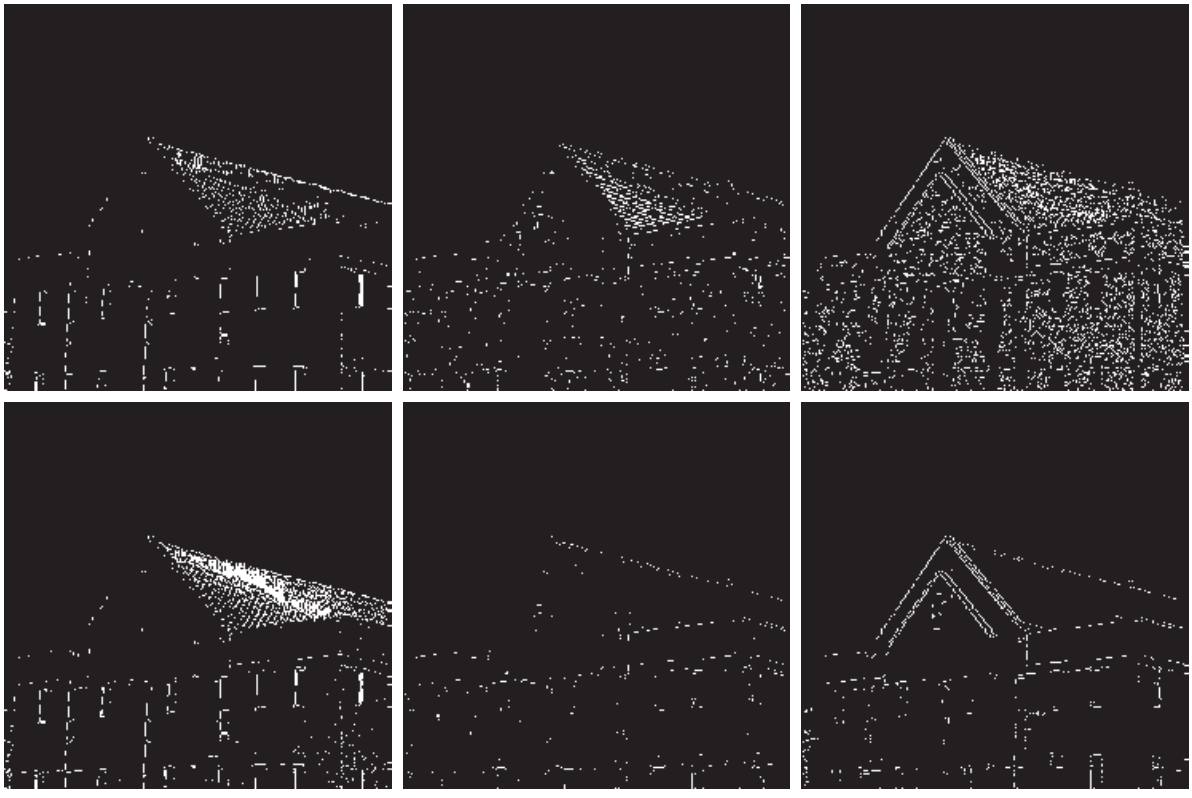
The third step thresholded the image after it was filtered. To simplify threshold selection, we used a fraction of the maximum absolute value in the filtered image. In this case using 40% of the maximum value extracted the edges oriented predominantly at +45°. A similar set of commands using the same threshold and the kernel in Fig. 11.3(b) resulted in Fig. 11.4(f), whose principal edge is in the −45° direction.

EXAMPLE 11.2: Comparison of the Sobel, LoG, and Canny edge detectors.

In this example we compare the relative performance of the Sobel, LoG, and Canny edge detectors. Our objective is to produce a clean *edge map* by extracting the principal edge features of the building image in Fig. 11.4(a), while reducing "irrelevant" detail such as the fine texture in the brick walls and tile roof. The principal features of interest in this discussion are the edges forming the building corners, the windows, the structure framing the entrance, the entrance itself, the roof line, and the concrete band surrounding the building about two-thirds of the distance above ground level.

The top row in Fig. 11.5 shows the edge images obtained using the default syntax for the `'sobel'`, `'log'`, and `'canny'` options:

```
>> f = tofloat(imread('building.tif'));
>> [gSobel_default,ts] = edge(f,'sobel');
>> imshow(gSobel_default) % Fig. 11.5(a).
>> [gLoG_default,tlog] = edge(f,'log');
>> figure, imshow(gLoG_default) % Fig. 11.5(b).
>> [gCanny_default,tc] = edge(f,'canny');
>> figure, imshow(gCanny_default) % Fig. 11.5(c).
```



a b c
d e f

FIGURE 11.5 Top row: Default results for the Sobel, LoG, and Canny edge detectors. Bottom row: Results obtained interactively to bring out the principal features in the original image of Fig. 11.4(a), while reducing irrelevant detail. The Canny edge detector produced the best result.

The defaults values of `sigma` for the `'log'` and `'canny'` options are `2.0` and `sqrt(2)`, respectively. The values of the thresholds in the output arguments of the preceding computations were `ts = 0.074`, `tlog = 0.0020`, and `tc = [0.019, 0.047]`. None of the default results shown in the first row of Fig. 11.5, especially the `'log'` and `'canny'` images, came close to meeting our objective of producing clean edge maps.

Starting with the default values, we varied the parameters in each option interactively to bring out the principal features mentioned earlier, while reducing irrelevant detail as much as possible. We obtained the results in the bottom row of Fig. 11.5 using the following commands:

```
>> gSobel_best = edge(f,'sobel',0.05); % Fig. 11.5(d).
>> gLoG_best = edge(f,'log',0.003,2.25); % Fig. 11.5(e).
>> gCanny_best = edge(f,'canny',[0.04 0.16],1.5); % Fig. 11.5(f).
```

As Fig. 11.5(d) shows, the Sobel result deviated even more from our objective when we tried to detect both edges of the concrete band and the left edge of the entrance. The LoG result in Fig. 11.5(e) is

11.3 THRESHOLDING

Because of its intuitive properties and simplicity, image thresholding enjoys a central position in applications of image segmentation. We introduced basic thresholding in Section 2.7 and we have used it in various discussions in the preceding chapters. In this section, we discuss ways of choosing the threshold value automatically and present a method for varying the threshold based on local image properties.

FOUNDATION

Suppose that the intensity histogram in Fig. 11.10(a) corresponds to an image, $f(x, y)$, composed of light objects on a dark background, in which objects and background pixels have intensity levels grouped into two dominant modes. One obvious way to extract the objects from the background is to select a threshold, T, that separates these modes. Then, any image point (x, y) with the property $f(x, y) > T$ is called an *object* (or *foreground*) *point*; otherwise, the point is called a *background* point (the reverse holds for dark objects on a light background). The thresholded (binary) image, $g(x, y)$, is defined as

We use the terms *object points* and *foreground points* interchangeably.

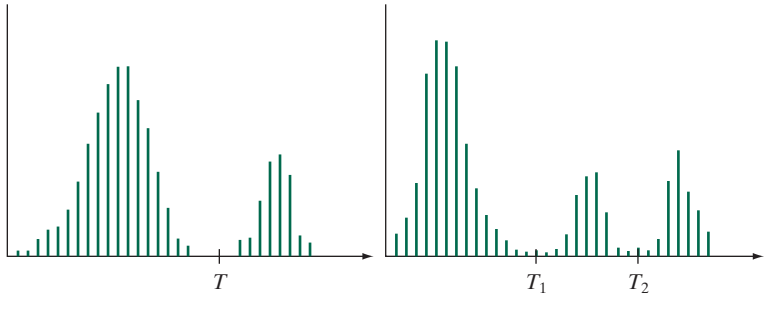
$$g(x, y) = \begin{cases} a & \text{if } f(x, y) > T \\ b & \text{if } f(x, y) \le T \end{cases} \tag{11-10}$$

Pixels labeled a correspond to objects and pixels labeled b correspond to the background. Typically, $a = 1$ (white) and $b = 0$ (black) by convention, but any two distinct values are acceptable, provided that you are consistent.

When T is a constant applicable over an entire image, the preceding equation is referred to as *global thresholding*. When the value of T at any point (x, y) in an image depends on properties of a neighborhood of (x, y) (e.g., the neighborhood average intensity), we use the term *variable thresholding*. The terms *adaptive*, *local*, or *regional thresholding* are used also to denote variable thresholding. If T depends on the spatial coordinates themselves, variable thresholding is often referred to as *dynamic thresholding*. Use of these terms is not universal and you are likely to see them used interchangeably in the literature on image processing.

a b

FIGURE 11.10 Intensity histograms that can be partitioned (a) by a single threshold and (b) by two thresholds. These are *bimodal* and *trimodal* histograms, respectively.



Toolbox function `graythresh` computes *Otsu's threshold*. It's syntax is

`graythresh`

```
[T,SM] = graythresh(f)
```

where `f` is the input image, `T` is the resulting Otsu threshold normalized to the range [0, 1], and `SM` is the separability measure defined in Eq. (11-19). After the threshold is computed, the image is segmented using function `imbinarize`, as explained in the previous section.

EXAMPLE 11.6: Image segmentation using Otsu's method.

We begin by examining the performance of Otsu's method when applied to the fingerprint image from Example 11.5:

```
>> f = imread('fingerprint.tif');
>> [T,SM] = graythresh(f);
>> T
T =
    0.4902
>> SM
SM =
    0.9437
>> T*255
ans =
    125
```

This threshold has nearly the same value as the threshold obtained using the basic global thresholding algorithm, so we would expect the same segmentation result. The high value of `SM` indicates a high degree of separability of the intensity values into two classes.

Polymersomes are cells artificially engineered using polymers. Polymersomes are invisible to the human immune system and can be used, for example, to deliver medication to targeted regions of the body.

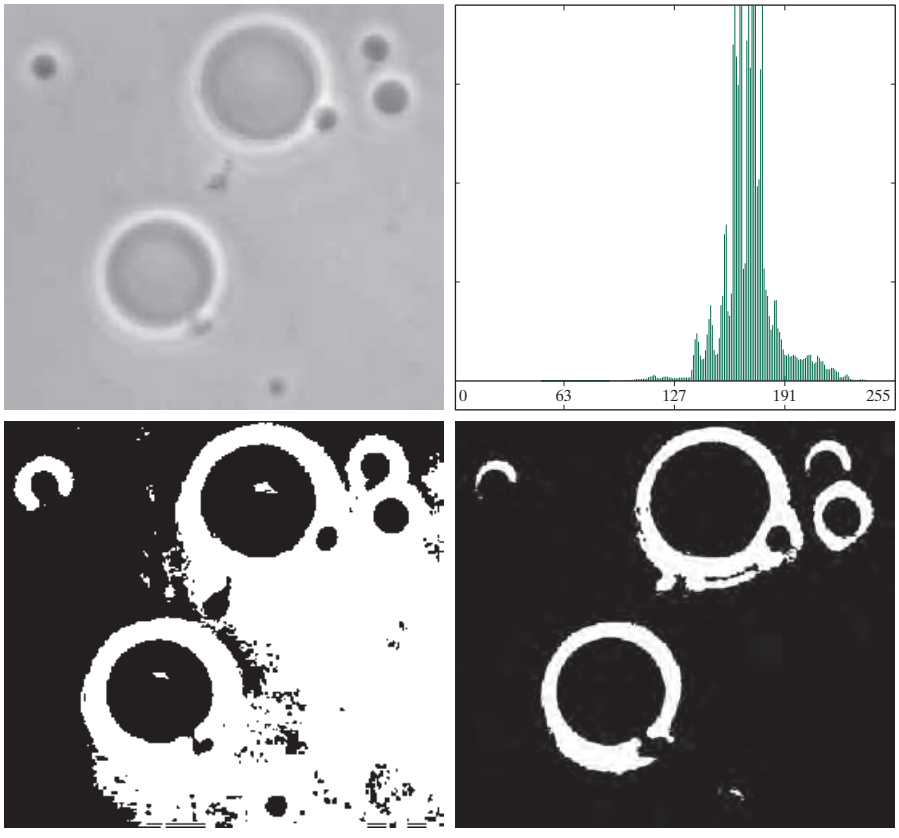
Figure 11.12(a) presents us with a more difficult segmentation task. This is an image of polymersome cells and our objective is to segment from the background the boundaries of the cells, which are the brightest regions in the image. The image histogram, shown Fig. 11.12(b), is far from bimodal so we would expect the simple algorithm from the last section to have difficulty in achieving a suitable segmentation. The image in Fig. 11.12(c) confirms this. It was obtained using the same procedure we used to obtain Fig. 11.11(c). The algorithm converged in one iteration and yielded a threshold, `T`, equal to 169.4:

```
>> f = imread('polymercell.tif');
>> figure, imshow(f) % Fig. 11.12(a).
>> figure, imhist(f) % Fig. 11.12(b).
>> g = imbinarize(f,169.4/255);
>> figure, imshow(g) % Fig. 11.12(c).
```

As Fig. 11.12(c) shows, the segmentation was unsuccessful.

Next we segment the image using Otsu's method:

```
>> [T,SM] = graythresh(f);
>> SM
```



a b
c d

FIGURE 11.12
(a) Image image of polymersome cells. (b) Histogram (high values were clipped to highlight details in the lower values). (c) Segmentation result using the basic global algorithm. (d) Result obtained using Otsu's method. (Original image courtesy of Professor Daniel A. Hammer, the University of Pennsylvania.)

```
SM =
    0.4662
>> T*255
ans =
   181
>> g = imbinarize(f,T);
>> figure, imshow(g) % Fig. 11.12(d).
```

As Fig. 11.12(d) shows, the segmentation using Otsu's method was effective. The borders of the polymersome cells were extracted from the background with reasonable accuracy, despite the relatively low value of the separability measure.

All the parameters of the between-class variance are based on the image histogram. As you will see shortly, there are applications in which it is advantageous to be able to compute Otsu's threshold using a given image histogram, rather than having to compute it from the image as function `graythresh` does. The following custom function computes `T` and `SM` directly from a given image histogram.

We can enhance the power of local thresholding significantly by adding *logical predicates* to the method. For example, we can define local thresholding in terms of a logical AND as follows:

$$g(x,y) = \begin{cases} 1 & \text{if } f(x,y) > a\sigma_{S_{xy}} \text{ AND } f(x,y) > bm \\ 0 & \text{otherwise} \end{cases} \tag{11-28}$$

where m is either the local or the global mean. You will write and test a local thresholding function based on these concepts in Project 11.6.

11.4 REGION-BASED SEGMENTATION

In Section 11.2 we performed image segmentation based on intensity discontinuities and in Section 11.3 we did it by comparing pixel values against one or more thresholds. In this section we segment an image into regions based on properties of the regions themselves.

REGION GROWING

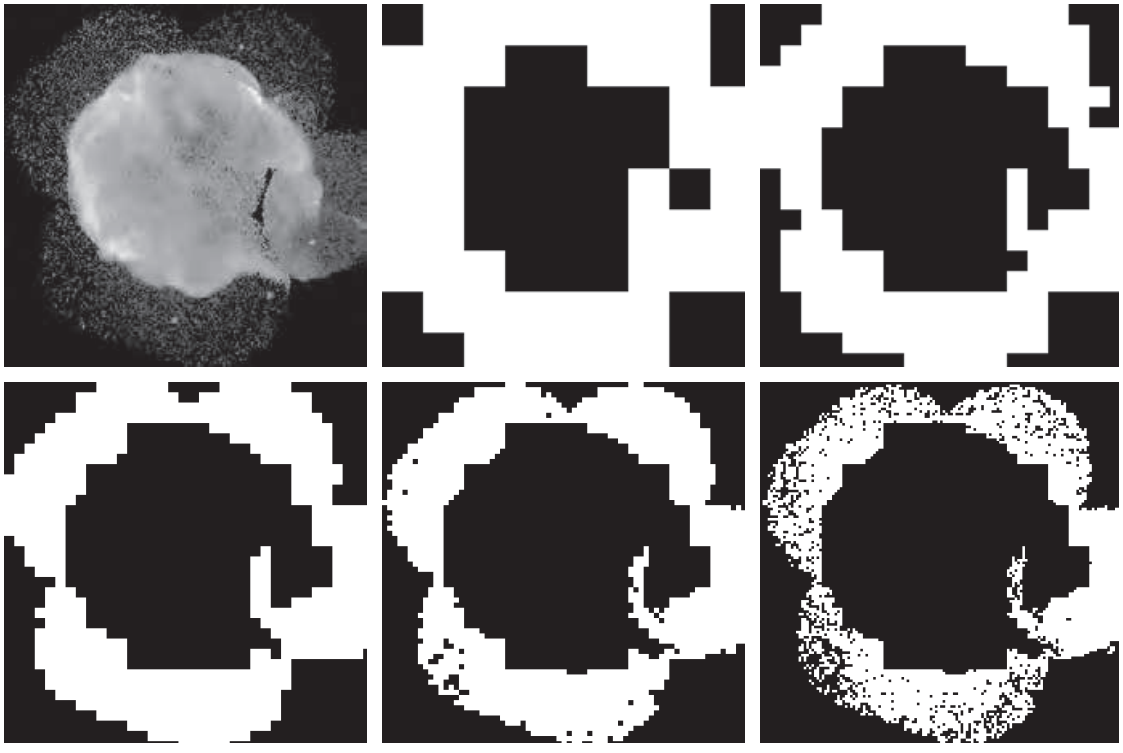
As its name implies, *region growing* is a procedure that groups pixels or subregions into larger regions based on predefined criteria for growth. The approach is to start with a set of *seed points* and from these grow regions by appending to each seed those neighboring pixels that have predefined properties similar to the seed, such as specific ranges of gray level or color.

Selecting a set of one or more seed points often can be based on the nature of the problem, as we will show later in Example 11.10. When a priori information is not available, one procedure is to compute at every pixel the same set of properties that ultimately will be used to assign pixels to regions during the growing process. If the result of these computations shows clusters of values, the centroids of these clusters can be used as seeds.

The selection of similarity criteria depends on the application. For example, the analysis of land-use satellite imagery depends heavily on the use of multispectral bands such as color and infrared bands. This problem would be significantly more difficult, or even impossible, to tackle without the inherent information available in those bands. When the images are monochrome, region analysis must be carried out with a set of descriptors based on intensity levels (such as moments of the intensity histogram) and spatial properties (such as connectivity). We will discuss descriptors for region characterization in Chapter 13.

Using intensity alone can yield misleading results if connectivity (adjacency) information is not used in the region-growing process. For example, visualize a random spatial arrangement of pixels with only three distinct intensity values. Grouping pixels with the same intensity level to form a "region" without taking connectivity into consideration would yield a segmentation result that is meaningless in the context of this discussion.

Another problem in region growing is the formulation of a stopping rule. Growing a region should stop when no more pixels satisfy the criteria for inclusion in a region.



a b c
d e f

FIGURE 11.22 Image segmentation using a split-and-merge algorithm. (a) Original image of size 566×566 pixels. (b) through (f) Results of segmentation using function `splitmerge` with values of `mindim` equal to 64, 32, 16, 8, and 4, respectively. (Original image courtesy of NASA.)

would be the best choice because it is the solid region with the most detail. An important aspect of the method just illustrated is its ability to "capture" in function `predicate` information about a problem domain that can help in segmentation. In the next chapter we will study segmentation methods that can give an even better segmentation of Fig. 11.22(a).

REGION SEGMENTATION USING K-MEANS CLUSTERING

The objective of clustering is to partition a set, Z, of vector observations into a specified number, k, of clusters. In *k-means clustering*, each observation is assigned to the cluster whose mean is nearest. A *k-means algorithm* is an iterative procedure that successively refines the means until convergence is achieved.

Let $Z = \{\mathbf{z}_1, \mathbf{z}_2, \ldots, \mathbf{z}_N\}$ denote a set of n-dimensional *vector observations*, often called *samples*:

This algorithm is known to converge to a local minimum in a finite number of iterations. However, it is not guaranteed to yield the global minimum required to minimize Eq. (11-30). The result at convergence in general depends on the initial values chosen for $\mathbf{m}_i$. An approach used frequently is to specify the initial means as k randomly chosen samples from the given sample set and to run the algorithm several times, using a new random set of initial samples each time. This is to test the "stability" of the solution. Topics related to the best way to initialize a k-means algorithm and to reduce the number of computations are numerous, but the "nuances" of these topics are beyond the scope of this discussion. When implemented in MATLAB, the key approach to speed-up performance of a k-means algorithm is to vectorize as many operations as possible, as you will learn by examining the custom function `kmeansClustering` introduced below.

In general data analysis, one of the principal applications of a k-means algorithm is to determine if a given set of observations tends to cluster about a set of clusters that is much smaller than the total number of observations. That is, the focus is to "discover" k. In image segmentation, k will determine the number of segmented regions, so k typically is specified and the objective is to determine if the chosen value results in a meaningful number of regions.

The MATLAB *Statistics and Machine Learning Toolbox* has a function called `kmeans` that implements the algorithm just discussed, with a number of embellishments. If you do not have that toolbox installed, you can use custom function `kmeansClustering` to segment an image using k-means. The syntax is


```
[L,C] = kmeansClustering(Z,k,Linit,Cinit)
```

where `Z` is a matrix whose rows are the sample vectors, `k` is the number of desired cluster centers, `C` is a matrix whose rows are the resulting cluster centers, and `L` is a vector containing a cluster label for each row of `Z`. If `Linit` and `Cinit` are provided, the algorithm starts with them. Otherwise, it generates `k` initial cluster centers by randomly choosing `k` rows from `Z`. The code is in your Support Package.

EXAMPLE 11.12: Image segmentation using k-means clustering.

