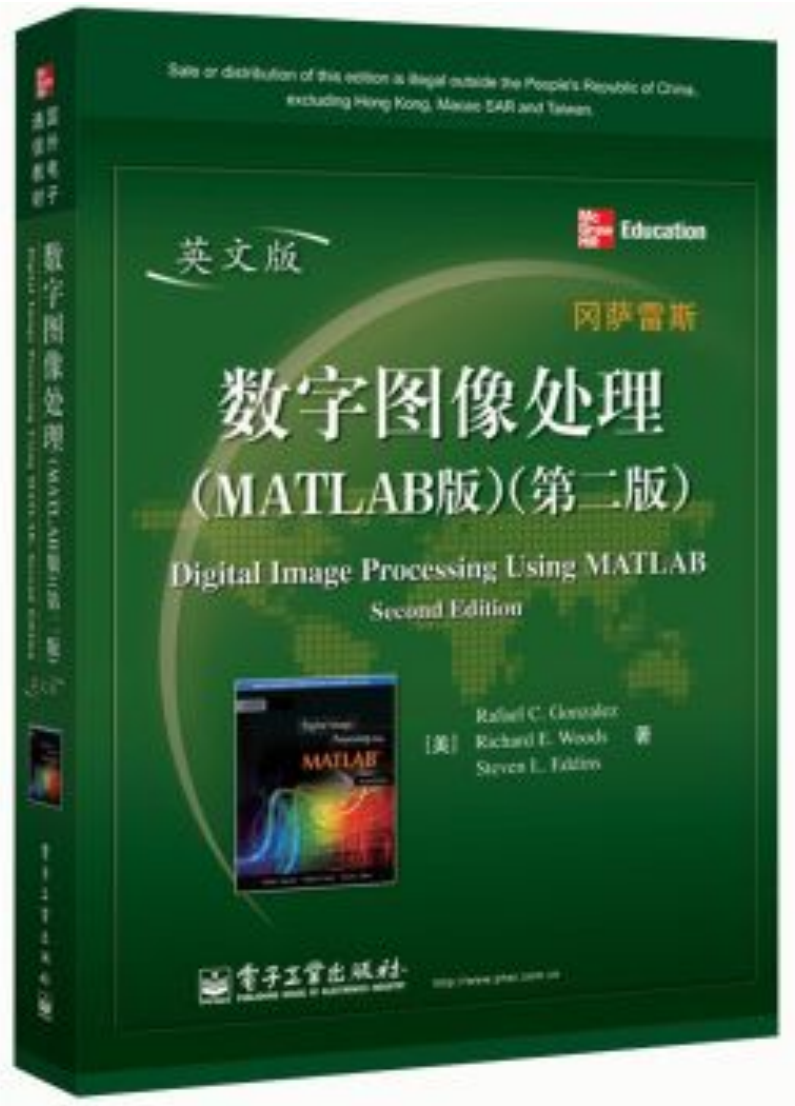


数字图像处理（MATLAB 版）（英文版）（第 2 版）

[Digital Image Processing Using MATLAB（Second Edition）]

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

电子工业出版社

2016

山东大学 信息科学与工程学院 江铭炎 教授

Preface[1]

Solutions to problems in the field of digital image processing require extensive experimental work involving software simulation and testing with large sets of sample images. Algorithm development is based on theory, the actual implementation of these algorithms requires parameter estimation and, frequently, algorithm revision and comparison of candidate solutions. Thus, selection of a flexible, comprehensive, and well-documented software development environment is a key factor that has important implications in the cost, development time, and portability of image processing solutions.

Little has been written on this aspect in the form of textbook material dealing with both theoretical principles and software implementation of digital image processing concepts. This book is written for just this purpose. Its main objective is to provide a foundation for implementing image processing algorithms using modern software tools. A complementary objective is to prepare a book with a basic background in digital image processing, mathematical analysis, and computer programming. Basic knowledge of MATLAB also is desirable.

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

To meet the first objective, most of the theoretical concepts in the following chapters were selected from *Digital Image Processing* by Gonzalez and Woods, which textbook used by educators all over the world for over two decades. The software tools selected are from the MATLAB Image Processing Toolbox (IPT), in both education and industrial applications. A basic integration of well-established theoretical concepts and their implementation using state-of-the-art software tools.