Figure 11.23(a) is a 688×688 grayscale image and Fig. 11.23(b) resulted from the following commands:

```
>> f = im2double(imread('book-cover.tif'));
>> figure, imshow(f) % Fig. 11.23(a).
>> k = 3;
>> % Working only with intensities, so Z [Z = f(:)] only has one column.
>> [L,C] = kmeansClustering(f(:),k);
>> % Assign the corresponding label to each element of f to
>> % produce the segmented image.
>> fseg = zeros(size(f));
>> for i = 1:k
    fseg(L == i) = i;
end
>> % Scale the result to the full intensity scale.
>> fseg = intensityScaling(fseg);
```

a b

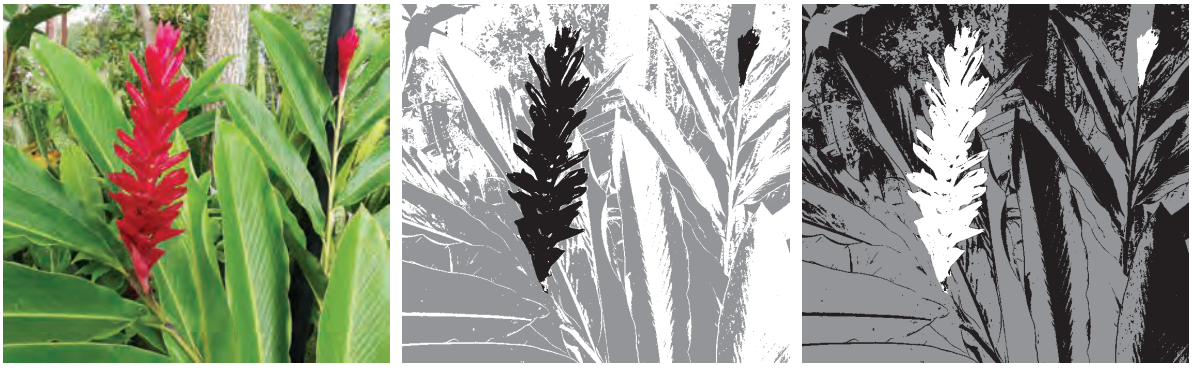
FIGURE 11.23
(a) Grayscale image. (b) *k*-means segmentation using $k = 3$.


```
>> figure, imshow(fseg) % Fig. 11.23(b).
```

As the result in Fig. 11.23(b) shows, the image was segmented into regions with three separate intensities. For example, all the dark tones in the original image show in white in the segmented image. Similarly, the gray tones remained gray and the white tones are shown in black. The particular shades shown depend on the initial cluster centers and are not important. What is important is that the original image was partitioned into consistent and meaningful regions.

Figure 11.24(a) shows an RGB image of size $693 \times 750 \times 3$ from which we want to extract the regions corresponding to the large and small red flowers. Figure 11.24(b) resulted from the following commands:

```
>> f = im2double(imread('flowers-red.tif'));
>> figure, imshow(f) % Fig. 11.24(a).
>> % Extract the RGB component images.
>> R = f(:,:,1);
>> G = f(:,:,2);
```



a b c

FIGURE 11.24 (a) RGB color image. (b) *k*-means segmentation using $k = 3$. (c) Complement of (b). The objects of interest in this example are the two red flowers, which were segmented properly by the *k*-means algorithm.

```
>> B = f(:,:,3);
>> % Form Z so that each z corresponds to one RGB pixel (triplet).
>> Z = [R(:) G(:) B(:)];
>> % Interested three regions: red flowers, green vegetation, and
>> % background.
>> k = 3;
>> [L,C] = kmeansClustering(Z,k);
>> % The segmented image is gray scale. We could code each region with a different
>> % color, but do not need that here because the number of regions is small.
>> fseg = zeros(size(f,1),size(f,2));
>> % Assign the labels to the pixels.
>> for j = 1:k
     fseg(L == j) = j;
end
>> fseg = intensityScaling(fseg);
>> figure, imshow(fseg) % Fig. 11.24(b).
>> % The regions corresponding to the flowers are shown in black.
>> % We can make them more visible by using the image complement.
>> fseg = imcomplement(fseg);
>> figure, imshow(fseg) % Fig. 11.24(c).
```

The final result depends on the initial clusters, so you may need to run this code more than once to get the same results we did.

REGION SEGMENTATION USING SUPERPIXELS

The idea behind superpixels is to replace the standard pixel grid by grouping pixels into primitive regions that are more perceptually meaningful than individual pixels. The objectives are to lessen computational load and to improve the performance of segmentation algorithms by reducing irrelevant detail.

To illustrate the concept, Fig. 11.25(a) shows an image of size 600×800 (480,000) pixels containing various levels of detail that could be described verbally as: "This is an image of two large carvings in the foreground and at least three smaller carvings

a b c

FIGURE 11.25 (a) Image of size 600×800 (480,000) pixels. (b) Image composed of 3,000 superpixels. The boundaries between superpixels (in white) are superimposed on the superpixel image for reference—the boundaries are not part of the data. (c) Superpixel image. (Original image courtesy of the U.S. National Park Services.)

```
>> % Show overlay on superpixel image.
>> mask = boundarymask(Lsp);
>> fSPO = imoverlay(fSP,mask,'w');
>> figure, imshow(fSPO) % Fig. 11.28(b).

>> % Show the superpixel image.
>> figure, imshow(fSP) % Fig. 11.28(c).
```

Figure 11.28(c) shows that a 100 superpixel image retained all the important regions and Fig. 11.28(b) shows how well the flower superpixels captured the two regions of interest. All that remains is to segment the color superpixel image using *k*-means clustering. We use the same approach as in Example 11.12:

```
>> % Apply k-means algorithm:
>> % Extract the RGB component images first.
>> R = fSP(:,:,1);
>> G = fSP(:,:,2);
>> B = fSP(:,:,3);
>> % Form Z so that each z corresponds to an RGB pixel.
>> Z = [R(:) G(:) B(:)];
>> % We are interested in three regions: red flowers, green vegetation,
>> % and background.
>> k = 3;
>> [L,C] = kmeansClustering(Z,k);
>> % The segmented image is grayscale. We could code each region with a different
>> % color but that is not needed here because the number of regions is small.
>> fSPseg = zeros(size(f,1),size(f,2));
>> % Assign the labels to the pixels.
>> for j = 1:k
       fSPseg(L == j) = j;
end

>> % Scale the intensities and show the result.
>> fSPsegS = intensityScaling(fSPseg);
>> figure, imshow(fSPsegS) % Fig. 11.28(d).

>> % Show only the segmented flowers against a black background. The
>> % flowers show white in the image, so they correspond to i = 3.
>> % (Running this experiment again could produce a different number
>> % because we start each time with a different, random set of seeds.)
>> imFlowers = fSPseg == 3;
>> figure, imshow(imFlowers) % Fig. 11.28(e).
```

As Fig. 11.28(e) shows, we obtained the same basic regions as in Example 11.12. The resolution in Fig. 11.28(e) is lower because we worked with only 100 superpixels, as opposed to 519,750 pixels.

IMAGE SEGMENTATION USING GABOR FILTERS

Although image texture is a topic in Chapter 13, we discuss in this section a texture extraction approach that is used directly for region segmentation. Intuitively, we often think of texture as periodic patterns and we know from Chapter 4 that periodicity gives rise to distinct burst of energy in the frequency domain. Studies of animal and human vision suggest a model of texture detection that incorporates

Figure 11.32 shows the results. The two images in Fig. 11.32(b) are the results corresponding to the `[wavelength, direction]` pairs, $[\lambda, \theta]$ for $[2, 0]$ (left) and $[8, 0]$ (right). Because the orientation of the two kernels is vertical, we expect the two results to be stronger for vertical objects. This is the case, as you can see in Fig. 11.32(b). The result on the left is much sharper than the result on the right, indicating that a wavelength of 2 is a much better match than a wavelength of 8 for the vertical components of this image. Another way of looking at this is that a wavelength of 2 provides better *discrimination* than a wavelength of 8. This is true also in Figs. 11.32(c) and (d) for the orientations at 45° and 90°, but looking at the three images on the left, we note that the response of the 45° kernels is much weaker than the other two. If you think of the principal "texture" of this image as being the repetition of vertical and horizontal edges, then the results in Fig. 11.32 show that Gabor filtering can be used to discriminate between image regions based on their texture content. We show next how this concept can be used in segmentation.

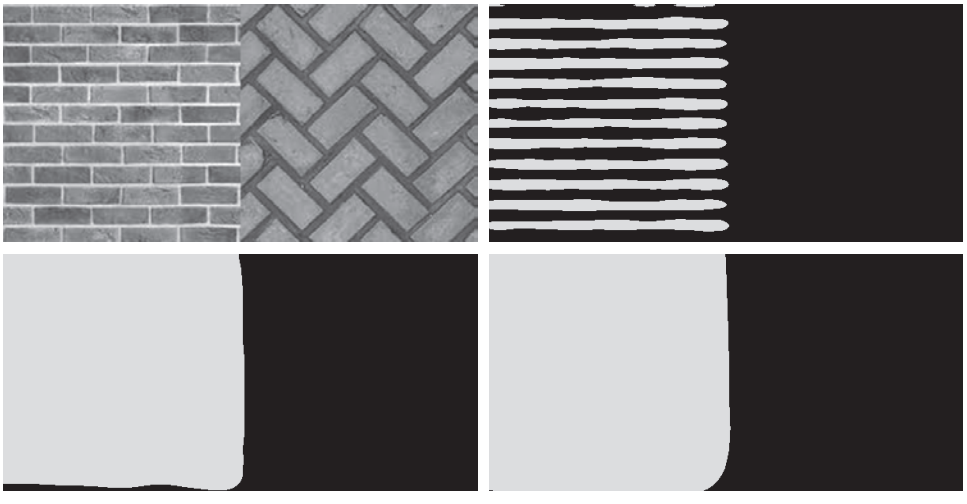
As we discussed in Section 11.1, segmentation assigns each pixel in an image to a region based on the properties being used in the segmentation algorithm. Thus, the final step in using Gabor filtering for image segmentation is to assign pixels in the input image to regions based on the filtering results. The mechanics are the same as those we used for k-means clustering. For example, using a filter bank of six Gabor kernels will result in one filtered image per filter, for a total of six images. For each spatial location (x, y) in the image we form a vector $\mathbf{z}$ based on Eq. (11-29), where each of the six elements of $\mathbf{z}$ corresponds to the response of one of the six filters at (x, y). The following example demonstrates the procedure using k-means clustering.

k-means clustering is just one of the many classification methods we could use. You will learn many more in Chapter 14.

EXAMPLE 11.19: Image segmentation using Gabor texture and k-means clustering.

In this example, we segment Fig. 11.33(a) using various combinations of parameters `wavelength` and `orientation`. As a starting point, we used the same wavelengths as in Example 11.18:

```
>> f = imread('texture-bricks.tif');
```



a b
c d

FIGURE 11.33 Image texture segmentation using functions `gabor`, `imgaborfilt`, and `kmeansClustering`.

and then repeating the segmentation with all other parameters unchanged:

```
>> % Smooth.
>> for j = 1:size(gaborMag,3)
     sigma = 0.8*h(j).Wavelength;
     Q = 3;
     gaborMag(:,:,j) = imgaussfilt(gaborMag(:,:,j),Q*sigma);
end
```

Figure 11.33(c) shows that smoothing did improve the segmentation, but the bottom left part of the image is not segmented properly. We can bring more "detail" to this area by increasing the wavelength, which shortens the frequency. Figure, 11.33(d) is the result of using

```
>> wavelength = [8 16];
```

in the preceding code, including smoothing. This time the segmentation was more successful, except for the area in the bottom middle of the figure, where part of brick region on the left was interpreted erroneously as being part of the brick region on the right.

When the resolution of wavelength values needs to be higher, Toolbox documentation suggests using the following code, which is based on the work of Jain and Farrokhnia [1991] (see the function documentation for a full reference citation):

```
>> wavelengthMin = 4/sqrt(2);
>> wavelengthMax = hypot(numRows,numCols);
>> n = floor(log2(wavelengthMax/wavelengthMin));
>> wavelength = 2.^(0:(n-2)) * wavelengthMin;
```

Similarly, finer resolution in orientation can be specified using

```
>> deltaTheta = a_numeric_scalar;
>> orientation = 0:deltaTheta:(180-deltaTheta);
```

IMAGE SEGMENTATION USING GRAPH CUTS

In image processing, a graph cut is a method based on graph theory that is used for segmenting a digital image into foreground and background. Basically, the approach is to construct a graph where each image pixel is a node connected to other nodes by graph edges, each of which has an associated weight such that, the higher the probability that pixels connected by an edge are related, the higher the weight for that edge should be. Graph cut algorithms perform segmentation by cutting along weak edges of a graph.

Images as Graphs

A *graph*, G, consists of a set V of *nodes* (also called *vertices*) and a set E of *edges* (also called *links*) connecting the nodes:

$$G = (V, E) \tag{11-38}$$

After the polygon is completed, we get the positions of its vertices using the command

```
>> roiPoints = handle.Position;
```

where `roiPoints` is a matrix of size $nv \times 2$, each row of which contains the `(col,row)` coordinates of one of the nv vertices.

To generate the binary `ROI` mask we use function `poly2mask`, whose syntax is

`poly2mask`

```
ROI = poly2mask(cv,rv,m,n)
```

where `cv` and `rv` are column vectors containing the `(col,row)` coordinates of the polygon vertices and `m` and `n` are the row and column sizes of the `ROI`. Using the preceding notation, we generate an `ROI` using the command

```
>> ROI = poly2mask(roiPoints(:,1),roiPoints(:,2),size(L,1),size(L,2));
```

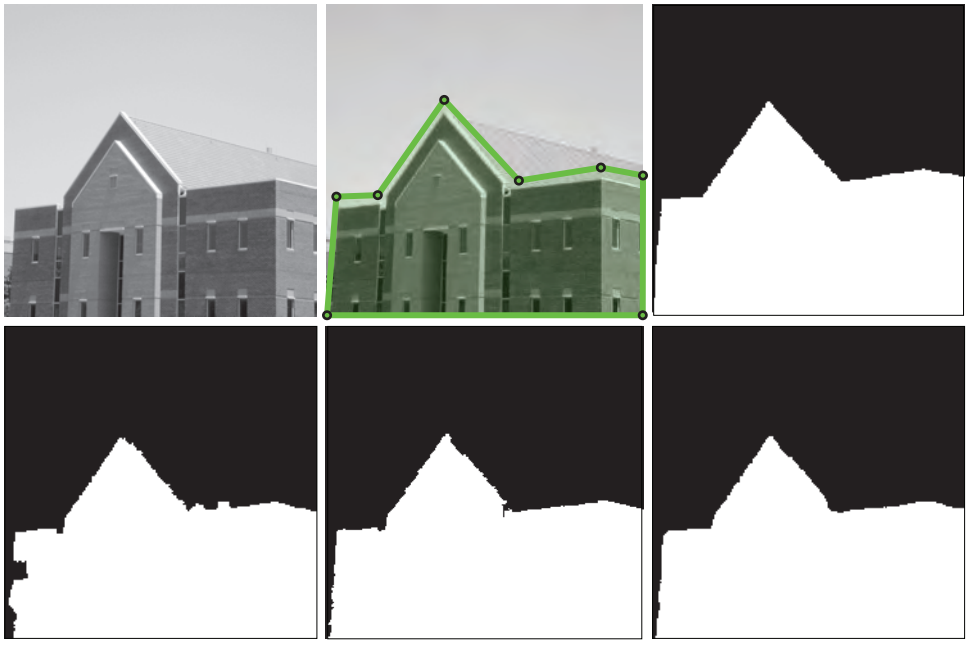
Then, we segment the image using the command

```
>> fseg = grabcut(f,L,ROI);
```

EXAMPLE 11.20: Image segmentation using function `grabcut`.

We want to use function `grabcut` to segment the following image:

```
>> f = imread('building.tif');
```



a b c
d e f

FIGURE 11.35
(a) Input image. (b) Polygonal boundary of ROI, specified interactively. (c) ROI. (d) Segmentation using 500 labeled regions. (e) Segmentation using 5000 labeled regions. (f) The image in (e) smoothed using a Gaussian kernel.

11.5 SEGMENTATION USING THE WATERSHED TRANSFORM

Thus far, we have discussed segmentation based on three principal concepts: edge detection, thresholding, and region extraction. Each of these approaches was found to have advantages (e.g., speed in the case of global thresholding) and disadvantages (e.g., the need for edge linking in edge-based segmentation). In this section, we discuss segmentation based on the concept of *morphological watersheds*. Segmentation by watersheds embodies many of the concepts of the other three approaches and has the advantage that it often produces more stable segmentation results, including connected segmentation boundaries. This approach also provides a simple framework for incorporating *knowledge-based constraints* in the segmentation process, as we will discuss at the end of this section.

In geography, a *watershed* is the ridge that divides areas drained by different river systems and a *catchment basin* is the geographical area draining into a river. The *watershed transform* applies these ideas to image processing to solve a variety of image segmentation problems.

To understand the watershed transform think of the intensity values of an image as a topological surface, where intensity values are interpreted as heights. For example, we can visualize the simple image in Fig. 11.37(a) as the three-dimensional surface in Fig. 11.37(b). If we imagine rain falling on this surface, it is clear that water would collect in the two areas labeled as catchment basins. Rain falling exactly on the watershed ridge line would be equally likely to collect in either of the two catchment basins. The watershed transform finds the ridge lines and basins in an intensity "relief map" of an image. In terms of image segmentation, the key concept is to change the starting image into another image whose catchment basins are the objects or regions we want to identify.

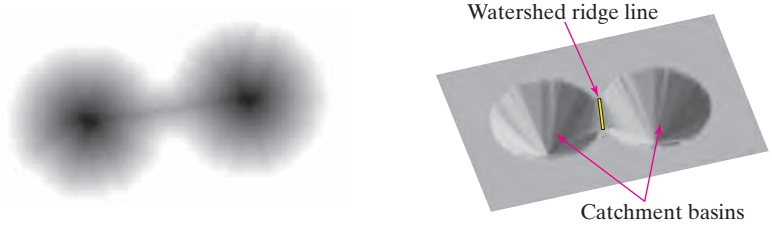
Methods for computing the watershed transform are discussed in Gonzalez and Woods [2018] and in Soille [2003]. The algorithm used in the Toolbox is adapted from Meyer [1994].

THE DISTANCE AND WATERSHED TRANSFORMS

A tool used frequently in conjunction with the watershed transform is the *distance transform*, defined for a binary image as the distance from every pixel to its nearest nonzero-valued pixel. For example, Fig. 11.38(a) shows a small binary image and

a b

FIGURE 11.37
(a) Image. (b) Image viewed as a surface, showing two catchment basins and the watershed ridge line between them.



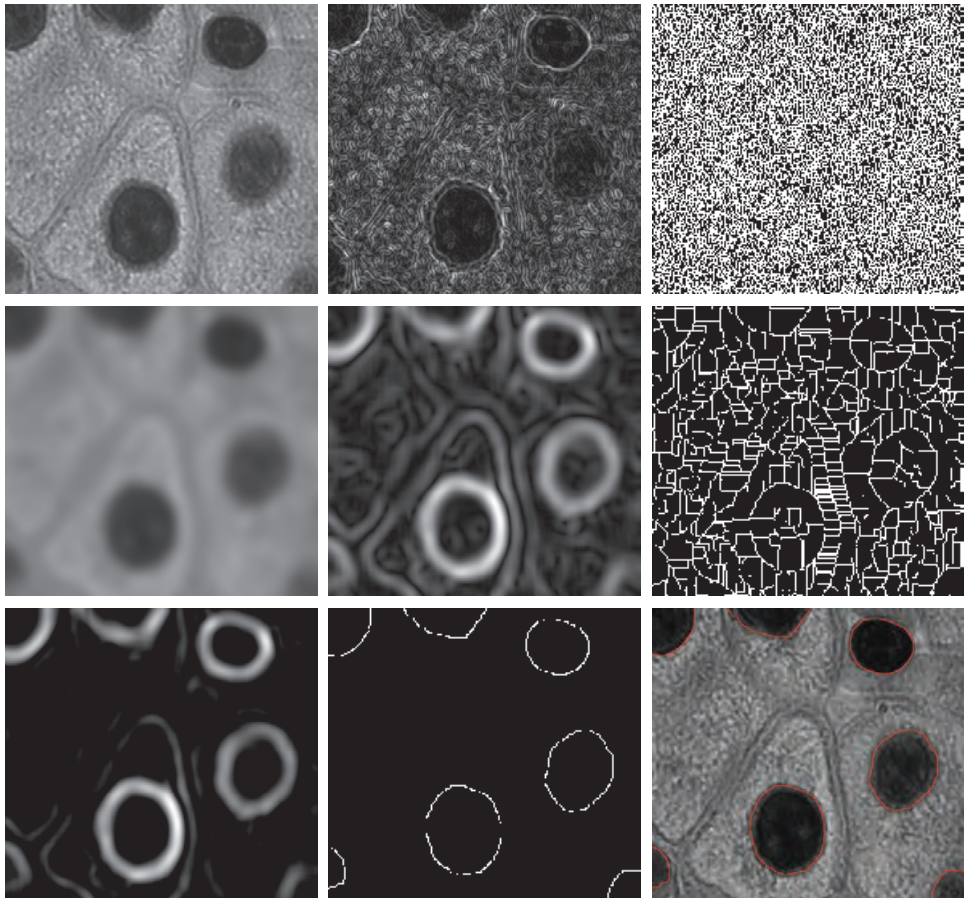
EXAMPLE 11.22: Watershed segmentation of a grayscale image.

Figure 11.42(a) shows a transmission electron microscope image of liver cells. The objective of this example is to segment the cell nuclei (the dark regions) using watershed segmentation. We begin by computing the gradient:

```
>> f = im2double(imread('liver-cells-gray.tif'));
>> figure, imshow(f) % Fig. 11.42(a).
>> g = imgradient(f);
>> figure, imshow(g,[]) % Fig. 11.42(b).
```

Figure 11.42(b) is in a form suitable for function `watershed` because the regions we want to segment are darker than the general background:

```
>> L = watershed(g);
>> ridges = L == 0;
>> figure, imshow(ridges) % Fig. 11.42(c).
```



a b c
d e f
g h i

FIGURE 11.42
(a) Input image. (b) Magnitude of the gradient. (c) Watershed segmentation of (b) showing oversegmentation. (d) Smoothed image. (e) Magnitude of the gradient. (f) Watershed segmentation of (e)—oversegmentation is still evident. (g) Image (e) after processing with function `imhmin`. (h) Watershed segmentation boundaries. (i) Boundaries coded red and superimposed on the original image. (Image (a) courtesy of NIH.)

problem. Humans often aid segmentation and higher-level tasks in everyday vision by using a priori knowledge, one of the most familiar being the use of context. Thus, the fact that segmentation by watersheds offers a framework that can make effective use of this type of knowledge is a significant advantage of this method.

Summary

Image segmentation is an essential preliminary step in most automatic pictorial pattern recognition and scene analysis problems. As indicated by the range of methods and examples presented in this chapter, the choice of one segmentation technique over another is dictated mostly by the characteristics of the problem being considered. The methods discussed in this chapter, although far from being exhaustive, are representative of techniques used commonly in practice.

MATLAB Projects

*Solutions to the projects marked with an asterisk * are in the DIPUM3E Student Support Package (consult the book web site). All your code must be documented so that typing* `help` *at the prompt, followed by the script or function name, gives enough detail for a user to be able to run it. Test the functionality of all your code thoroughly.*

11.1 Point detection.

(a)* The image `sphere-with-embedded-white-point.tif` contains an isolated single bright point that is almost invisible. Give a set of commands that will find this single point. (*Hint:* Use a Laplacian kernel followed by thresholding.)

(b) Confirm that a single point was found.

11.2 Edge detection.

(a)* Generate a black image of size 512×512 pixels with a white square of size 256×256 at its center. Compute and display the gradient magnitude image using the Sobel kernels. Are all the values along the resulting perimeter the same? Explain.

(b)* Compute the gradient angle image of the image from (a) using the Sobel kernels. Determine if all angle values are positive and/or negative.

(c) You noticed in (b) that the bottom part of the angle image around the object is not showing, while the other three parts are. Explain the reason.

(d) Read the image `wingding-square-empty.tif` and generate a new image of the same size with only the horizontal edges detected.

11.3 Edge detection using compass kernels.

(a) The Sobel and Prewitt kernels provided by the Toolbox are well suited for detecting vertical and/or horizontal edges. It is useful sometimes to be able to find edges in other directions. The *Kirsch compass kernels* shown in the figure below are well-suited for this purpose. To determine the strongest edge response of a Kirsch kernel, consider a binary image. A kernel has its strongest response when all the pixels on the left of the 5s are 0s and all the image pixels under the 5s are 1s. For example the Kirsch kernel with the strongest response at the highlighted pixel in Fig. 11.1 would be the SE kernel, whose edge direction is 45°.

12 Image Segmentation II

Active Contours: Snakes and Level Sets

Four snakes gliding up and down a hollow for no purpose that
I could see—not to eat, not for love, but only gliding.
Ralph Waldo Emerson

In this chapter, we develop the foundation for image segmentation using active contours, which are deformable models confined to the image plane. We discuss two types of active contours—*snakes* and *level sets*. Snakes are active contours based on explicit (e.g., parametric) representations of segmentation curves; they derive their name from the way the curves appear to "slither" on the plane in the process of seeking region boundaries. Level sets are based on implicit representation of curves, which are techniques for representing active contours as the intersection of a 3-D surface with a plane. We will discuss in the following sections the fundamental equations of both approaches starting from basic principles, write code to implement all function needed for each method, and give numerous examples that illustrate the strengths and limitations of each approach.

Functions Developed in this Chapter:

- `snakeIterate` implements a snake iterative solution.
- `snakeMap` computes an edge map for use in the snake iterative solution.
- `snakeForce` implements a variety of snake forces.
- `snakeRespace` respaces the coordinates of an evolving snake.
- `levelsetIterate` implements a level set iterative solution.
- `levelsetCurvature` computes the curvature during level set interface evolution.
- `levelsetFunction` generates a signed distance function for use as a level set function.
- `levelsetReset` resets the level set function during iteration so that it remains a signed distance function.
- `levelsetForce` implements a variety of level set forces.
- `levelsetHeaviside` implements the Heaviside equation and its derivative.

It is useful to think of a snake as a thin, closed, flexible body lying on a planar surface and surrounding a solid planar region. If we think of the planar surface as the plane of an image, the planar region as an image object, and the snake as a flexible contour, our interest is on the dynamics of what it would take to push that contour so that it adheres to the perimeter of the region and thus becomes a segmentation boundary.

Accomplishing this objective requires knowledge about the mechanical properties of the contour and the type of external energy field that would push it toward the perimeter of a region. The two most important mechanical properties in this case are the *elasticity* and *stiffness* of the snake. The first property describes the ability of the snake, $\mathbf{c}(s)$, to stretch and shrink along it length while the second is a property that makes the snake stiff enough so that it can be manipulated by the force field.

We know from basic mechanics that elastic energy is proportional to the first derivative squared of $\mathbf{c}$ with respect to s. Stiffness is proportional to bending energy, which we know from mechanics to be proportional to the second derivative squared of $\mathbf{c}$ with respect to s. Conceptually, we can imagine an energy field acting on a body whose properties can be described in the terms of the mechanical energy needed to deform it. The objective is to have the internal and external energies act on the snake so that the total energy is minimized with respect to $\mathbf{c}$, the idea being that the curve yielding the minimum energy will correspond to a suitable segmentation contour (i.e., a snake adhering to the boundary of a region). Mathematically, this becomes an optimization problem that is solved using variational calculus. The details of the solution are addressed in Gonzalez and Woods [2018]. Here, our focus is on implementing the results in MATLAB.

The solution to our energy minimization problem is

$$\alpha\mathbf{c}''(s) - \beta\mathbf{c}''''(s) + \mathbf{F}\big(\mathbf{c}(s)\big) = 0 \tag{12-5}$$

This equation indicates that finding a snake contour can be interpreted as a process of *balancing* internal (elastic and bending) forces against an external force. The double and quadruple quotes indicate the second and fourth derivatives of $\mathbf{c}$ with respect to s. We started with internal snake energies as first and second derivatives. The higher-order derivatives in Eq. (12-5) resulted from the solution of the problem. Also, note that the last term is an external force instead of external energy. This is because the negative derivative of energy is a force.

Equation (12-5) is the fundamental *snake equation* that we must solve to find the optimum segmentation contour. Unfortunately, this equation cannot be solved analytically because $\mathbf{c}$ (which is what we are looking for) must be known before we can compute the force. Thus, we must resort to numerical methods to find a solution, as we will discuss in the following section.

ITERATIVE SOLUTION OF THE SNAKE EQUATION

We begin by making the snake *dynamic*, in the sense of adding an *artificial time variable* t and restating the snake equation as

Implementation of the Snake Iterative Solution

Implementing the iterative matrix formulation of the snake in Eq. (12-7) is straightforward. We use MATLAB function `interp2` for the interpolation required to obtain the force vectors $\mathbf{f}_x$ and $\mathbf{f}_y$ from the 2-D arrays $F_x(x, y)$ and $F_y(x, y)$. As noted earlier, interpolation is required because the coordinates of $F_x(x, y)$ and $F_y(x, y)$ are integers, whereas the coordinates of the snake during iteration generally are not. The `interp2` syntax of interest in this section is

`interp2`

```
fx = interp2(Fx,y,x,'linear',0)
```

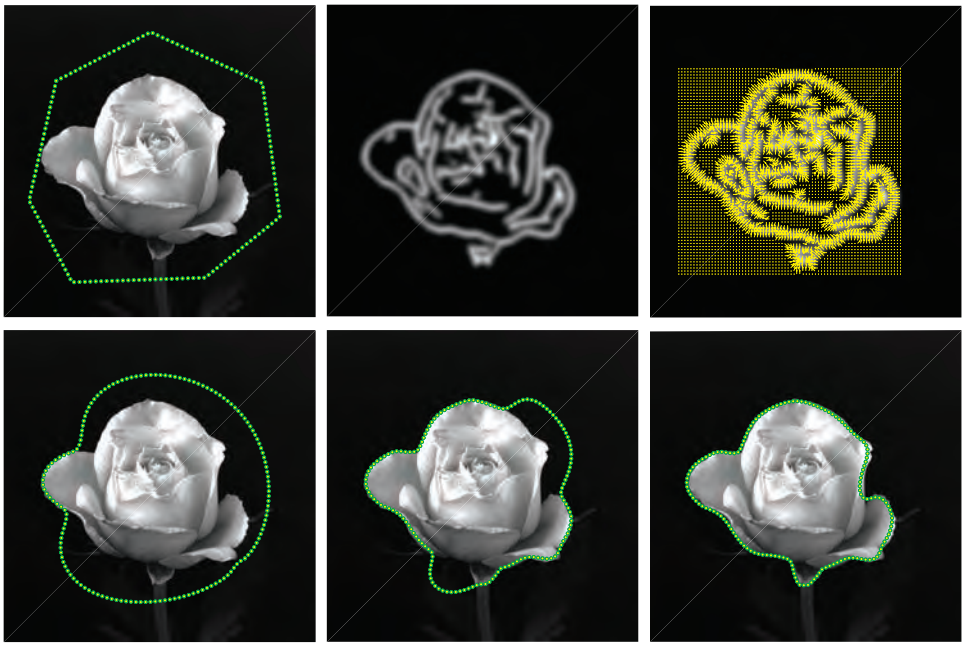
where `x` and `y` are the coordinates of the snake (note the order in which they are input). To obtain vector `fy`, we use `Fy` intead of `Fx` in the function call. Although `interp2` is capable of more advanced interpolation modes, *linear interpolation* is faster and generally is sufficient for snake work. The `0` is used to suppress `NaN`s in areas where there are not enough points for interpolation.


```
function [xs,ys] = snakeIterate(alpha,beta,gamma,x,y,NI,Fx,Fy)
%snakeIterate Iterative solution of the snake equation.
%   [XS,YS] = SNAKEITERATE(ALPHA,BETA,GAMMA,X,Y,NI,Fx,Fy) computes the
%   [XS,YS] coordinates of a segmentation snake using the iterative
%   solution in Eq.(12-7) of DIPUM3E. Vectors X and Y are the initial
%   coordinates of the snake (provided in sequential order). These
%   vectors are updated during iteration. ALPHA, BETA, and GAMMA are
%   parameters in Eq. (12-7) and (12-8), and Fx, Fy are the 2D force
%   arrays obtained, for example, using DIPUM3E function snakeForce.
%
%   This function is normally run within an outer loop with snake-point
%   respacing after each execution of the loop. NI controls the number
%   of iterations of Eq. (12-7) before the snake points are respaced. A
%   common value of NI is 1, indicating one execution of point respacing
%   after each iteration of Eq. (12-7).


% PRELIMINARIES.
K = numel(x);
% Multiply the forces by gamma.
Fx = gamma*Fx;
Fy = gamma*Fy;

% CONSTRUCT MATRIX A IN EQ. (12-8) FOR USE IN EQ. (12-7).
% First construct matrix D2 in Eq. (12-9).
a = -2*ones(K,1);
b = 1*ones(K-1,1);
D2 = diag(a) + diag(b,-1) + diag(b,1);
D2(1,K) = 1;
D2(K,1) = 1;
% Next construct D4 in Eq. (12-10).
a = 6*ones(K,1);
b = -4*ones(K-1,1);
c = 1*ones(K-2,1);
D4 = diag(a) + diag(b,-1) + diag(b,1) + diag(c,-2) + diag(c,2);
```

EXAMPLE 12.5: Snake segmentation of the rose image.

In this example, we consider an image with a more complex boundary. Figure 12.8(a) shows the familiar rose image and a 150-point initial snake. Figure 12.8(b) is the edge map obtained using a Gaussian lowpass kernel with the parameters indicated in the figure caption. As before, we used smoothing on the image and also on the edge map. We thresholded the smoothed map at 0.005. Figure 12.8(c) shows a MOG field superimposed on the edge map. Figure 12.8(d) is the result after 300 iterations of function `snakeIterate` using the parameters listed in the figure caption. Similarly, Figs. 12.8(e) and (f) show the results after 600 and 900 iterations. The latter image shows a nearly-perfect segmentation snake. The code for this example is similar to what we have used in previous examples, so we leave it as an exercise for you to duplicate the results in Fig. 12.8 (see Project 12.5).

Figure 12.8 illustrates several important factors of segmentation using snakes. The most important factor is to start with a clean, continuous edge map of the boundary we want to find—we will show later in Example 12.14 what can happen when the edge of the boundary has breaks in it. Another factor is that at least part of the initial snake should lie in the region of influence of the force field. In fact, you can see by comparing Figs. 12.8(a) and (d) that the first part of the snake that converged to the boundary was the segment of the initial snake closest to the boundary. As part of a snake adheres to a boundary, it brings other of its parts closer to the force field. It makes sense that a snake starting far away from the influence of the force field will take longer to converge, or simply not converge at all. As we showed earlier and as you will see again in Project 12.5, the main reason for using a GVF field is that its region of influence is typically larger than for a MOG field. Yet another important factor to keep in mind when specifying the parameters for use in function `snakeIterate` is our discussion of Eq. (12-5), where we indicated that snake behavior during iteration is affected by the *balance* between internal forces (controlled by parameters α and β) and external forces (controlled by γ). Because the edge map and force



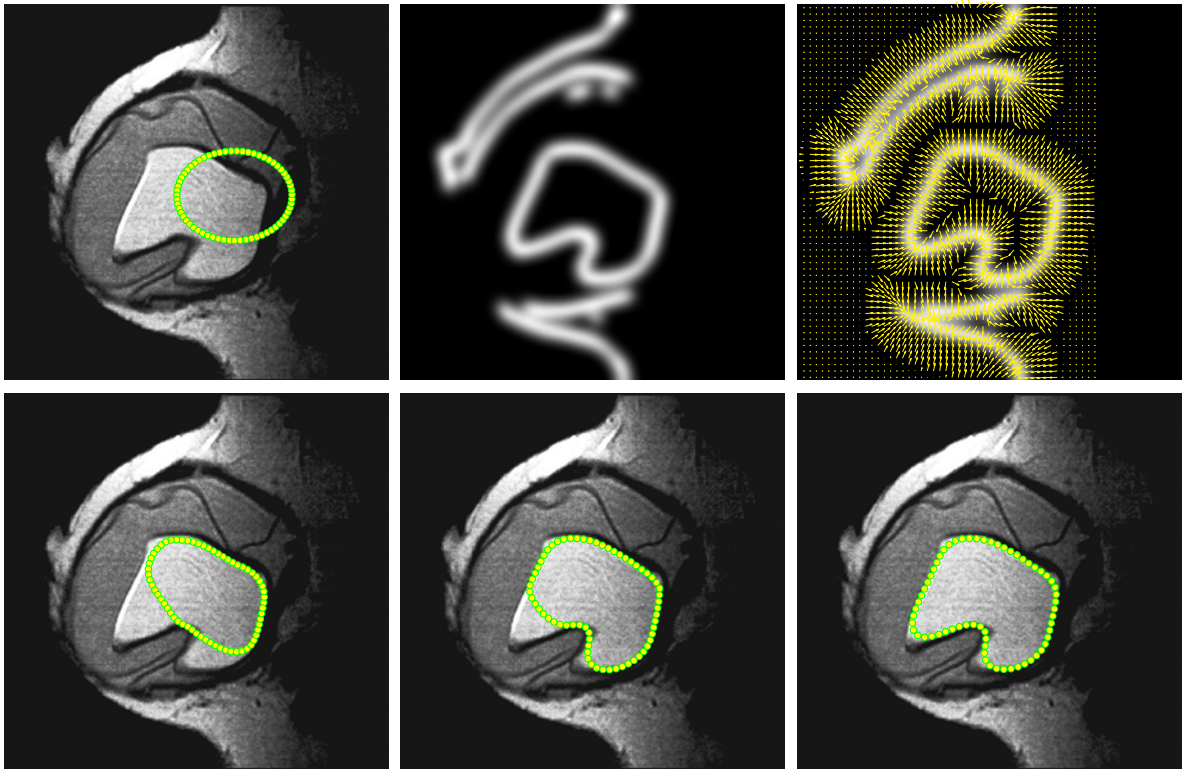
a b c
d e f

FIGURE 12.8 (a) A 1024×1024 image and 150-point initial snake. (b) Edge map using the `'both'` option in `snakeMap`, a Gaussian kernel with $\sigma = 11$, and a threshold of 0.005. (c) MOG force field. (d) Snake after 300 iterations using $\alpha = 0.05$, $\beta = 0.5$, and $\gamma = 5$. (e) and (f) Snake after 600 and 900 iterations, respectively.

fields are independent of the starting snake, it often helps to use them as guides to specify the initial snake configuration.

EXAMPLE 12.6: Snake segmentation of a region in a more complex image.

In all examples thus far, we have specified initial contours outside the object of interest, in which the snakes evolved inwardly. However, snakes can also evolve outwardly, or in a combination of both motions, implying that the initial snake can straddle the boundary of the region to be segmented. Figure 12.9 illustrates this behavior. Figure 12.9(a) is an MRI image of a human breast showing a partially collapsed breast implant (the ellipse shown is the initial snake). We are interested in obtaining the boundary of the implant. As motivation for this type of processing, imagine that you were conducting a study of a historical medical database containing thousands of images of breast implants. An important aspect of such a study might be to analyze the shape of the implants in order to quantify abnormalities



a b c
d e f

FIGURE 12.9 (a) 586×600 MRI image of a human breast and a 64-point initial snake. (b) Edge map obtained using an $11\sigma \times 11\sigma$ lowpass kernel with $\sigma = 5$ and a threshold of 0.01. (c) GVF force field superimposed on the edge map. (d) Result of 25 iterations using $\alpha = 0.05$, $\beta = 0.5$, and $\gamma = 2.5$. (e) Result of 50 iterations using the same parameters. (f) Result of 100 iterations. (Original image courtesy of NIH/National Library of Medicine.)

(e.g., collapsed implants) as a percentage of normal implants. Even if total automation is not acceptable—a typical constraint in medical image processing—a semi-automated technique in which a human expert initiates the process by pointing to a starting location in the implants and lets a computer extract and quantify the boundary, often is acceptable. Such an approach can save many hours of effort and yield more accurate measurements than manual estimates.

To generate the results in Fig. 12.9, we used the parameters listed in the figure caption. For the GVF force field we used $\mu = 0.25$ and 100 iterations, which are the same settings we used in Example 12.4. The results in Figs. 12.9(a) and (d) through (f) show how the snake evolved from an initial position straddling the boundary of the implant, to an almost perfect segmentation of that region. You are asked in Project 12.6 to duplicate the results in Fig. 12.9. You will also experiment with snakes starting inside and outside a region of interest.

12.3 IMAGE SEGMENTATION USING LEVEL SETS

As we mentioned in Section 12.1, level sets in the context of image segmentation are sets of points of a 2-D curve formed by the intersection of a plane and a 3-D surface. Unlike the parametric representation used for snakes, level sets are based on implicit representations. An important aspect of this approach is that it can adapt to changing topology, such as the emergence of new regions, during curve evolution. Inherently, parametric curves do not have this capability. However, as we will illustrate later in this section, each approach has strengths that make it an appropriate choice in a given application. As noted in Section 12.1, level sets were used initially to describe the propagation of interfaces between fluids. In the terminology of image segmentation, "fluids" represent image regions and "interfaces" become segmentation contours that separate one region from another.

IMPLICIT REPRESENTATION OF ACTIVE CONTOURS

The representation of snakes discussed in Section 12.2 is *explicit*, in the sense that an active contour is represented by an equation written in Cartesian or, more frequently, parametric form. An alternate representation of a 2-D contour is to define it *implicitly* as the intersection of a plane and a 3-D surface. To illustrate, consider the explicit equation of a circle centered at point (x_0, y_0) in the xy-plane:

$$(x - x_0)^2 + (y - y_0)^2 = r^2$$

Figure 12.10(a) shows a generic plot of this function. We can write this equation equivalently as

$$(x - x_0)^2 + (y - y_0)^2 - r^2 = 0$$

Remember, a scalar function outputs a scalar value, independently of the number of scalar variables on which it depends.

Suppose that we define the following *scalar function* of two variables:

$$\phi(x, y) = (x - x_0)^2 + (y - y_0)^2 - r^2$$

a ϕ that satisfies Eq. (12-22); and (4) extract the segmentation contour as the zero level set of ϕ. As with snakes, the choice of F plays a central role in the effectiveness of segmentation algorithms based on level sets. In the formulations to be discussed shortly, F will depend in general on both image data (e.g., edges) *and* the level set function itself (e.g., the curvature of ϕ). Level set algorithms operate using scalar fields, unlike snakes which work with vector fields.

ITERATIVE SOLUTION OF THE LEVEL SET EQUATION

As noted earlier, Eq. (12-22) cannot be solved directly because the evolution of F and ϕ can seldom be expressed analytically, except in simple cases. So, as we did with snakes, we resort to numerical techniques by discretizing the level set equation. This consists of discretizing the *temporal* (i.e., *time*) *derivative* and also the *spatial derivatives* needed to compute the norm of the gradient vector.

As before, we approximate the time derivative using finite differences:

$$\frac{\partial \phi(x,y,t)}{\partial t} = \frac{\phi(x,y,t+\Delta t) - \phi(x,y,t)}{\Delta t} \tag{12-23}$$

The result of discretizing t is the following iterative equation:

$$\phi^{n+1} = \phi^n - \Delta t\left\{F(\phi^n)\|\nabla\phi^n\|\right\} \tag{12-24}$$

where

$$\phi^n = \phi(x,y,n\Delta t) \tag{12-25}$$

is the value of ϕ after n iterative steps.