The book is organized along the same lines as *Digital Image Processing.* Easy

---

[1] Book: Image processing with Matlab Tutor : Prof. Mingyan Jiang (江铭炎)

Address: Shandong University, Information school building, 2 floor (room 203)

Email: jiangmingyan@sdu.edu.cn

网页: 信息学院=>精品课程=>数字图像处理

access to a more detailed treatment of all the image processing concepts discussed here, Possible to present theoretical material in a simple manner. To maintain a focus on the software implementation aspects of image processing problem solutions. The Image Processing Toolbox offers some significant advantages, not only in the breadth of its computational tools, but also because it is supported under most operating systems in use today. This book, its emphasis on showing how to develop new code to enhance existing MATLAB and IPT functionality. This is an important feature in image processing, is characterized by the need for extensive algorithm development and experimental work.

With an introduction to the fundamentals of MATLAB functions and programming, the book proceeds to address the mainstream areas of image processing. The major areas covered include: intensity transformations, linear and nonlinear spatial filtering, filtering in the frequency domain, image restoration and registration, color image processing. The major areas covered include: wavelets, image data compression, morphological image processing, image segmentation, region and boundary representation and description, and object recognition. This material is complemented by numerous illustrations of how to solve image processing problems using MATLAB and IPT functions. If unction did not exist, a new function is written and documented as part of the instructional focus of the book. Over 60 new functions are included in the following chapters. These functions increase the scope of IPT by approximately 35 percent for new image processing software solutions.

The material is presented in textbook format, not as a software manual. Although the book is self-contained, we have established a companion Web site (see Section 1.5) designed to provide support in a number of areas. For students following a course study or individuals self study, the site contains tutorials and reviews on background material, as well as projects and image databases, including all images in the book. For instructors, the site contains classroom presentation materials images and graphics used in the book. The site with image processing and IPT fundamentals is a useful place for up-to-date references, new implementation techniques, and materials. Download executable files of all the new functions developed in the text.

Significant effort to the selection of material is fundamental, whose value is likely to remain applicable in a rapidly evolving body of knowledge.

Readers of the book will benefit from this effort and thus find the material timely and useful in their work.

Chapter1 Introduction

Preview

Digital image processing is an area characterized by the need for extensive experimental work to establish the viability of proposed solutions to a given problem.

A theoretical base and state-of-the-art software can be integrated into a prototyping environment.

Objective is to provide a set of well-supported tools for the solution of a broad class of problems in digital image processing.

1.1 Background

An important characteristic underlying the design of image processing systems is the significant level of testing and experimentation before arriving at an acceptable solution.

This characteristic implies that the ability to formulate approaches and quickly prototype candidate solutions plays a major role in reducing the cost and time required.

Little has been written in the way of instructional material to bridge the gap between theory and application in a well-supported software environment.

The objective of this book is to integrate under theoretical concepts with the knowledge required to implement those concepts using state-of-the-art image processing software tools.

The theoretical basis of the material in the following chapters are mainly from the leading textbook in the field: Digital Image Processing, by Gonzalez and Woods, published by Prentice Hall.

The software code and supporting tools are based on the leading software package in the field: The MATLAB Image Processing Toolbox.

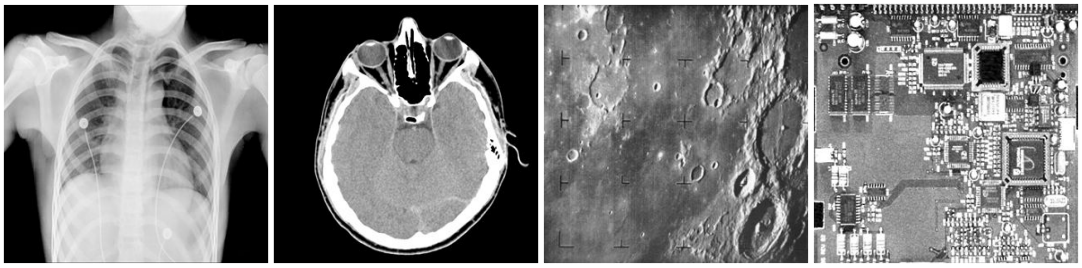
To the book, the reader should have introductory preparation in digital image processing, either by having taken a formal course of study on the subject at the senior or first-year graduate level, or by acquiring the necessary background in a program of self-study.