We still have to discretize Eq. (12-24) with respect to x and y. This means discretizing the computation of the term $F(\phi^n)\|\nabla\phi^n\|$ in Eq. (12-24). As with snakes, the process for doing this is not difficult, but it is tedious and outside the scope of this discussion (see Gonzalez and Woods [2018] for the derivations). The result is

$$\phi^{n+1} = \phi^n - \Delta t\left\{\max(F,0)\left\|\nabla\phi^n\right\|^+ + \min(F,0)\left\|\nabla\phi^n\right\|^-\right\} \tag{12-26}$$

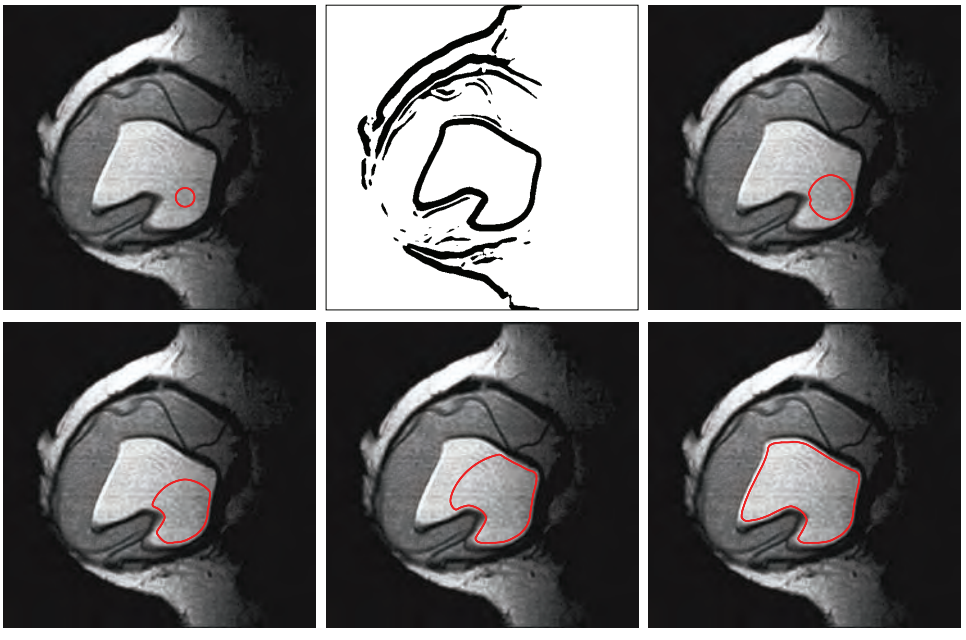
where the terms are explained in the next paragraph. This is the complete *iterative solution* of the level set equation given in Eq. (12-22). Equation (12-26) is said to have converged if $\phi^{n+1} = \phi^n$ within a tolerance bound. If $\phi^{n+1} \neq \phi^n$ within that tolerance bound, we update the right side of Eq. (12-26) using ϕ^{n+1}, increase n by 1, and compute the next iteration using that equation. The final contour is obtained as the zero-level set of the final ϕ. When the increments of x and y are unity (as with digital images), the so-called *Courant-Friedreichs-Lewy (CLF) condition* from

where $G_\sigma(x,y)\star f(x,y)$ denotes smoothing by performing spatial convolution of $f(x,y)$ with a Gaussian lowpass kernel, G_σ, of a specified size and standard deviation, σ. This is force (2) in Table 12.1. As the following example shows, even this simple force can produce quite effective segmentation results.

EXAMPLE 12.10: Level set segmentation of a grayscale image using a force based on the image gradient.

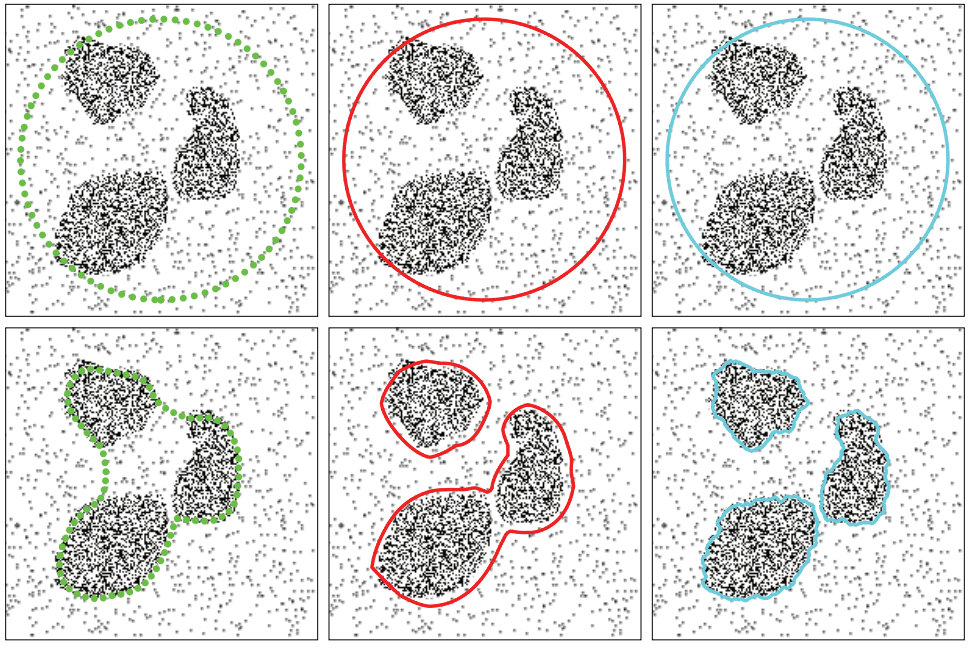
Figure 12.17(a) is the same image we used in Example 12.6. As before, we are interested in obtaining the boundary of the collapsed breast implant. We obtained the image in Fig. 12.17(a) using the following commands:

```
>> f = im2double(imread('breast-implant.tif'));
>> [M,N] = size(f);

>> % Smooth the image.
>> n = 15;
>> sig = 5;
>> G = fspecial('gaussian',n,sig);
>> fsmooth = imfilter(f,G,'replicate');

>> % Initial level set function.
>> x0 = 370;
>> y0 = 350;
>> r = 18;
>> phi0 = levelsetFunction('circular',M,N,x0,y0,r);
```



a b c
d e f

FIGURE 12.17 Level set segmentation of a grayscale image using a gradient force. (a) 586×600 MRI image of a human breast and initial zero level set function.(b) Thresholded gradient force. (c)–(f) Results of 50, 100, 200, and 400 iterations, respectively. (Original image courtesy of NIH/ National Library of Medicine.)



a b c
d e f

FIGURE 12.28
First row: Initial contours. Second row: Results of segmentation using (d) snakes with a GVF force, (e) level sets with a geodesic force, and (f) level sets with a Chan-Vese force.

Summary

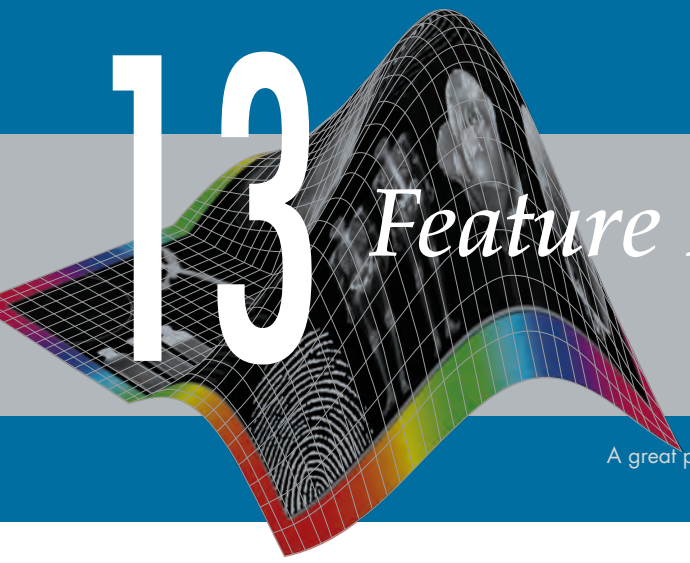
The material in this chapter is a comprehensive foundation for many of the approaches you are likely to encounter using active contours for image segmentation. In particular, a good understanding of the mechanics of the snake and level set equations are essential when it becomes necessary to develop new techniques based on the concepts discussed in the preceding sections. In the process of programming all the various details of these two important segmentation approaches, you learned how to handle iterative equations and the subtleties of what it takes to achieve convergence when using active contours for image segmentation.

MATLAB Projects

*Solutions to the projects marked with an asterisk * are in the DIPUM3E Student Support Package (consult the book web site). All your code must be documented so that typing* `help` *at the prompt, followed by the script or function name, gives enough detail for a user to be able to run it. Test the functionality of all your code thoroughly.*

12.1 Do the following:

(a) Write a function `ellipticalCurve` for generating and optionally superimposing an ellipse on an image. Your function should have the syntax `[x,y] = ellipticalCurve(param)` where `[x,y]` are the (row, col)coordinates of the ellipse and `param` is a structure whose fields specify: (1) the number of points in the ellipse; (2) the coordinates of the center of the ellipse; (3) the orientation of the ellipse as the angle (in degrees) of its major axis with respect to the image x-axis (see Section 2.1 for a discussion of the image coordinate system); (4) the lengths of the ellipse semimajor and semiminor axes; and (5) a display option that optionally displays it superimposed on a white image of specified size. The default is not to display the ellipse.



13 Feature Extraction

A great product isn't just a collection of features. It's how it all works together.
Marco Arment

After an image has been segmented into regions or their boundaries using methods such as those in Chapters 11 and 12, the resulting sets of segmented pixels usually have to be converted into a form suitable for further computer processing. In most applications, the step after segmentation is *feature extraction*, which consists of feature detection and feature description. *Feature detection* refers to finding features in an image, region, or boundary. *Feature description* assigns quantitative or qualitative attributes to the detected features. Some of the methods discussed in this chapter are capable of extracting features directly from an image, thus combining segmentation and feature extraction into one step.

Functions Developed in this Chapter:

- `bound2im` converts a boundary to a binary image.
- `uppermostLeftmost` finds the uppermost, leftmost foreground point in a binary image.
- `bound2eight` converts a boundary to an 8-connected path.
- `bound2four` converts a boundary to a 4-connected path.
- `bsubsamp` subsamples a boundary.
- `connectpoly` connects polygon vertices.
- `intline` constructs a digital line.
- `freemanChainCode` generates the Freeman chain code of a boundary.
- `im2minperpoly` finds a minimum-perimeter polygon enclosing a region.
- `boundarydir` determines the direction of a set of boundary points.
- `signature` computes the signature of a boundary.
- `diameter` generates the diameter, major axes, and basic rectangle of a boundary or region.
- `frdescp` and `ifrdescp` compute forward and inverse Fourier descriptors, respectively.
- `statxture` generates statistical texture.
- `specxture` generates spectral texture.
- `invmoments` computes 2-D moment invariants.
- `imstack2vectors` converts a stack of images to vectors.
- `covmatrix` computes the covariance matrix of a set of vector samples.
- `principalComponents` obtains the principal components of a set of vector samples.

$$\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}$$

in which x_1 is the intensity value of the red image at a point and the other components are the intensity values of the green and blue images at the same point. If color is used as a feature, then a region in an RGB image would be represented as a set of feature vectors (points) in 3-D space. When n descriptors are used, feature vectors become n-dimensional and the space containing them is referred to as an *n-dimensional feature space*.

In this chapter, we group features into three principal categories: *boundary*, *region*, and *whole image* features. This subdivision is not based on the applicability of the methods we are about to discuss; rather, it is based on the fact that some categories of features make more sense than others when considered in the context of what is being described. Many of the features in the following sections are applicable to boundaries and regions, and some apply to whole images as well.

We will be working with numeric and logical images. As explained in Chapter 2, *numeric images* are principally images of the familiar `uint8` and `double` classes. A *binary image* is a bivalued numeric image with values generally equal to 0 and 255 for `uint8` images and 0 and 1 for `double` images. Pixels in logical images are logical constants: 0 (FALSE) and 1 (TRUE). We mention this again because some of the functions in this chapter require logical inputs and sometimes output logical results. Generally, we will use lowercase letters such as `f` and `g` to denote numeric (including binary) images and upper case such `BW` for `logical` images. When input and/or output images can be logical or numeric, we will generally use lowercase letters to denote both.

13.2 REGION AND BOUNDARY PREPROCESSING

Most of the segmentation methods discussed in the last two chapters yield raw data in the form of pixels along a boundary or pixels contained in a region. It is standard practice to preprocess raw segmented data into forms that ultimately help improve feature uniqueness and invariance. In this section, we discuss various preprocessing approaches suitable for this purpose.

DEFINITIONS

For convenience, we repeat here some definitions given in Chapters 2 and 10.

Let S represent a subset of foreground pixels in an image. Two pixels p and q are said to be *connected* in S if there exists a path between them consisting entirely of pixels from S. For any pixel p in S, the set of pixels connected to it in S is called a *connected component*. If it only has *one* connected component, S is called a *connected set*. A subset, R, of pixels in an image is called a *region* of the image if R is a connected set.

In image processing applications, connected components typically only have one component, so use of the term *connected component* generally refers to a region.

The *boundary* (also called the *border* or *contour*) of a region is defined as a set of pixels in the region that have one or more neighbors that are not in the region. Initially we are interested in binary images, so foreground pixels are represented by

a b c

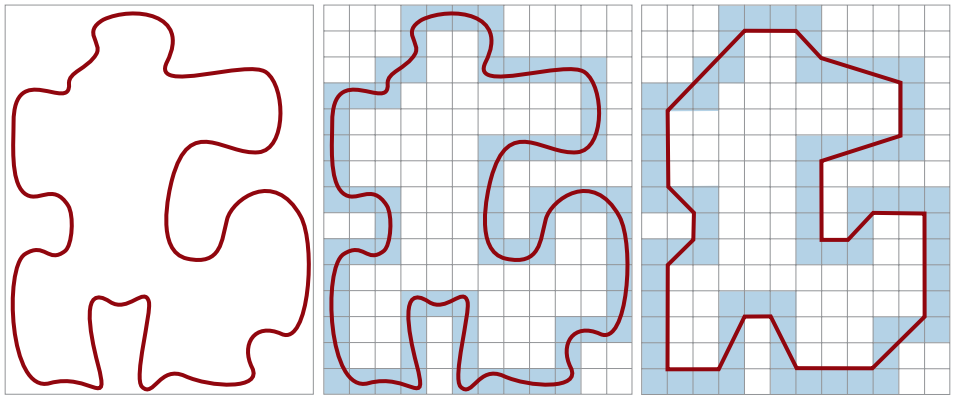


FIGURE 13.5
(a) An object boundary. (b) Boundary enclosed by cells. (c) Minimum-perimeter polygon created when the boundary shrinks. The vertices are created by the corners of the inner and outer walls.

concave vertices (black dots) in the outer wall. A little thought will reveal that only convex vertices of the inner wall and mirrored concave vertices of the outer wall can be vertices of the MPP. Thus, our algorithm needs to focus attention only on those vertices.

An Algorithm for Finding MPPs

The set of cells enclosing a boundary is called a *cellular complex*. We assume that the boundaries under consideration are not self intersecting, a condition that leads to *simply connected* cellular complexes. Based on these assumptions and letting *white* (W) and *black* (B) denote convex and *mirrored* concave vertices, respectively, we state the following observations:

1. The MPP bounded by a simply connected cellular complex is not self intersecting.
2. Every convex vertex of the MPP is a W vertex, but not every W vertex of a boundary is a vertex of the MPP.

a b c

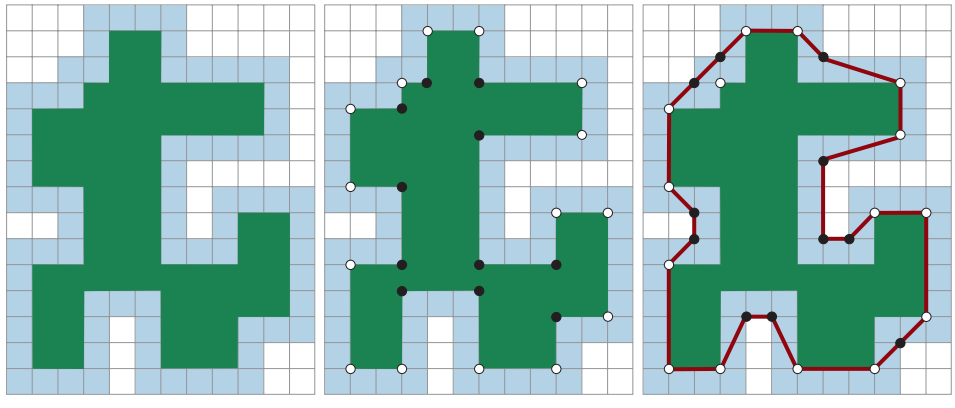
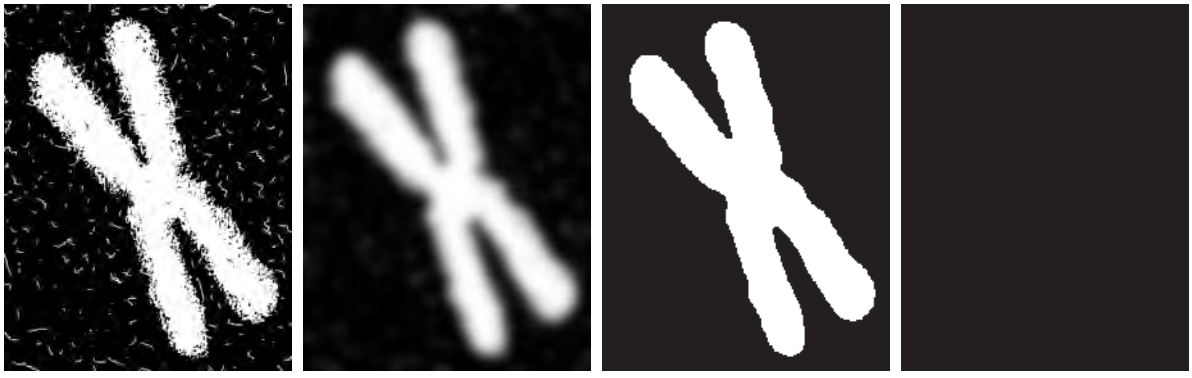


FIGURE 13.6
(a) Region (green) defined by the inner wall. (b) Convex (white) and concave (black) vertices obtained by following the boundary counterclockwise. (c) Concave vertices displaced diagonally to the outer wall. The MPP is shown for reference.



a b c d

FIGURE 13.13 (a) Noisy chromosome. (b) Image smoothed using a 67×67 Gaussian kernel with $\sigma = 11$. (c) Thresholded image. (d) Skeleton.

```
>> BW = imbinarize(fs);
>> figure, imshow(BW) % Fig. 13.13(c).
```

The result in Fig. 13.13(c) is a clean binary image containing a single object. We find its skeleton using function `bwskel`:

```
>> S = bwskel(BW);
>> figure, imshow(S) % Fig. 13.13(d).
```

The result, shown in Fig. 13.13(d), is a skeleton that captures the essence of the object: four elongated arms connected in the center. In earlier versions of the Toolbox, skeletonization was done using function `bwmorph`, which is based on morphological thinning. As you will learn in Project 13.3(f), function `bwmorph` can yield results that are not consistent with the definition of the medial axis transformation.

13.4 BOUNDARY FEATURES

In this section we discuss a number of features that are useful when working with region boundaries. Many of these descriptors are applicable to regions also and the grouping of descriptors in the Toolbox does not make a distinction regarding their applicability. Therefore, some of the concepts introduced here are repeated in Section 13.5 when we discuss regional features.

SOME BASIC BOUNDARY FEATURES

We extract the boundary of objects contained in image `f` using function `bwperim`, introduced in Section 13.2:

The output of function `bwperim` is an image.

```
BW = bwperim(f,conn)
```

```
%   length(Z)-by-2 containing the coordinates of a closed boundary.
%
%  See function FRDESCP for computing the descriptors.

% Preliminaries.
np = length(z);
% Check inputs.
if nargin == 1
   nd = np;
end
if np/2 ~= round(np/2)
   error('length(z) must be an even integer.')
elseif nd/2 ~= round(nd/2)
   error('nd must be an even integer.')
end

% Create an alternating sequence of 1s and -1s for use in centering the
% transform (see Gonzalez and Woods [2018]).
x = 0:(np - 1);
m = ((-1) .^ x)';

% Use only nd descriptors in the inverse.  Because the descriptors are
% centered, (np - nd)/2 terms from each end of the sequence are set to
% 0.
d = (np - nd)/2;
z(1:d) = 0;
z(np - d + 1:np) = 0;

% Compute the inverse and convert back to obtain the boundary
% coordinates.
zz = ifft(z);
s(:,1) = real(zz);
s(:,2) = imag(zz);

% Multiply by alternating 1 and -1s to undo the centering done in
% function frdescp.
s(:,1) = m.*s(:,1);
s(:,2) = m.*s(:,2);
```

EXAMPLE 13.7: Using Fourier descriptors.

Figure 13.16(a) is the same as Fig. 13.13(c) and Fig. 13.16(b) is the boundary of the chromosome extracted using function `bwboundaries`:

```
>> f = imread('chromosome.tif');
>> figure, imshow(f) % Fig. 13.16(a)
>> % Obtain the boundary and display it as an image.
>> B = bwboundaries(f);
>> b = B{1}; % There is only one boundary in this case. Its length is 2688 points.
>> disp(length(b))
     2688
>> bim = bound2im(b,size(f,1),size(f,2));
>> figure, imshow(bim) % Fig. 13.16(b).
```

EXAMPLE 13.8: Working with function `regionprops`.

We will use Fig. 13.21(a) to explore the capabilities of function `regionprops`. The first step in using this function is to convert the input to a logical image. Figure 13.21(b) is the result of binarizing the input image:

```
>> f = rgb2gray(imread('liver-cells.tif'));
>> figure,imshow(f) % Fig. 13.21(a).
>> fb = imbinarize(f);
>> figure, imshow(fb) % Fig. 13.21(b).
```

As you can see, the thresholded image contains a large number of small irrelevant regions. We could clean up this image using morphological techniques or by using `regionprops` to obtain all regions and then deleting the "small" ones. However, in this case it is more effective to smooth the image first, before thresholding it:

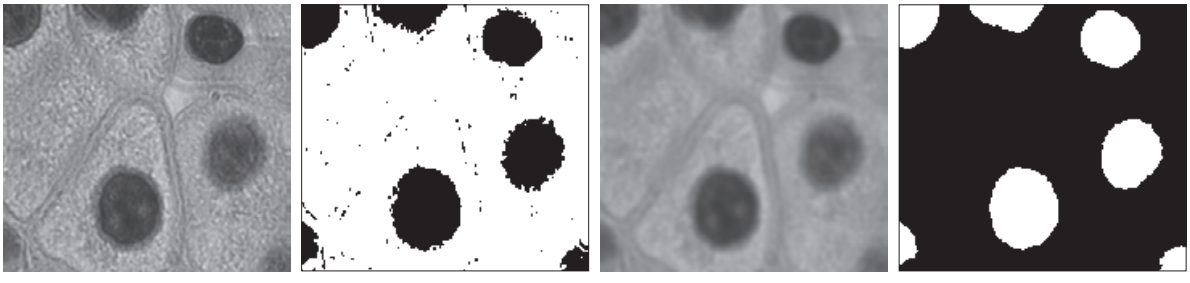
```
>> fs = imgaussfilt(f,3,'FilterSize',19); % Filter size: odd integer ~ 6 x sigma (see Section 3.4 for
                                          % an explanation of why we chose 6 x sigma).
>> figure, imshow(fs) % Fig. 13.21(c).
>> BW = imcomplement(imbinarize(fs));
>> figure, imshow(BW) % Fig. 13.21(d).
```

The result in Fig. 13.21(d) is in the form required by `regionprops`, in the sense that the image is a logical array and the regions are white on a black background.

We begin by computing all the features:

```
>> fdvalues = regionprops(BW,'all');
>> disp(fdvalues)
```

```
7×1 struct array with fields:

Area                MinorAxisLength         ConvexArea      Extrema         PixelList
Centroid            Eccentricity            Image           EquivDiameter   Perimeter
BoundingBox         Orientation             FilledImage     Solidity        PerimeterOld
SubarrayIdx         ConvexHull              FilledArea      Extent
MajorAxisLength     ConvexImage             EulerNumber     PixelIdxList
```



a b c d

FIGURE 13.21 (a) Image of liver cells. (b) Thresholded image.(c) Smoothed image. (d) Thresholded result. (Image (a) courtesy of NIH.)

useful when the same features are used in various sections of code, or when calling `regionprops` from within another function.

TEXTURE

Texture is an important feature for differentiating between regions. While no formal definition of texture exists, we intuitively think of texture features descriptors as quantifiers of properties such as smoothness, coarseness, and regularity. We already introduced the concept of texture features in Section 11.4, where we discussed using Gabor filters for segmentation based on periodic patterns. In this section, we discuss statistical and spectral approaches for describing the texture of a region. Statistical approaches yield spatial descriptions of textures such as smooth, coarse, grainy, and so on. Spectral techniques are based on properties of the Fourier spectrum and are used primarily to detect global periodicity in an image by identifying narrow peaks of high energy in its spectrum.

Statistical Approaches

In this section we discuss two approaches for generating texture descriptors: statistical moments and co-occurrence matrices.

Statistical Moments

One of the simplest approaches for describing texture is to use statistical moments of the intensity histogram of an image or region. Let z be a random variable denoting intensity and let $p(z_i)$, $i = 0,1,2,\ldots,L-1$, be the corresponding normalized histo gram components, where L is the number of distinct intensity levels. From Eq. (5-10), the nth moment of z about the mean is

$$\mu_n(z) = \sum_{i=0}^{L-1} (z_i - m)^n p(z_i) \tag{13-12}$$

where

$$m = \sum_{i=0}^{L-1} z_i p(z_i) \tag{13-13}$$

is the mean (average) value of z. These moments can be computed using function `statmoments` discussed in Section 5.2. Table 13.4 lists some common descriptors based on statistical moments and also on uniformity and entropy. Remember that the second moment, μ_2, is the variance, σ^2.

Custom function `statxture` (see your Support Package for the code) computes the texture descriptors in Table 13.4. Its syntax is


```
t = statxture(f,scale)
```

a
b
c

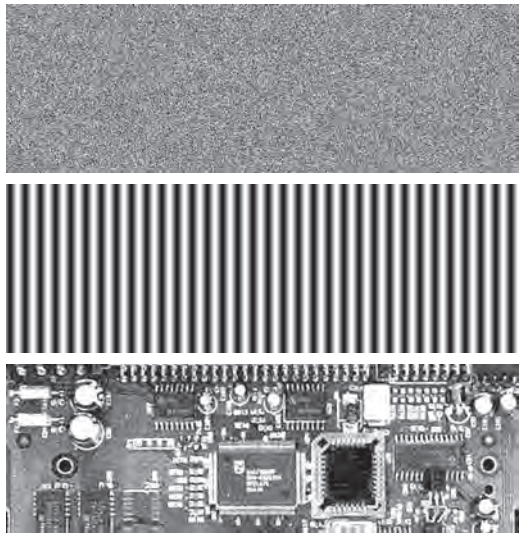


FIGURE 13.27
Images whose pixels exhibit (a) random, (b) periodic, and (c) mixed texture patterns. All images are of size 263 × 800 pixels.

```
>> energy2 = fdvalues2.Energy;
>> hom2 = fdvalues2.Homogeneity;
>> for k = 1:size(G2n,1)
    sumcols(k) = sum(-G2n(k,1:end).*log2(G2n(k,1:end) + eps));
end
>> entropy2 = sum(sumcols);
```

The values of these features are listed in the $\mathbf{G}_2/n_2$ row of Table 13.7. The other two rows were generated using the same procedure with the other two images. The entries in this table agree with what one would expect from looking at the images in Fig. 13.27. For example, consider the Maximum Probability column in Table 13.7. The highest probability corresponds to the third co-occurrence matrix, which tells us that this matrix has the highest number of counts (largest number of pixel pairs occurring in the image relative to the positions in O) than the other two matrices. Examining Fig. 13.27(c) we see that there are large areas characterized by low variability in intensities in the horizontal direction, so we would expect the counts in $\mathbf{G}_3$ to be high.

The third column indicates that the highest correlation corresponds to $\mathbf{G}_2$. This tells us that the intensities in the second image are highly correlated. The repetitiveness of the periodic pattern in Fig. 13.27(b)

TABLE 13.7
Texture feature descriptor values for the images in Fig. 13.27 based on individual co-occurrence matrices.

Normalized Co-occurrence Matrix	Feature Name					
	Max Probability	Correlation	Contrast	Energy	Homogeneity	Entropy
$\mathbf{G}_1/n_1$	0.00006	−0.0005	10838	0.00002	0.0366	15.75
$\mathbf{G}_2/n_2$	0.01500	0.9650	570	0.01230	0.0824	6.43
$\mathbf{G}_3/n_3$	0.05894	0.9043	1044	0.00360	0.2005	13.63

13.6 WHOLE-IMAGE FEATURES

As we mentioned in Section 13.1, it is useful to categorize features as being principally applicable to boundaries, regions, or whole images. These are not mutually-exclusive categories. Rather, they are rough guidelines that help us organize our discussion into categories where these features are generally most applicable.

MOMENT INVARIANTS

We defined the statistical moments of a single random variable in Chapters 3 and 5, and have since used them in various parts of the book. Two-dimensional moments can similarly be defined and used as features. In addition, as we will show in this section, they can be normalized for invariance to translation, scale change, mirroring (within a minus sign), and rotation. As you learned at the beginning of this chapter, these are desirable characteristics of features in general.

The *2-D moment* of order $(p+q)$ of an $M \times N$ digital image, $f(x,y)$, is defined as

$$m_{pq} = \sum_{x=0}^{M-1} \sum_{y=0}^{N-1} x^p y^q f(x,y) \tag{13-26}$$

where p and q are nonnegative integers. The corresponding *central moment* of order $(p+q)$, denoted μ_{pq}, is defined as

$$\mu_{pq} = \sum_{x=0}^{M-1} \sum_{y=0}^{N-1} (x-\bar{x})^p (y-\bar{y})^q f(x,y) \tag{13-27}$$

where

$$\bar{x} = \frac{m_{10}}{m_{00}} \quad \text{and} \quad \bar{y} = \frac{m_{01}}{m_{00}} \tag{13-28}$$

The *normalized central moment* of order $(p+q)$, denoted η_{pq}, is defined as

$$\eta_{pq} = \frac{\mu_{pq}}{\mu_{00}^{\gamma}} \tag{13-29}$$

where

$$\gamma = \frac{p+q}{2} + 1 \tag{13-30}$$

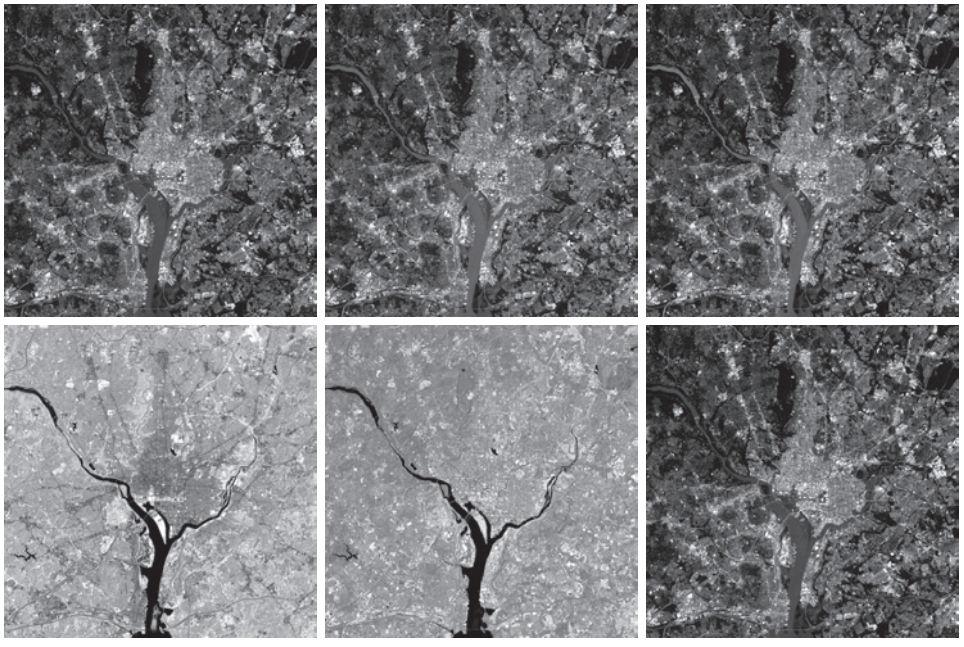
for $p+q = 2,3,\ldots$.

Derivation of the seven moment invariants requires concepts that are beyond the scope of this discussion. See Gonzalez and Woods [2018] for sources containing derivations, and also generalizations to orders higher than seven, and dimensions greater than two.

The set of seven, 2-D *moment invariants* in Table 13.8 can be derived from the second and third normalized central moments. We can attach physical meaning to

a b c
d e f

FIGURE 13.35
Multispectral images reconstructed using only the two principal-component images with the largest eigenvalues. Compare these images with the originals in Fig. 13.33.



```
>> % Display the image that gave the maximum difference.
>> idx = find(maxdiff == max(maxdiff(:)));
>> figure, imshow(diff{idx(1)}) % Fig. 13.36(a)-A black image.
```

We can view the tiny values in the difference image by expanding its values to the [0,1] range. Figure 13.36(b) shows the result:

```
>> % Scale the difference image to see the intensity differences.
>> diffscaled = intensityScaling(diff{idx});
>> figure, imshow(diffscaled) % Fig. 13.36(b).
```

This figure shows that the differences are distributed over the entire image field.

a b

FIGURE 13.36
(a) Difference between the images in Figs. 13.33(a) and 13.35(a). The maximum value of this image is 7.7716e–16. (b) Difference image scaled to the range [0,1].

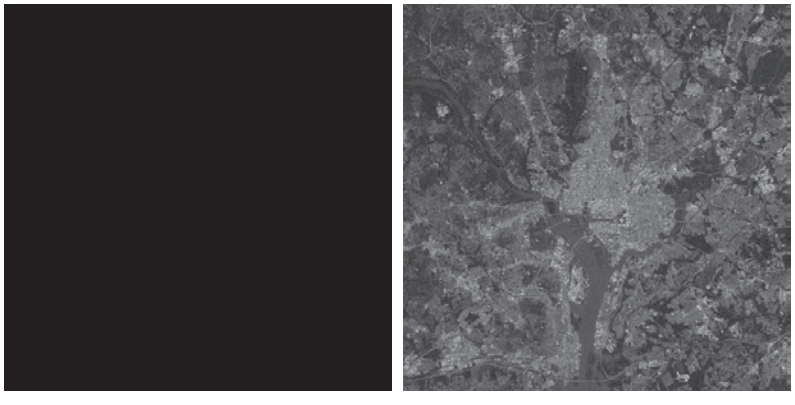


TABLE 13.11
Functions associated with object `corner.Points` for extracting information about the corners stored in the object.

Function	Explanation
`plot`	Plot corner points.
`isempty`	Determine if no corners were found.
`length`	Number of stored corner points.
`selectStrongest`	Select a specified number of the strongest corner points stored.
`size`	The number of corner points found.
`selectUniform`	Select uniformly distributed subset of corner points.
`gather`	Retrieve corner points information from the GPU (not used in this book).

TABLE 13.12
Name/Value pairs for function `detectHarrisFeatures`. Multiple pairs can be specified simultaneously.

Name	Value
`'MinQuality'`	A scalar in the range [0,1] representing the minimum acceptable quality of the corners. The default is 0.01.
`'FilterSize'`	The square dimension of a Gaussian filter kernel used to smooth the input image. Acceptable values are odd integers in the range `[3,min(size(f))]`. The default is 5. The standard deviation is chosen automatically as `FilterSize/3`.
`'ROI'`	Rectangular region of interest over which corners are computed. Specified as a vector `[col,row,width,height]` where `(col,row)` are the coordinates of the upper-left corner of the rectangle.

so many times before in the book. Values in the corner map that are local extrema are potential corners. As noted earlier, the metric for the Harris function is R from Eq. (13-52) and for the eigenvalue detector it is Eq. (13-53). The `MinQuality` parameter is a threshold normalized to the range [0,1] that "passes" corner values in the map whose metric values exceed this threshold, while rejecting those that do not. The `selectStrongest` function selects a specified number of higher-value corners from the set that passed the `MinQuality` threshold. Finally, the `selectUniform` function returns a more uniform spatial spread of corners throughout the image; this prevents "bunching" of the features in one area of the image that could bias further computations, especially when it comes to applications such as image registration. Finally, because `corners` is a `cornerPoints` object, its properties are the three properties we listed earlier when discussing `cornerPoints`.

EXAMPLE 13.15: Corner detection.

Figure 13.39(a) shows an image with numerous corners and Fig. 13.39(b) shows the output of function `detectHarrisFeatures` obtained using the following commands:

```
>> f = imread('national-archives-bld.tif');
```

a b
c d

FIGURE 13.39
(a) Image with numerous corners. (b) Corners detected by function `detectHarrisFeatures` with its default settings. (c) Result of increasing the value of parameter `MinQuality` by a factor of ten and increasing the size of the smoothing kernel to 91×91. (d) Further refinement by selecting the 50 strongest corners. (Image courtesy of the U. S. National Archives.)

```
>> figure, imshow(f) % Fig. 13.39(a).
>> % Use default values for all parameters.
>> corners = detectHarrisFeatures(f);
>> % Plot the corners superimposed on the image.
>> figure, imshow(f)
>> hold on
>> plot(corners) % Fig. 13.39(b).
>> hold off
```

As Fig. 13.39(a) shows, the number of corners detected is large. To find the exact number we type

```
>> size(corners)
ans =
        2768   1
```

Whether these are too many corners depends on the application. If we were looking for urban structures in an image, 2768 corners would be a clue that the image may contain a large building. In general, the default settings generate too many false corners, as is evident in the "corners" that were found in the trees, the street and the sidewalk in Fig. 13.39(b).

The other parameters available in function `detectHarrisFeatures` are designed to refine the results of this function. For example, we can increase the minimum quality of acceptable corners and smooth the image with a larger smoothing kernel. Both actions will invariably result in fewer corners:

```
>> corners2 = detectHarrisFeatures(f,'MinQuality',0.1,'FilterSize',91);
```

```
>> figure, imshow(f)
>> hold on
>> plot(corners2) % Fig. 13.39(c).
>> hold off
>> size(corners2)
ans =
        615   1
```

As Fig. 13.39(c) shows, the tighter parameters indeed resulted in fewer corners. The majority are on the building, but there are still some false corners in the trees. We could continue to experiment with quality and filtering to further reduce the number of corners, but an alternative to reducing the number of corners is to accept only a specified number of the strongest corners. For example, the following commands selected and displayed the fifty strongest corners in object `corners2`:

```
>> figure, imshow(f)
>> hold on
>> plot(corners2.selectStrongest(50)) % Fig. 13.39(d).
>> hold off
```

The result in Fig. 13.39(d) is considerably more informative, in the sense that all corners that passed the strength test are part of the building and thus are true corners in the context of this example.

MAXIMALLY STABLE EXTREMAL REGIONS (MSERs)

The Harris-Stephens (HS) corner detector discussed in the previous section is useful in applications characterized by sharp transitions of intensities, such as the intersection of straight edges that result in corner-like features in an image. Conversely, the maximally stable extremal regions (MSERs), introduced by Matas et al. [2002] and discussed in detail below, are more "blob" oriented. As with the HS corner detector, MSERs are intended to yield whole image features for the purpose of establishing correspondence between two or more images.

A grayscale image can be viewed as a topographic map, with the xy-axes representing spatial coordinates and the z-axis representing intensities. Imagine that we start thresholding an 8-bit grayscale image one intensity level at a time. The result of each thresholding is a binary image in which we show the pixels at or above the threshold in white and the pixels below the threshold in black. When the threshold, T, is 0, the result is a white image because all pixel values are at or above 0. As we start increasing T in increments of one intensity level, we will begin to see black components in the resulting binary images. These correspond to local minima in the topographic map view of the image. These black regions may begin to grow and merge, but they will never get smaller from one image to the next. Finally, when we exceed $T = 255$, the resulting image will be black—there are no pixel values above this level. Because each stage of thresholding results in a binary image, there will be one or more connected components of white pixels in each image. The set of all such components resulting from all thresholding operations is the set of *extremal regions*. Extremal regions that do not change size (number of pixels) appreciably over a range of threshold values are called *maximally stable extremal regions*.

TABLE 13.16
Fields of the structure `cc` output by the function `detectMSERfeatures`.

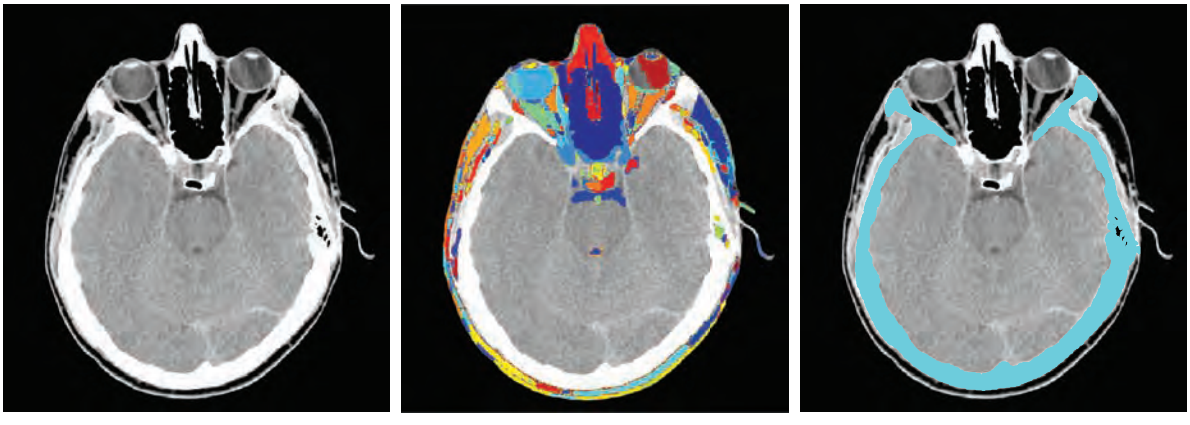
Fields	Explanation
`Connectivity`	Connectivity of the MSER regions. The default is 8.
`ImageSize`	Size of image `f`.
`NumObjects`	Number of MSER regions in `f`.
`PixelIdxList`	`1-by-NumObjects` cell array containing `NumObjects` vectors. Each vector contains the linear indices of the pixels in the element's corresponding MSER region.

```
>> disp(regions)
  499×1 MSERRegions array with properties:
          Count: 499
       Location: [499×2 single]
           Axes: [499×2 single]
    Orientation: [499×1 single]
      PixelList: {499×1 cell}
```

The total number of MSER regions detected was 499, many of which were very small. Note that the default settings failed to detect the white region (a cross section of the skull) enclosing the brain.

Suppose next that we want to extract the skull region and exclude all other regions in the image. We know from Table 13.15 that the default area range is [30,14000]. The white region was not detected because its area is greater than this. We can try a new range: [14001,28000] and see if the area of the region is in that range:

```
>> [regions,cc] = detectMSERFeatures(f,'RegionAreaRange',[14001 28000]);
```



a b c

FIGURE 13.41 (a) Image from a CT scan of a human head. (b) The 499 MSER regions detected using the defaults of function `detectMSERFeatures`. (c) Region detected with parameter `'RegionAreaRange'` set to `[14000 28000]`. (Original image courtesy of Dr. David R. Pickens, Vanderbilt University.)

a b

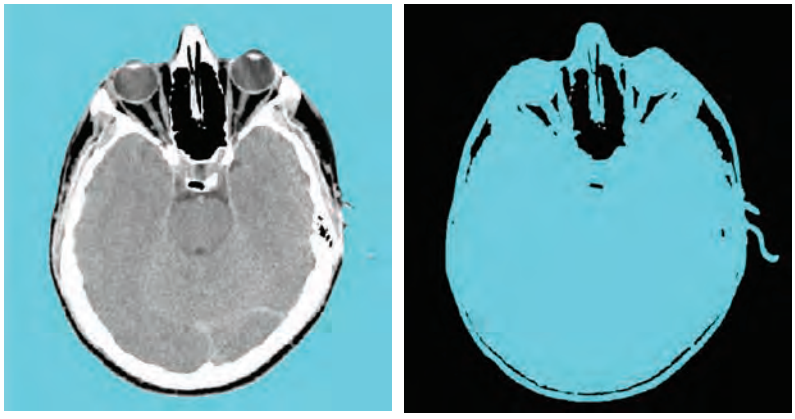


FIGURE 13.43 The (a) smallest, and (b) largest MSERs in the range `[14001,... 0.5*imarea]`. The MSERs are shown in cyan.

KEYPOINT FEATURES

Keypoints are spatial image points that are characteristic of on image or class of images, in the sense that no matter how the image(s) is (are) transformed (e.g., by rotation, shrinking, expanding, translation, and changes in intensity and view point) you should be able to find the same keypoints that you found in the original image(s). When coupled with descriptors, keypoints are referred to as *keypoint features*. Corners are an example of keypoint features if we couple them with descriptors such as the dominant direction of the gradients computed in a neighborhood of each corner point.

Keypoint features based on corners and MSERs are suitable for applications in which variability between images is limited. However, in the presence of variables such as those mentioned in the previous paragraph, we are forced to look at more comprehensive techniques designed to achieve invariance to as many of those variables as possible, thus yielding so-called *robust keypoint features*. Because methods for extracting keypoint features are complex, computational speed is a fundamental requirement. Central to the usefulness of any keypoint-feature-based scheme is *repeatability*; that is, the ability to find the same keypoints across images obtained under different viewing conditions.