The reader has some familiarity with MATLAB, as well as rudimentary knowledge of the basics of computer programming in a technically oriented language.

Because MATLAB is an array-oriented language, basic knowledge of matrix analysis also is helpful.

The book is organized and presented in a textbook format, 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.



1- 1

This approach allows orderly progression from simple concepts to sophisticated implementation of image processing algorithms.

1- 2

Readers already familiar with MATLAB, IPT, and image processing fundamentals can proceed directly to specific applications of interest, the functions in the book can be used as an extension of the family of IPT functions.

Over 60 new functions are developed in the chapters, extend by 35% the set of about 175 functions in IPT.

In addition to addressing specific applications, the new functions are clear examples of how to combine existing MATLAB and IPT functions with new code to develop prototypic solutions to a broad spectrum of problems in digital image

processing.

The toolbox functions, as well as the functions developed in the book, run under most operating systems. Consult the book Web site (see Section 1.5) for a complete list.

1.2 What Is Digital Image Processing?

An image may be defined as a two-dimensional function, $f(x, y)$,where x and y are *spatial coordinates*, and the amplitude f of 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.

1- 3

Note that a digital image is composed of a finite number of elements, each of which has a particular location and value. These elements are referred to as *picture elements, image elements, pels,* and *pixels*. *Pixel* is the term most widely used to denote the elements of a digital image. We consider these definitions formally in Chapter 2.

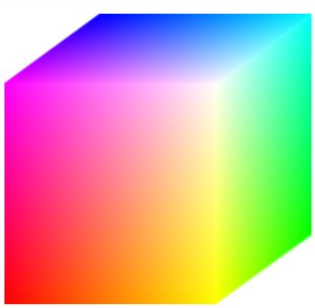
Vision is the most advanced of our senses, so it is not surprising that images play the single most important role in human perception. *Imaging machines* cover almost the entire electromagnetic (EM) spectrum, ranging from gamma to radio waves. These include ultrasound, electron microscopy, and computer-generated images. Thus, digital image processing encompasses a wide and varied field of applications.

There is no general agreement among authors regarding where image processing stops and other related areas, such as image analysis and computer vision, starts. We

believe this to be a limiting and somewhat artificial boundary. For example, under this definition, even the trivial task of computing the average intensity of an image would not be considered an image processing operation. On the other hand, there are fields such as computer vision whose ultimate goal is to use computers to emulate human vision, including learning and being able to make inferences and take actions based on visual inputs. This area itself is a branch of artificial intelligence (AI), whose objective is to emulate human intelligence. The area of image analysis (also called image understanding) is in between image processing and computer vision.

There are no clear-cut boundaries in the continuum from image processing at one end to computer vision at the other. However, one useful paradigm is to consider three types of computerized processes in this continuum: low-, mid-, and high-level processes.





1- 4

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 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 that its inputs generally are images, but its outputs are attributes extracted from those images (e.g., edges, contours, and the identity of individual objects). Finally, *higher-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. In this book *digital image processing* encompasses processes whose inputs and outputs are images, in addition, encompasses processes that extract attributes from images, up to and including the recognition of individual objects. As a simple illustration to clarify these concepts, consider the area of automated analysis of text. The processes of acquiring an image of the area containing the text, preprocessing that image, extracting (segmenting) the individual characters, describing the characters in a form suitable for computer processing, and recognizing those individual characters, are in the scope of what we call digital image processing in this book. Making sense of the content of the page may be viewed as being in the domain of image analysis and even computer vision. Digital image processing, as we have defined it, is used successfully in a broad range of areas of exceptional social and economic value.

1.3 Background on MATLAB and the Image Processing Toolbox

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

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

MATLAB is an interactive system whose basic data element is an array that does not require dimensioning. This allows formulating solutions to many technical computing problems, especially those involving matrix representations, in a fraction of the time it would take to write a program in a scalar non-interactive language such as C or Fortran.

The name MATLAB stands for *matrix laboratory*. MATLAB was written originally to provide easy access to matrix software developed by the LINPACK (Linear System Package) and EISPACK (Eigen System Package) projects. Today,

MATLAB engines incorporate the LAPACK (Linear Algebra Package) and BLAS (Basic Linear Algebra Subprograms) libraries, constituting the state of the art in software for matrix computation.