Among the many uses of keypoints in image processing are image registration, camera calibration, image stitching to generate panoramic views, object recognition, image retrieval, and map-based autonomous navigation. We demonstrated the use of keypoints for image registration in Chapter 6. In this section, we summarize some of the fundamental aspects of keypoint feature extraction and discuss several functions for implementing and using keypoint features. As with earlier methods in this section, keypoint detection and description functions are from the Computer Vision Toolbox.

```
>> matchedPoints1 = valpoints1(indexPairs(:,1),:);
>> matchedPoints2 = valpoints2(indexPairs(:,2),:);
```

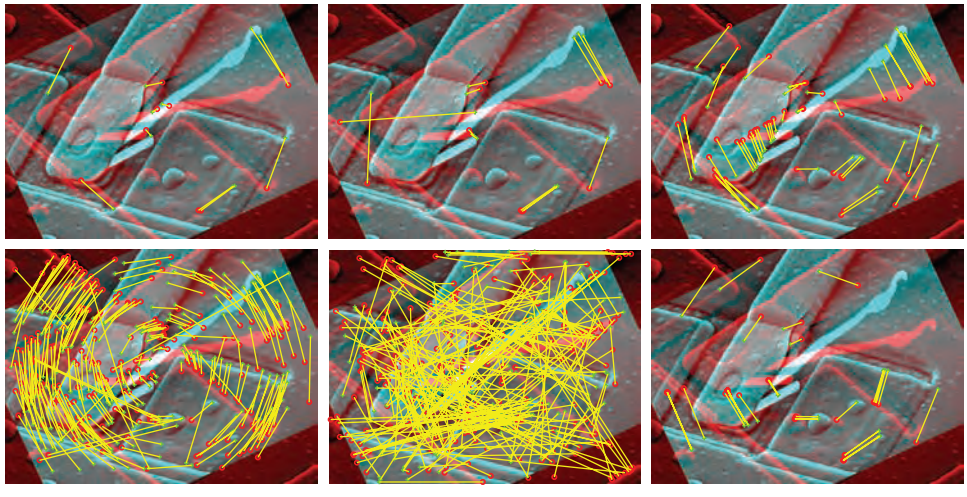
We can investigate the contents of these results as follows:

```
>> disp(matchedPoints1)
  11×1 cornerPoints array with properties:
    Location: [11×2 single]
      Metric: [11×1 single]
       Count: 11
```

Thus, we see that there are 11 matched features whose coordinates are contained in the 11×2 matrix `matchedPoints1.Location`. To display the matches we type

```
>> figure, showMatchedFeatures(f1,f2,matchedPoints1,matchedPoints2) % Fig. 13.50(a)
```

As Fig. 13.50(a) shows, the 11 keypoint matches found are between points that are clearly corresponding. In particular, two matches were found in the high-contrast white protrusions mentioned earlier. The rest of Fig. 13.50 shows results with the keypoint features listed in the figure caption, using commands very similar to those we used to obtain Fig. 13.50(a). The result in (b) is also based on corners and it found three matches between the protrusions. However, this method produced a matching error between the points shown joined by the long line. The BRISK result in (c) produced more correct matches than the previous two methods. This is not surprising, considering that the BRISK method is based on multi-level detection, while the corner methods just work on the image plane. SURF keypoint features gave the "richest" matching result. It has a significant number of correct matches with few errors. In general numerous correct keypoint matches imply a higher probability that any two images being compared are of the same object or scene. The KAZE result is useless for all practical purposes, as you can see by the randomness of the matches and the lengths of the lines connecting them. The diffusion model on which KAZE is based is not applicable in this case. Finally, MSER keypoint features found several correct matches, but failed to find correspondence between the white protrusions mentioned earlier.



a b c
d e f

FIGURE 13.50 Matches found between the two images in Fig. 13.49 using: (a) FAST, (b) Harris, (c) BRISK, (d) SURF, (e) KAZE, and (f) MSER keypoint features.

Summary

Feature extraction is a fundamental process in the operation of most automated image processing applications. As indicated by the range of feature detection and description techniques covered in this chapter, the choice of one method over another is determined by the problem under consideration. The objective is to choose features that "capture" essential differences between objects, or classes of objects, while maintaining as much independence as possible to changes in variables such as location, scale, orientation, illumination, and viewpoint.

MATLAB Projects

*Solutions to the projects marked with an asterisk * are in the DIPUM3E Student Support Package (consult the book web site). All your code must be documented so that typing* `help` *at the prompt, followed by the script or function name, gives enough detail for a user to be able to run it. Test the functionality of all your code thoroughly.*

13.1 In the following, selecting regions, holes, or boundaries by inspection is not acceptable. All answers must be based on computations.

(a)* Read the image `multiple-regions.tif` and extract the boundary of the smallest hole. Display the boundary as an image.

(b) Extract all the pixels comprising the smallest hole and display them as an image. Boundary pixels are not considered part of the hole, so they should not appear in your image.

(c) Extract the boundary of each of the regions with no holes and display it as an image.

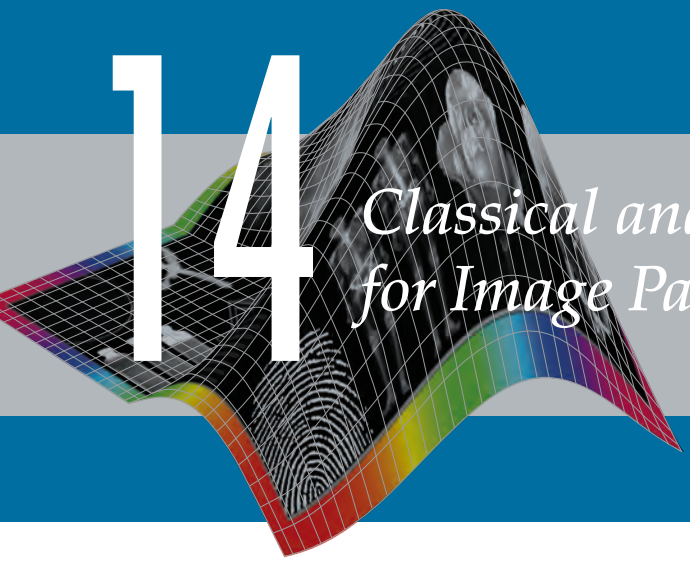
(d) Extract only the outer boundaries of the regions in the image and show them superimposed on the regions, using a different color for each boundary.

13.2 Do the following:

(a)* Start with the small image `I1 = zeros(11)` and `I1(5:7,4:7) = 1`. Then form `I2` as `I2 = I1'` (i.e., `I1` rotated by 90°). Process each image with function `freemanChainCode` to obtain structures `c1` and `c2`. Explain the following: (1) Why are the integers of minimum magnitude `c1.mm` and `c2.mm` different? (2) Why are `c1.diff` and `c2.diff` different? (3) Why are the integers of minimum magnitude `c1.diffmm` and `c2.diffmm` equal?

(b)* Write a function to determine if a given Freeman chain code corresponds to a closed curve or not. The function specifications are as follows:

```
function oc = isCodeClosed(fcc,conn)
%isCodeClosed Determines if a Freeman chain code is of a closed curve.
%   OC = isCodeClosed(FCC,CONN) determines if Freeman chain code FCC
%   corresponds to a closed curve with connectivity CONN (4 or 8, the
%   latter being the default). Output OC is 1 if the curve is closed
%   and 0 otherwise.
%
%   The following table lists the changes in deltax and deltay to
%   transition from a point in the direction of the code symbol. The
%   origin is based on our image coordinate system, with the origin on
%   the top, left (see Fig. 2.1). The index is used in the body of the
%   function to determine the appropriate deltax and deltay to use from
%   one element of the code to the next.
%
%          -----------------------------------------------
```



14 Classical and Deep Learning Methods for Image Pattern Classification

Art is the imposing of a pattern on experience, and our aesthetic enjoyment is recognition of the pattern.
Alfred North Whitehead

We conclude our coverage of digital image processing with an introduction to techniques for image pattern classification. The approaches developed in this chapter are divided into three principal categories: classification using prototype matching, classification based on an optimal statistical formulation, and classification based on neural networks. The first two approaches are used extensively in applications in which the nature of the data is well understood, leading to an effective pairing of features and classifier design. These approaches often rely on a great deal of engineering to define features and elements of a classifier. Approaches based on neural networks rely less on such knowledge and lend themselves well to applications in which pattern class characteristics (e.g., features) are learned by the system from massive databases, rather than being specified a priori by a human designer. The focus in this chapter is on principles and how to write functions for implementing classification methods using MATLAB and the Image Processing Toolbox. We also use a few functions from the MATLAB Deep Learning and Computer Vision Toolboxes.

Functions Developed in this Chapter:

- `mahalanobis` implements a highly vectorized computation of the Mahalanobis distance.
- `minDistanceClassifier` implements a minimum distance classifier.
- `bayesgauss` implements a Bayes classifier for Gaussian pattern classes.
- `strsimilarity` computes measures of similarity between pattern strings.
- `randvertex` randomizes the location of polygon vertices.
- `polyangles` computes the interior angles of a set of polygon vertices.
- `perceptronTrain` implements the perceptron training algorithm.
- `patternShuffle` shuffles the order of pattern vectors.
- *fcnn Functions* is a suite of custom functions for training fully-connected neural networks and for using them for pattern vector classification.
- *cnn Functions* is a suite of custom functions for training convolutional neural networks and for using them for image pattern classification.

```
%   input data can be real or complex. The outputs are real quantities.
%
%   D = MAHALANOBIS(Y,CX,MX) computes the Mahalanobis distance between
%   each vector in Y and the given mean vector, MX. The results are
%   output in vector D, whose length is size(Y, 1). The vectors in Y are
%   assumed to be organized as the rows of this array. The input data
%   can be real or complex. The outputs are real quantities. In addition
%   to the mean vector MX, the covariance matrix CX of a population of
%   vectors X must be provided also. Uses custom function COVMATRIX.

% Preliminaries.
param = varargin;
Y = param{1}; % param is a cell array.
if length(param) == 2
   X = param{2};
   % Compute the mean vector and covariance matrix of the vectors in X
   % using DIPUM3E custom function covmatrix.
   [Cx,mx] = covmatrix(X);
elseif length(param) == 3 % Cov. matrix and mean vector provided.
   Cx = param{2};
   mx = param{3};
else
   error('Wrong number of inputs.')
end

% Make sure that mx is a row vector for the next step.
mx = mx(:)';

% Subtract the mean vector from each vector in Y.
Yc = Y - mx;

% Compute the Mahalanobis distances.
D = real(sum(Yc/Cx.*conj(Yc),2));
```

The MATLAB matrix operation `A/B` is more accurate (and generally faster) than the operation `A*inv(B)`. Similarly, `A\B` is preferred to `inv(A)*B`. See Table 2.8.

The call to `real` in the last line of code is to remove "numeric noise" arising from complex-number computations in earlier versions of MATLAB. If the data are known to be real, the code can be simplified by removing functions `real` and `conj`.

14.3 PATTERN MATCHING CLASSIFIERS

In this section we discuss two of the earliest approaches to image pattern classification, both of which are based on matching an unknown pattern against two or more prototypes whose classes are known. The first, generally referred to as a *minimum-distance*, or *nearest neighbor*, *classifier* works with pattern vectors. The second, referred to as an *image correlation classifier*, works with images directly.

MINIMUM-DISTANCE CLASSIFIER

Suppose that we have N_c pattern classes, $c_1, c_2, \ldots, c_{N_c}$, of vectors and we characterize each class by a single *prototype vector*, which we can set equal to the *mean vector* of the population:

The number of elements in all three cases is `100`, so the entire training set was classified correctly. Later in this chapter you will learn more elegant ways to determine classification accuracy.

The preceding data set can be made more realistic by rotating the objects in random directions and scaling their sizes. Also, we normally work with a training set to learn the system parameters (mean vectors in this example) and an independent test set to determine system performance with patterns it has never "seen" before. You will work with these additional requirements in Project 14.1.

The matching of *keypoint features* discussed in the previous chapter also utilizes the concept of minimum-distance classification of feature vectors, but often using simplifications to gain speed. See Example 13.17 for an illustration of keypoint matching.

2-D IMAGE MATCHING USING CORRELATION

We introduced the basic idea of spatial correlation and convolution in Section 3.4 and used these concepts extensively in Chapter 3 for spatial filtering. From Eq. (3-12) we know that correlation of a kernel w with an image $f(x, y)$ is given by

$$(w \star f)(x, y) = \sum_s \sum_t w(s, t) f(x + s, y + t) \tag{14-14}$$

where the limits of summation are taken over the region shared by w and f. This equation is evaluated for all values of the displacement variables x and y so that all elements of w visit every pixel of f. Correlation has its highest value(s) in the region(s) where w and f are equal. In other words, Eq. (14-14) finds locations where w matches a region of f. But this equation has the drawback that the result is sensitive to changes in the amplitude of either function. In order to normalize correlation to amplitude changes in one or both functions, we perform matching using the *correlation coefficient* instead:

To be formal, we should refer to correlation (and the correlation coefficient) as *cross-correlation* when the functions are different and as *autocorrelation* when they are the same. However, it is customary to use the generic term *correlation* and *correlation coefficient*, unless the distinction is important—as in deriving equations where it makes a difference which is being applied.

$$\gamma(x, y) = \frac{\sum_s \sum_t \left[w(s,t) - \bar{w}\right]\left[f(x+s, y+t) - \bar{f}_{xy}\right]}{\left\{\sum_s \sum_t \left[w(s,t) - \bar{w}\right]^2 \sum_s \sum_t \left[f(x+s, y+t) - \bar{f}_{xy}\right]^2\right\}^{\frac{1}{2}}} \tag{14-15}$$

where the limits of summation are taken over the region shared by w and f, $\bar{w}$ is the average value of the kernel (computed only once), and $\bar{f}_{xy}$ is the average value of f in the region coincident with w. In image correlation work, w is often referred to as a *template* and correlation is referred to as *template matching*.

Templates are also referred to as *prototypes* or *subimages*.

It can be shown (Gonzalez and Woods [2018]) that $\gamma(x, y)$ has values in the range $[-1, 1]$ and is thus normalized to changes in the amplitudes of w and f. The maximum value of γ occurs when the normalized w and the corresponding normalized region in f are identical. This indicates *maximum correlation* (the best possible match). The *minimum* occurs when the two normalized functions exhibit the least similarity in the sense of Eq. (14-15). Although this equation is normalized to provide invariance

a b
c d

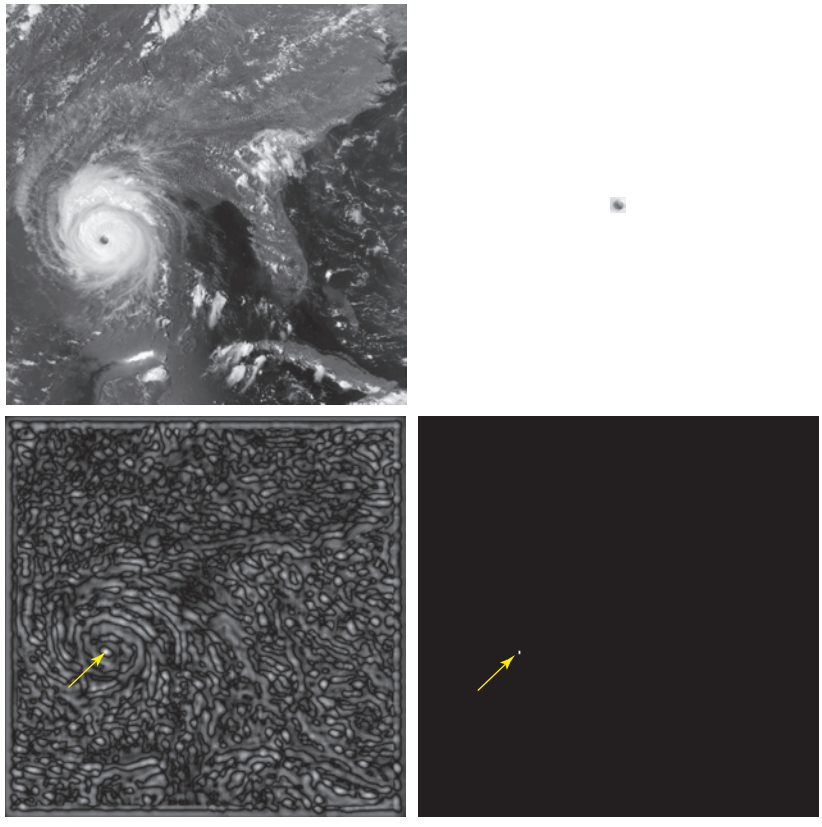


FIGURE 14.4
(a) Image of Hurricane Andrew. (b) Template. (c) Correlation of image and template. (d) Location of the best match. (Original image courtesy of NOAA.)

```
>> figure, imshow(f)  % Fig. 14.4(a).
>> figure, imshow(w)  % Fig. 14.4(b).

>> % Compute the correlation coefficient.
>> g = abs(normxcorr2(w,f));
>> figure, imshow(g,[]) % Fig. 14.4(c).

>> % Find all the max values.
>> gT = g == max(g(:)); % gT is a logical array.
>> % Find out how many peaks there are.
>> idx = find(gT == 1); % We use idx again later in this example.
>> disp(numel(idx))
        1
>> % A single point is hard to see. Increase its size using dilation.
>> gT = imdilate(gT,ones(7));
>> figure, imshow(gT) % Fig. 14.4(d).
```

The blurring in the correlation image in Fig. 14.4(c) should not be a surprise because the template in Fig. 14.4 (b) has two dominant, nearly constant regions and thus behaves like a lowpass filter kernel.

14.5 FEEDFORWARD FULLY-CONNECTED NEURAL NETWORKS

In this section we discuss the architecture and operation of *fully-connected neural networks* (FCNNs) in which the propagation of an input pattern to the output of the network occurs in a feedforward direction. This is as apposed to *recurrent neural networks* (RNNs), which can have internal loops. RNNs are outside the scope of the present discussion. The term *fully-connected* means that the output of *each* node in a layer of the network feeds into the input of *every* node in the next layer.

MODEL OF AN ARTIFICIAL NEURON

FCNNs are interconnected perceptron-like computing elements called *artificial neurons*. These neurons perform the same computations as the perceptron, but they differ from the latter in how they process the result of the sum-of-products computation, which we denoted by z in Eq. (14-23). As illustrated in Fig. 14.7, the perceptron uses a "hard" thresholding function that outputs two values, such as +1 and −1, to perform classification. A hard threshold can have large swings between its limits for infinitesimally-small changes in the input to the thresholder. Because FCNNs are formed by layering computing units, the output of one unit affects the behavior of all units following it. The perceptron's sensitivity to the sign of small signals can cause serious stability problems in an interconnected system, making perceptrons unsuitable for layered architectures.

The solution is to change the hard-limiter to a smooth function. Instead of being binary, the output of an artificial neuron is a function, $h(z)$, of the sum-of-products computation, z. We denote the output value of the ith neuron in layer ℓ of a network by $a_i(\ell) = h\left[z_i(\ell)\right]$. As before, we refer to h as an *activation function* and call the output value of a neuron its *activation value*. Figure 14.13 is a schematic of an artificial based on this notation. Comparing it with Fig. 14.7, we see that the form of the computation is the same, with the exception of more complex subscripted notation—to be explained shortly—and the fact that we now denote the *bias* term, w_{n+1}, by b instead. The inputs to the ith neuron in layer ℓ are denoted by $a_k(\ell-1)$, for $k = 1, 2, \ldots, n_{\ell-1}$, where $n_{\ell-1}$ is the number of neurons in layer $\ell-1$. The ith neuron in layer ℓ has a single output, $a_i(\ell)$. The most important thing to note for now is that

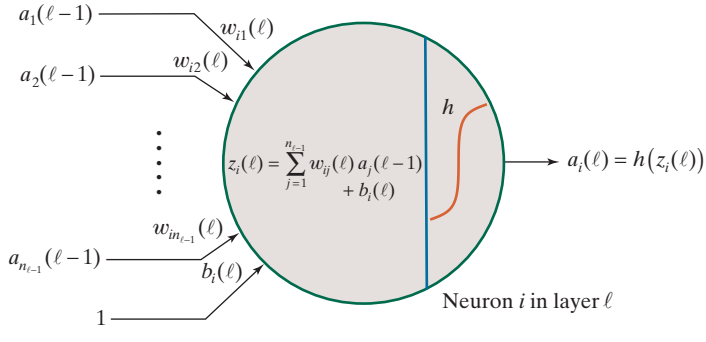


FIGURE 14.13 Model of an artificial neuron, showing all the operations it performs. The "ℓ" denotes a particular layer in a layered network. The "i" denotes the ith neuron.

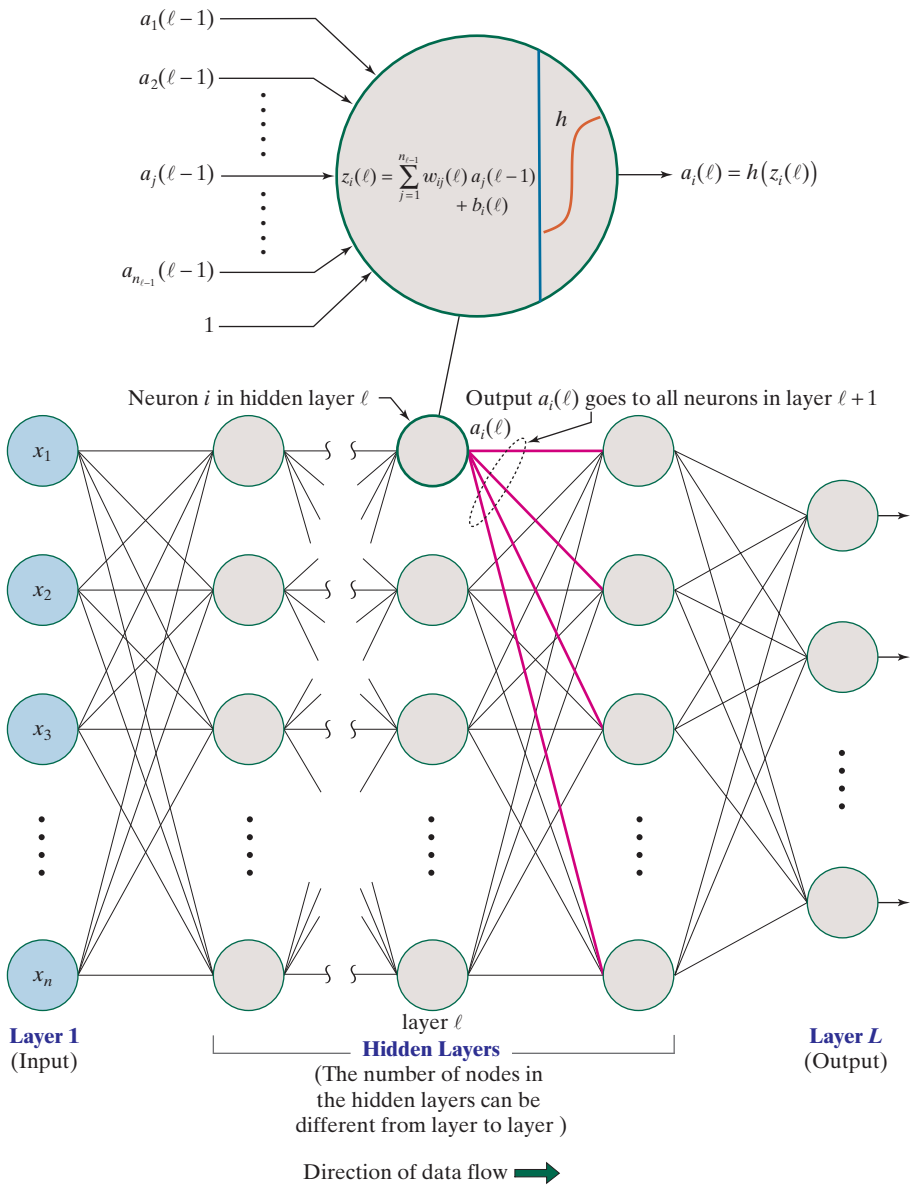


FIGURE 14.15
Model of a feedforward, fully connected neural net. The neurons are the same as in Fig. 14.13. Note how the output of each neuron goes to the inputs of all neurons in the following layer, hence the name *fully connected* for this type of architecture.

node in the figure, all the nodes in the network are artificial neurons of the form shown in Fig. 14.13, except for the input layer, whose nodes are the components of an input pattern vector $\mathbf{x}$. Therefore, the outputs (activation values) of the first layer are the values of the elements of $\mathbf{x}$. The outputs of all other nodes are the activation values of neurons in a particular layer. Each layer in the network can have a

USING BACKPROPAGATION TO TRAIN FCNNs

As noted earlier, when we refer generically to "weights," we mean both weights *and* biases.

If you want to skip the details, the results of this section are summarized in Table 14.8

An FCNN is defined completely by its weights and (typically) one specified activation function used by all neurons. Training of such a network refers to using one or more sets of training patterns to estimate the weights. During training, we know the desired response of every neuron in the output layer, but we have no way of knowing what the outputs of the hidden neurons should be. In this section we discuss and implement the method of *backpropagation*. As we mentioned previously, backpropagation was the breakthrough in 1986 that established a procedure for training an FCNN so that it learns all its weights by cycling through training data.

Backpropagation consists of: (1) a feedforward pass to classify all the patterns of the training set and compute the classification error; (2) a backward (backpropagation) pass that feeds the output error back through the network to compute the changes required to update the weights; and (3) the updating all the weights in the network. These steps are repeated until the classification error reaches an acceptable level.

The Equations of Backpropagation

Given a set of training patterns and a multilayer feedforward neural network architecture, the approach in the following discussion is to find the network weights that minimize an *error* (also called a *cost* or *objective*) *function*.

For an input pattern, the activation value of neuron j in the output layer of an FCNN is $a_j(L)$. We define the error of that neuron as

$$E_j = \frac{1}{2}\left(r_j - a_j(L)\right)^2 \tag{14-48}$$

The *desired response* of an FCNN is for the output neuron corresponding to the class of the input to have the highest activation value.

for $j = 1, 2, \ldots, n_L$, where, as we defined when discussion perceptrons, r_j is the desired response of output neuron $a_j(L)$ for a given input pattern $\mathbf{x}$. The output error with respect to a single $\mathbf{x}$ is the sum of the errors of all output neurons with respect to that vector:

In this formulation, $\mathbf{r}$ is an $N_c \times 1$ class membership *column* vector whose kth element is 1 if the pattern it represents belongs to class c_k. All other elements of $\mathbf{r}$ are 0. The fact that we are now dealing with more than two classes requires this change from the way we defined $\mathbf{r}$ for perceptrons. As before, N_c denotes the number of classes.

$$\begin{aligned} E &= \sum_{j=1}^{n_L} E_j = \frac{1}{2}\sum_{j=1}^{n_L}\left(r_j - a_j(L)\right)^2 \\ &= \frac{1}{2}\left\|\mathbf{r} - \mathbf{a}(L)\right\|^2 \end{aligned} \tag{14-49}$$

where the second line follows from the definition of the Euclidean vector norm. The *total network output error* over all training patterns is defined as the sum of the errors of the individual patterns. We want to find the weights that minimize this total error. As we did for the LMSE perceptron, we find the solution using gradient descent. This implies an expression of the form in Eq. (14-32) that gives the gradient of the error with respect to the weights and biases as a function of observable responses. However, the only quantities we can observe in an FCNN are the activation values of the input and output neurons, so we cannot compute the gradients for the neurons of the hidden layers. Backpropagation gives us way to obtain these

TABLE 14.9
FCNN custom functions included in the *DIPUM3E Support Package*.

Function Name	Explanation
`fcnninfo`	Contains information about variables and constants used in the following functions.
`fcnninit`	Initializes an FCNN.
`fcnnff`	Implements feedforward.
`fcnnbp`	Implements backpropagation.
`fcnnactivate`	Computes neuron activation values.
`fcnntrain`	Trains an FCNN.
`fcnnupdateweights`	Updates the weights during training.
`fcnnclassify`	Classifies unknown pattern vectors.


```
classifieroutput = fcnnclassify(fcnn,X,R)
```

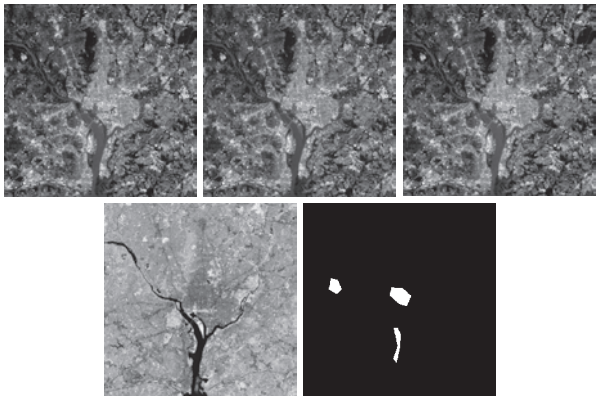
where `fcnn` is a trained FCNN and `classifieroutput` is a structure explained in the help section of `fcnninfo`.

The FCNN functions we have discussed thus far are the core of training and classification. They work in conjunction with the other functions listed in Table 14.9 to form a set of FCNN functions that is included in your Support Package. The following example illustrates how to use these functions.

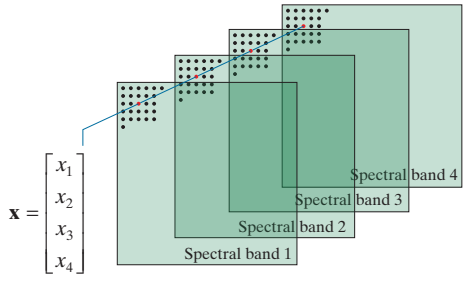
EXAMPLE 14.9: Using the FCNN functions for multispectral data classification.

In this example we use an FCNN to solve the multispectral data classification problem we solved in Example 14.3 using a Bayes classifier. In that example, training consisted of using training data to estimate the covariance matrix and mean vectors of three pattern classes. In this example, we train an FCNN directly using the same training set. The following commands read and format the data:

```
>> % As in Example 14.3 read the images and masks, stack the images, and extract
>> % the vectors of the training and test sets:
>> fileNames = {'washDC-band1-blue.tif','washDC-band2-green.tif',...
                          'washDC-band3-red.tif','washDC-band4-nrinfrared.tif'};
>> maskNames = {'washDC-mask-water.tif','washDC-mask-urban.tif',...
                                                    'washDC-mask-veg.tif'};
>> % Class 1 = water, class 2 = urban, class 3 = vegetation.
>> Nc = 3;
>> for k = 1:length(fileNames)
      f{k} = im2double(imread(fileNames{k})); % Fig. 14.17.
end
>> for k = 1:length(maskNames)
   mask{k} = im2double(imread(maskNames{k}));
end
>> % Form the image stack.
>> imstack = cat(3,f{1},f{2},f{3},f{4});
```



Images in spectral bands 1–4 and binary mask used to extract training samples



Approach used to extract pattern vectors

a b

FIGURE 14.17 (a) Starting with the leftmost image: blue, green, red, near infrared, and binary mask images. In the mask, the lower region is for water, the center region is for the urban area, and the left mask corresponds to vegetation. All images are of size 512×512 pixels. (b) Approach used for generating 4-D pattern vectors from a stack of the four multispectral images. (Multispectral images courtesy of NASA.)

```
>> % Extract training and test pattern sets.
>> for k = 1:Nc
      [X{k},~] = imstack2vectors(imstack,mask{k});
      % Training patterns.
      trainX{k} = X{k}(1:2:end,:)';
      % Number training patterns for each class.
      ntrain{k} = size(trainX{k},2);
      % Test patterns.
      testX{k} = X{k}(2:2:end,:)';
      % Number test patterns for each class.
      ntest{k} = size(testX{k},2);
end
>> % Number of training and test patterns.
>> nptrain = sum([ntrain{:}]);
>> nptest = sum([ntest{:}]);

>> % Form the pattern matrices.
>> % Training patterns.
>> Xtrain = cat(2,trainX{1},trainX{2},trainX{3});
>> nptrain = size(Xtrain,2); % Number of patterns.
>> % Test patterns.
>> Xtest = cat(2,testX{1},testX{2},testX{3});
>> nptest = size(Xtest,2); % Number of patterns.

>> % Construct membership matrices.
>> % Training patterns.
>> Rtrain = zeros(Nc,nptrain);
>> Rtrain(1,1:ntrain{1}) = 1;
```

14.6 CONVOLUTIONAL NEURAL NETWORKS

Thus far, we have worked with feature (pattern) vectors. The form of those features has been specified a priori (i.e., "engineered" by a human designer) and extracted from images prior to being input to a neural network. But, as you will see shortly, one of the strengths of neural networks is that they are capable of learning pattern features *on their own*—directly from training data. The approach is to input a set of training images into a neural network and have the network learn the necessary features during training. A significant advantage of this approach over vector representations is that it exploits spatial relationships that may exist between pixels in an image, such as pixel arrangements into corners, the presence of edge segments, and other features that may help differentiate one image from another. In this section, we present a class of neural networks called *deep convolutional neural networks* (*CNNs* or *ConvNets* for short) that accept images as inputs and interface with an FCNN whose function is to determine the class membership of each input image.

Figure 14.20 shows a model of the CNN/FCNN architecture used in this section for *image classification*. That is, this system accepts image inputs and outputs a label for each input image. You can think of the CNN as being a *feature extractor* and the FCNN as being the pattern classifier. The reason for using convolution is that convolution produces filtered images that exhibit features such as edges and smoothed regions. You will see shortly that CNNs are capable of exactly this type behavior. We discussed the operation of FCNNs in the previous section, so we focus initially on the operation of the components to the left of the Interface in Fig. 14.20. The inputs and outputs of a CNN are two-dimensional. As you can see from the figure, the outputs of the last layer of the CNN are converted to vectors of the form required for input into an FCNN.

All layers of our CNN model have the same basic architecture. Figure 14.21 shows the components of one layer of our model. The inputs and outputs are referred to as *maps* implying that they are two-dimensional. A registered stack of such maps is called a *maps volume*. The input maps volume to the first layer is the set of component images of a multispectral image (e.g., the three components of an RGB image). The input maps to all subsequent layers are the output maps of the previous layer. The computational processes in a layer are shown in purple and the data components are shown in green. Thus, we see that input maps are processed by convolution to which we add a bias. The results are then passed through an activation function identical to the ones we discussed in the previous section. Additional processing might consist of computations such as data normalization. The resulting activation values form so-called *feature maps*, for reasons that will be obvious when you see an example later in this section. The components of each feature map over a small neighborhood are pooled (e.g., by averaging their values) to create a feature map of lower resolution called a *pooled feature map*, or *pooled map* for short. These pooled maps are the outputs of each CNN layer and become the inputs to the next layer. We refer to the processes and feature maps volume in a layer of the CNN as a *convolutional layer*, but this terminology varies widely in the literature on CNNs.

The data that propagate through a CNN are two dimensional, so the nature of the computation in each layer is not evident in the "flat" diagram in Fig. 14.21.

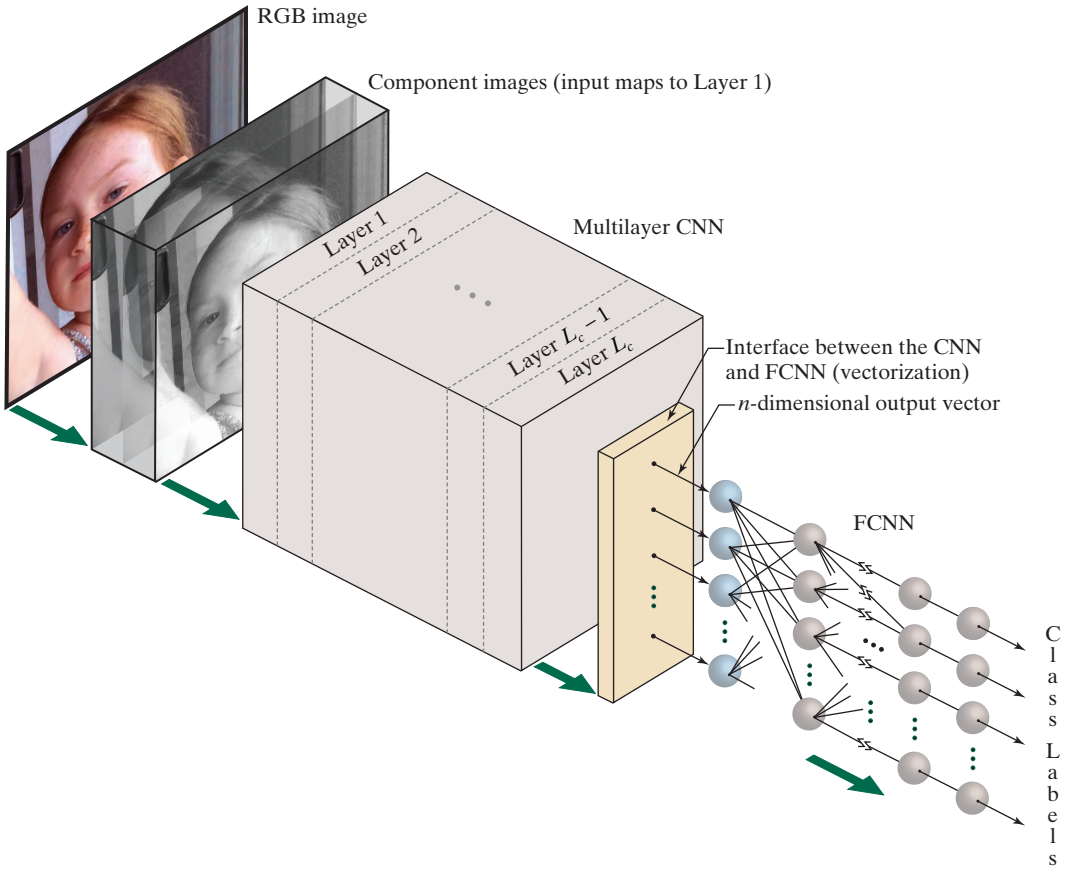


FIGURE 14.20 General model of the CNN/FCNN architecture we use in this chapter for image classification.

A CNN *volume* is a set of *registered* 2-D data *maps*, accessed across the *depth* of the volume.

Figure 14.22 shows the same flow of operations and data in 3-D perspective. All computations shown are for *one* spatial location, (x, y). For simplicity, we excluded any additional processing stages.

A multispectral input image is first decomposed into its component images, the set of which we call an *input maps volume*, whose *depth* is equal to the number of

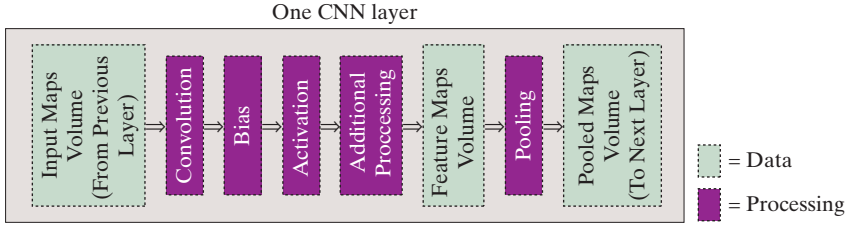


FIGURE 14.21 Data and processing components in one layer of a multilayer CNN.

map. The output values of the neurons in the pooled feature maps are generated by pooling the output values of the neurons in the feature maps. The outputs of the neurons in the pooled maps (or feature maps when no pooling is used) become the neurons of the input to the next stage. Because feature maps are the result of spatial convolution, we know from Chapter 3 that they are simply filtered images. It then follows that pooled feature maps are filtered images of lower resolution. Also, as noted earlier, filtered images exhibit behavior that we have learned to associate with image features, such as edges and blurred regions. Directional edge detection learned by the CNN from training data certainly is a key feature in this example.

The second row in Fig. 14.24 illustrates visually how feature maps and pooled maps look based on the input image shown in the figure. The kernel shown is as described in the previous paragraph and its weights (shown as intensity values) were learned from sample images during training of the CNN described later in Example 14.13. Therefore, the nature of the learned features is determined by the learned kernel coefficients. Note that the contents of the feature maps are specific features detected by convolution. For example, some of the features emphasize edges in the character. As mentioned above, the pooled features are lower-resolution versions of this effect.

Figure 14.24 also shows that in a CNN the into every neuron in a feature map a *single* value, determined by the convolution over a *spatial neighborhood* in the previous layer. This is unlike an FCNN, in which we feed the output of *every* neuron in a layer directly into the input of *every* neuron in the next layer. Therefore, *CNNs are not fully connected* in the sense defined in the last section.

EXAMPLE 14.11: Computational example showing the types of features that can be extracted by a CNN.

Figure 14.25 illustrates the types of features that volume convolution is able to extract. The input to the CNN stage is an RGB image of size 277×277 pixels. Its three component images form an input volume of depth three. We used the image of a human subject as the input so that the resulting feature maps would be easier to interpret visually.

The feature maps volume in this case contained 96 feature maps, each obtained by filtering the maps of the input volume with a different kernel volume of size $11 \times 11 \times 3$. Thus, there are 96 kernel volumes of depth three, composed of $3 \times 96 = 288$ 2-D convolution kernels of size 11×11, for a total of 34,848 kernel weights. These weights came from AlexNet, a CNN trained using more than 1 million images belonging to 1,000 object categories (see Krizhevsky, Sutskever, and Hinton [2012]). The system had never "seen" the image we used in this example.

The 96 feature maps resulting from the input image are shown as an 8×12 montage of 2-D images in the upper right of Figure 14.25. Several feature maps are shown zoomed, numbered, and grouped to illustrate the variety of complementary features that can result from volume convolution. The first group shows three feature maps. Two of them (4 and 35) emphasize edge content and the third (23) is a blurred version of the input. The second group has two maps (10 and 16) that capture complementary shades of gray (note the difference in the hair intensity, for example). In the third group, feature map 39 emphasizes the subject's eyes and dress, both of which are blue in the input RGB image. Map 45 emphasizes blue too, but it also emphasizes areas that correspond to red tones in the RGB image, such as the subject's lips, hair, and skin. These two feature maps are more sensitive to color content than the maps in the other two groups. Here you see again the fact that convolution resulted in filtered images with features that are distinctive. In subsequent examples later in this section, we will illustrate how feature maps appear as they propagate through other stages of a CNN.

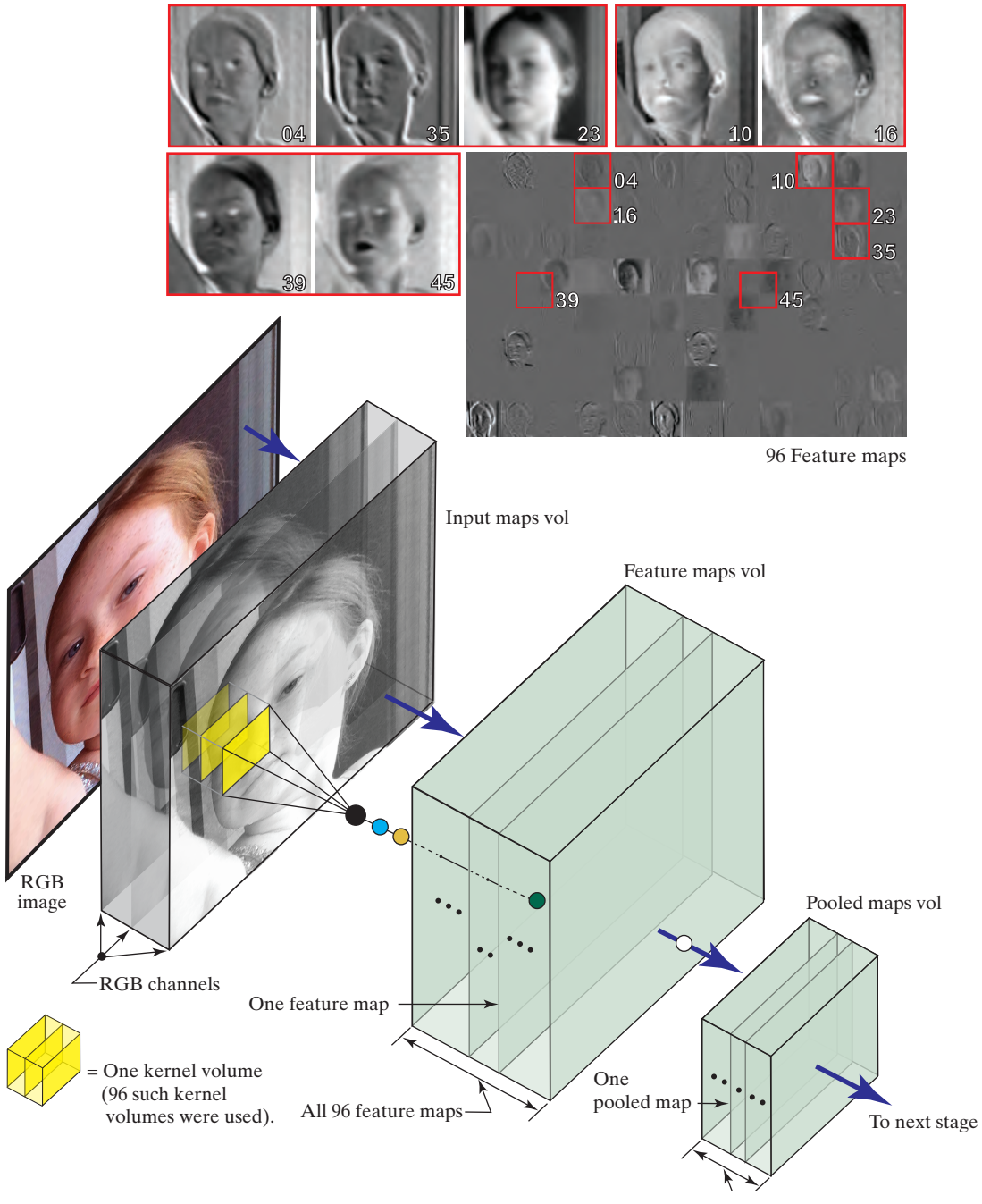


FIGURE 14.25 Detailed view of the first stage of a CNN, showing the feature maps as images. The highlighted, numbered feature maps shown zoomed at the top, illustrate the types of features extracted by the 96 kernel volumes. The operations performed at the point of the color bullets are explained in Fig. 14.22.

EXAMPLE 14.12: Illustration of how to train a CNN and use it for image classification.

Figure 14.26 shows the simplest possible form of the general model in Fig. 14.20. The CNN has a single 2-D kernel, so it can only process grayscale images, and the FCNN has no hidden layers, which makes it a linear classifier. In this example, we train this system using 150 images, fifty each of the three types of noisy stars shown in Fig. 14.27. The images we used are of size 454×454 but we subsampled them to size 32×32 for consistency with the size of the images we will use throughout the remainder of this section (we upsampled the small images to size 300×300 when displaying them in a figure). We trained and tested three systems with the architecture shown in Fig. 14.26. First, we used images corrupted with additive Gaussian noise of zero mean and standard deviation of 0.01, which is equivalent to 0.1 intensity levels on the [0,1] intensity scale of the images. We trained the other two systems using images corrupted by noise of 0.32 and 0.5 intensity levels, respectively. The images in Fig. 14.27 are samples of the training sets we used. After training each system, we tested it using 900 noisy images, 300 each of the three types of stars. The trained systems had never "seen" these images before.

We trained and tested the system with the first dataset using the following commands.

```
>> %% READ THE IMAGES, REDUCE THEM TO SIZE 32-BY-32, AND CONSTRUCT THE MEMBERSHIP MATRIX.
>> I0 = [];
>> fileNames = {'wingding-star-3pt.tif','wingding-star-5pt.tif','wingding-star-8pt.tif'};
>> for k = 1:length(fileNames)
      I0(:,:,k) = mat2gray(imresize(im2double(imread(fileNames{k})),[32 32]));
end

>> % Number of images.
>> NI = size(I0,3);

>> % Membership matrix for the images in I0.
>> R0 = [
        1 0 0
        0 1 0
        0 0 1
        ];

>> %% CREATE A NOISY TRAINING SET.

>> % Create a group of images by concatenating I0 NT times. The resulting
```

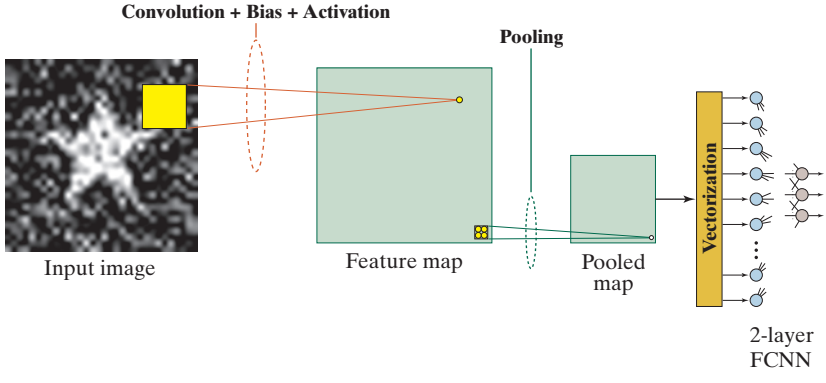


FIGURE 14.26 CNN/FCNN used to classify the images in Fig. 14.27. This is the simplest form of the general system in Fig. 14.20—it uses a single 2-D kernel and an FCNN with no hidden layers.

a b c
d e f
g h i

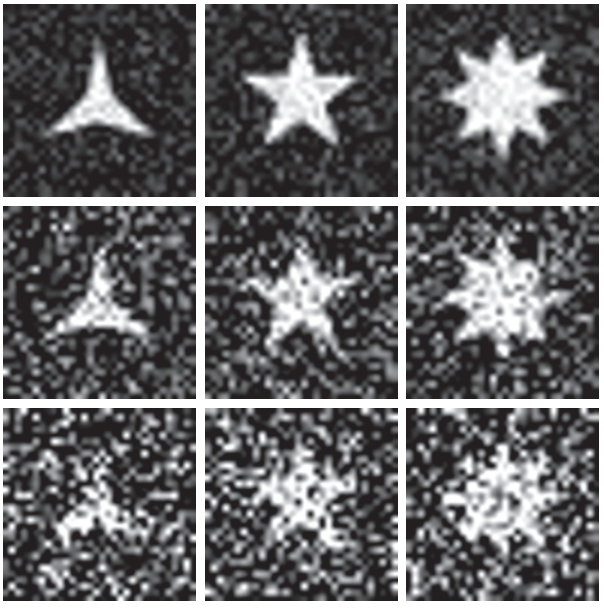


FIGURE 14.27
Row 1: Images corrupted by additive Gaussian noise of zero mean and variance of 0.01 ($\sigma = 0.1$) intensity levels on a [0,1] intensity scale. *Row 2:* Images corrupted by noise with a variance of 0.1 ($\sigma = 0.32$) intensity levels. *Row 3:* Images corrupted by noise with a variance of 0.25 ($\sigma = 0.5$) intensity levels.

```
>> % group will contain NT*size(I0,3) images.

>> % Number of groups of images.
>> NT = 50;
>> IG = [];

>> % Generate the groups of images.
>> for j = 1:NT
       IG = cat(3,IG,I0);
end

>> % Concatenate R NT times to correspond with the number of images in IG.
>> R = [];
>> for j = 1:NT
       R = cat(2,R,R0);
end

>> % Add Gaussian noise to each image in IG. This will be our training set.
>> % Image intensities have to be in the range [0,1].
>> rng('shuffle'); % Random start seed each time.
>> mean = 0;
>> var = 0.25;
>> IGn = imnoise(IG,'gaussian',mean,var);
>> IGn = mat2gray(IGn);

>> % Upsample and show one image from each class.
>> im1 = imresize(IGn(:,:,1),[300 300]);
```

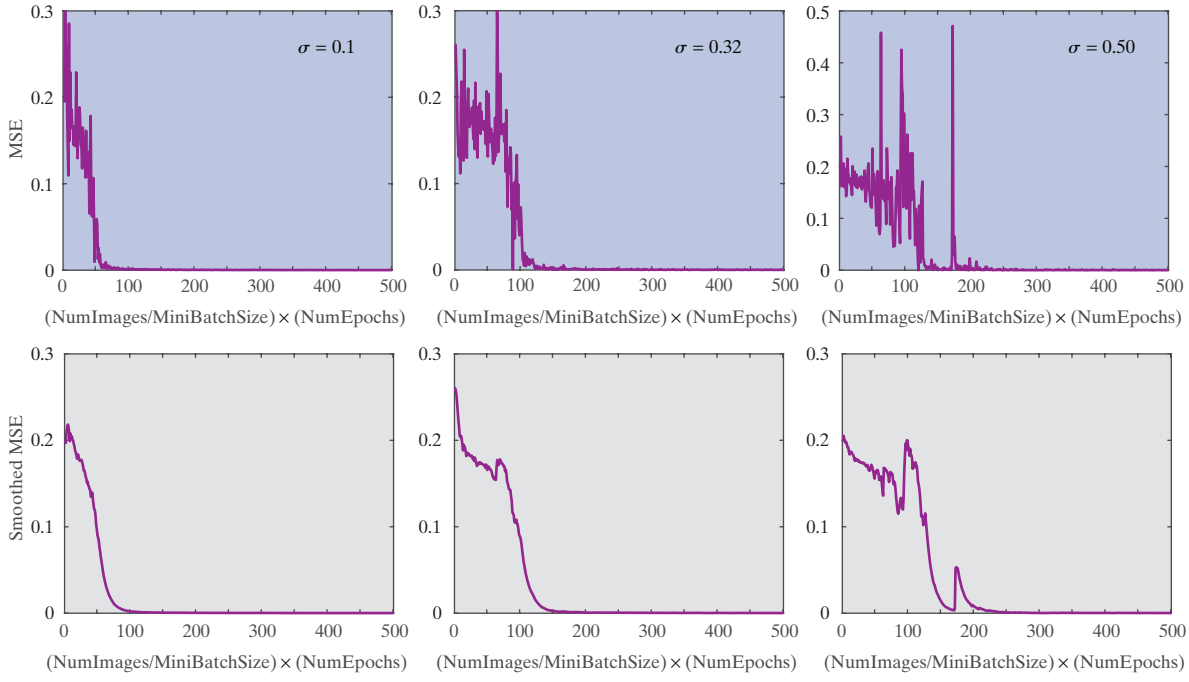


FIGURE 14.28 *Top row:* MSE curves corresponding to the images in Fig. 14.27 for the values of σ shown. *Bottom row:* Smoothed MSE curves. Compare the variability of the curves against the images in Fig. 14.27.

```
>> ylim([0, ceil(max(cnndataout.SmoothMSE)*10)*.1])

%% CLASSIFY.

>> classdataout = cnnclassify(cnndataout.cnn,cnndataout.fcnn,cnndatain.Images,cnndatain.R);
>> disp('Performance (correct classification rate in %) of training images:')
>> disp(classdataout.ClassificationRate)
       100
```

The plots of `MSE` and `SmoothedMSE` in Figs. 14.28(a) and (d) show quick convergence to zero, so we expect the classification results of the training set close 100% accurate. To create a test data set of 900 noisy images, we changed `NT` to 300 and repeated the second block of code. We input this set of images directly into function `cnnclassify`. The result was again 100% accurate, indicating that the system had not overfit in training. *Overfitting* is a condition in which the training set is recognized with high accuracy, but an independent test set of the same type of data is not. The plots in Fig. 14.28 also show that MSE curves can be quite variable, thus highlighting the need for smoothing in order to reveal the most important feature of an error curve—the rate at which it decreases as a function of epoch.

Several things are noteworthy in the preceding experiment. First, you should pay close attention to how the CNN functions were set up and used. We used the default settings for most network parameters,

but actually specified the default values to help you gain familiarity with the notation. The one exception was the size of the kernel. The default setting was erratic and took longer to converge than larger kernels. A kernel of size 9×9 performed satisfactorily, but larger values worked also. Independently of kernel size, the number of epochs required for convergence increased as the level of noise increased. You can see this phenomenon at play in Fig. 14.28.

We conducted similar experiments with images at the other two noise levels. Convergence took longer to achieve, but the classification results of the training and test sets were again 100%. Considering the level of distortion evident in the third row in Fig. 14.27, these are impressive results for the simplest possible CNN/FCNN. It is evident from these results that deep learning is able to extract meaningful features in the presence of high levels of noise. Observing the images in Fig. 14.27 at a distance will demonstrate immediately that the important distinguishing features of each star, although severely distorted, are still present in the noisy data. That is all the system needed to be able to extract those features.

Although the results in this example are indeed impressive, keep in mind that noise was the only source of image variability . The objects did not change position or undergo any other type of transformation found in practice, such as rotation and scale change. The way deep learning handles such variables is by requiring massive databases of training images that contain as many instances of these transformations as possible. As you will see in the following examples, using large data bases can yield accurate results using systems that are not particularly complex.

EXAMPLE 14.13: Using a large database to train a CNN/FCNN to recognize handwritten numerals.

In this example, we look at a more practical application using a database containing 60,000 training and 10,000 test images of handwritten numeric characters. The content of this database, called the *MNIST database*, is similar to a database from NIST (National Institute of Standards and Technology). The former is a "cleaned up" version of the latter, in which the characters have been semi centered and formatted into grayscale images of size 28×28 pixels. Both databases, including instructions and the names of the individuals and organizations responsible for their creation, are available online. The samples shown in Fig. 14.29 are typical of the numeric characters available in the two databases. As you can see, there is significant variability between characters of the same class.

The CNN/FCNN needed to classify the MNIST database with high accuracy is slightly more complex than the system in the previous example. As Fig. 14.30 shows, the system consists of a two-layer CNN followed by a two-layer FCNN that, because it has no hidden layers, is a linear classifier. The input images are grayscale, so the kernels in layer one have a depth of 1. We used 6 features maps in the first layer and twelve in the second. The sizes of the kernel volumes, feature and pooled maps, and the vector input into the FCNN are shown in the figure. The latter has ten output neurons because we are dealing with 10 pattern classes. See Fig. 14.24 for details on how to determine the sizes of the 2-D data and vectors shown in Fig. 14.30.

The MNIST data is in your *Support Package* under the name `mnist_uint8.mat`. To unpack it in a format ready for use with the CNN functions, we use the custom function `getMNISTimages`, whose help section contains detailed descriptions of all parameters (the function is in your Support Package also). In this example, we work with a subset of the database—500 training images from each of the ten classes for

FIGURE 14.29
Samples similar to those available in the NIST and MNIST databases. Each character subimage is of size 28×28 pixels.(Individual images courtesy of NIST.)

a total of 5,000 images, starting with the first image in the dataset. Results with the entire database are discussed at the end of the example and in Project 14.9.

getMNISTimages

```
>> %% LOAD THE TRAINING IMAGES AND CORRESPONDING MEMBERSHIP MATRIX.

>> [I,R] = getMNISTimages(500,'training',1);
>> cnndatain.Images = I;
>> cnndatain.R = R;
```

We specify the parameters of the network in Fig. 14.30 as follows. As is usually the case, we determined the values of these parameters experimentally:

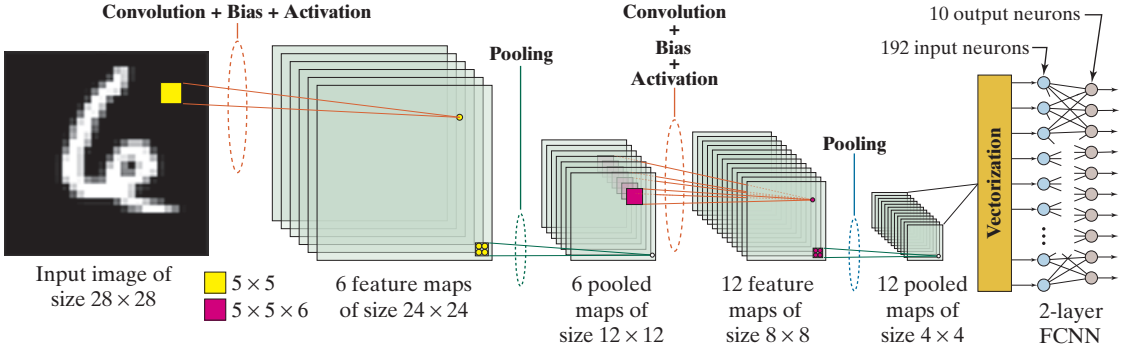


FIGURE 14.30 CNN/FCNN trained to recognize the ten digits in the MNIST database. The system was trained with 60,000 numerical character images and tested with 10,000 images.

EXAMPLE 14.14: Working with a large database of RGB images.

In this example we start with the CNN/FCNN in Fig. 14.30 and adapt it as necessary to be able to train it using all 50,000 training images in the *CIFAR-10 image database*, examples of which are shown in Fig. 14.34. These are RGB images, so we need to modify several parameters from the previous example:

```
>> % Modify the depth of the image size to be able to process RGB images.
>> cnnparam.ImageSize = [32,32,3];
>> % Also, based on several quick trials, we modified the values of the learning
>> % rate constants:
>> cnnparam.Alpha = 1.0;
>> cnnparam.fcnnAlpha = 0.01;

>> % We used the parameters from the previous example, including the changes just
>> % discussed, to initialize the cnn/fcnn:
>> [cnn,fcnn] = cnninit(cnnparam);
```

Next, we extracted all the training images from the CIFAR-10 image database using custom function `getCIFAR10images`, a listing of which is in your Support Package:

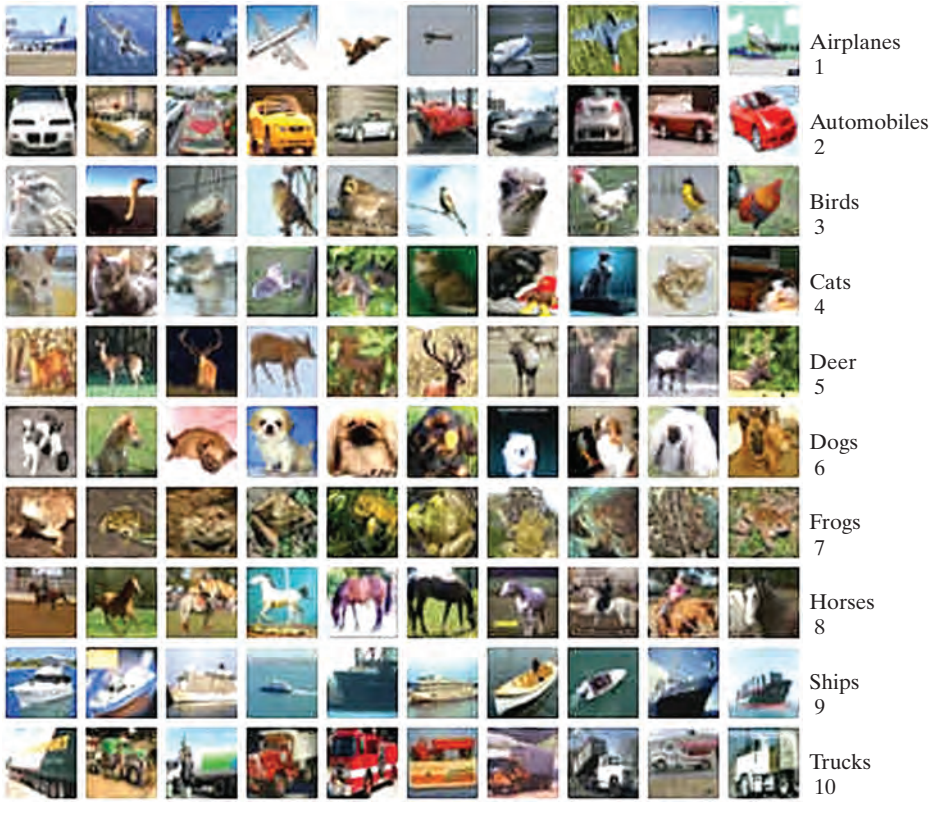


FIGURE 14.34
Samples of mini images from the CIFAR-10 database, which consists of a set of 50,000 training and 10,000 test RGB images of size $32 \times 32 \times 3$. The images are from the ten categories shown in the figure. The classes are mutually exclusive—neither the Automobiles nor Trucks classes include pickup trucks. The numbers on the right are class numbers used to construct the membership matrix. (Credit: Alex Krizhevsky, University of Toronto.)

intervals. Stochastic gradient descent with momentum is another important training refinement. These approaches were partially instrumental in the improvement in classification accuracy in Example 14.15 over the results in Example 14.14.

The topics just discussed are representative of the many issues one encounters when designing and training large-scale, deep neural networks. A paper by LeCun et al. [2012] is an excellent overview of the types of considerations introduced in the preceding discussion. In fact, the scope spanned by these topics is extensive enough to be the subject of an entire book (see Montavon et al. [2012]). The neural net architectures we discussed were by necessity limited. You can get a good idea of the requirements of implementing practical networks by reading a paper by Krizhevsky, Sutskever, and Hinton [2012], which summarizes the design and implementation of a large-scale, deep convolutional neural network. There are a multitude of designs that have been implemented over the past decade. A quick internet search will list a large number of experiments with different architectures. The book by Sejnowski [2018] is also of interest.

Summary

The material presented in the previous sections of this chapter cover the spectrum of methods used today for image pattern classification. The classical techniques presented early in the chapter are used when knowledge about an application allows features to be defined or engineered with enough precision to be truly representative of the objects to be classified. The key MATLAB concept underlying minimum-distance and Bayes classifiers is the vectorization of distance computations we developed early in the chapter. The concept of pattern classification using a single perceptron is of little use today, other than as an important historical footnote. However, the interconnection of perceptron-like units, which we referred to as artificial neurons, is the foundation of deep neural networks, and the current importance of this topic is without question. The emphasis of our approach in the last two sections was to give a solid foundation of the equations that govern the behavior of both fully-connected and convolutional neural networks, with an emphasis on the concept of backpropagation. We also showed in detail the fundamentals of how to approach MATLAB programming of deep neural networks at a basic level by developing all the functions necessary for learning and classification. As you surely noticed, convolutional neural nets are computationally intensive, requiring concepts of parallelization that are beyond the scope of our discussion. Ultimately, the design of complex deep neural networks capable of working with large data sets is an evolving experimental "art."

MATLAB Projects

*Solutions to the projects marked with an asterisk * are in the DIPUM3E Student Support Package (consult the book web site). All your code must be documented so that typing* `help` *at the prompt, followed by the script or function name, gives enough detail for a user to be able to run it. Test the functionality of all your code thoroughly.*

14.1 Minimum-distance classification.

(a)* The Mahalanobis distance defined in Eq. (14-5) can be used for minimum-distance classification. Explain how you would use custom function `mahalanobis` to write a minimum-distance classifier instead of the method we used in function `minDistanceClassifier`. You do not have to implement a function, but be specific as to how you would do it.