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

The MATLAB Student Version includes a full-featured version of MATLAB. The Student Version can be purchased at significant discounts at university bookstores and at the MathWorks' Web site (www.mathworks.com). Student versions of add-on products, including the Image Processing Toolbox, also are available.

1.4 Areas of Image Processing Covered in the Book

Every chapter contains the MATLAB and IPT material needed to implement the image processing methods discussed. When a MATLAB or IPT function does not exist to implement a specific method, a new function is developed and documented. a complete listing of every new function is included in the book. The remaining eleven chapters cover material in the following areas.

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

Chapter 3: Intensity Transformations and Spatial Filtering. This chapter covers in detail how to use MATLAB and IPT to implement intensity transformation functions. Linear and nonlinear spatial filters are covered and illustrated in detail.

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

domain filters from specified spatial filters.

Chapter 5: Image Restoration. Traditional linear restoration methods, such as the Wiener filter, are covered in this chapter. Iterative, nonlinear methods, such as the Richardson-Lucy method and maximum-likelihood estimation for blind deconvolution, are discussed and illustrated. Geometric corrections and image registration also are covered.

Chapter 6: Color Image Processing. This chapter deals with pseudocolor and full-color image processing. Color models applicable to digital image processing are discussed, and IPT functionality in color processing is extended via implementation of additional color models. The chapter also covers applications of color to edge detection and region segmentation.

Chapter 7:Wavelets. In its current form, IPT does not have any wavelet transforms. A set of wavelet-related functions compatible with the Wavelet Toolbox is developed in this chapter that will allow the reader to implement all the wavelet-transform concepts discussed in the Gonzalez-Woods book.

Chapter 8: Image Compression. The toolbox does not have any data compression functions. In this chapter, we develop a set of functions that can be used for this purpose.

Chapter 9: Morphological Image Processing. The broad spectrum of functions available in IPT for morphological image processing are explained and illustrated in this chapter using both binary and gray-scale images.

Chapter 10: Image Segmentation. The set of IPT functions available for image segmentation are explained and illustrated in this chapter. New functions for Hough transform processing and region growing also are developed.

Chapter 11: Representation and Description. Several new functions for object representation and description, including chain-code and polygonal representations, are developed in this chapter. New functions are included also for object description, including Fourier descriptors, texture, and moment invariants. These functions complement an extensive set of region property functions available in IPT.

Chapter 12: Object Recognition. One of the important features of this chapter is the efficient implementation of functions for computing the Euclidean and Mahalanobis distances. These functions play a central role in pattern matching. The chapter also contains a comprehensive discussion on how to manipulate strings of symbols in MATLAB. String manipulation and matching are important in structural

pattern recognition.

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

Appendix A: Contains a summary of all IPT and new image-processing functions developed in the book. Relevant MATLAB function also are included. This is a useful reference that provides a global overview of all functions in the toolbox and the book.

Appendix B: Contains a discussion on how to implement graphical user interfaces (GUIs) in MATLAB. GUIs are a useful complement to the material in the book because they simplify and make more intuitive the control of interactive functions.

Appendix B: Contains a discussion on how to implement graphical user interfaces (GUIs) in MATLAB. GUIs are a useful complement to the material in the book because they simplify and make more intuitive the control of interactive functions.

Appendix C: New function listings are included in the body of a chapter when a new concept is explained. Otherwise the listing is included in Appendix C. This is true also for listings of functions that are lengthy. Deferring the listing of some functions to this appendix was done primarily to avoid breaking the flow of explanations in text material.

1.5 The Book Web Site

An important feature of this book is the support contained in the book Web site. The site address is www.prenhall.com/gonzalezwoodseddins.

This site provides support to the book in the following areas: Downloadable M-files, including all M-files in the book; Tutorials; Projects; Teaching materials; Links to databases, including all images in the book; Book updates ; Background publications.

The site is integrated with the Web site of the Gonzalez-Woods book: www.prenhall.com/gonzalezwoods

which offers additional support on instructional and research topics.

www.imageprocessingplace.com

1.6 Notation

Equations in the book are typeset using familiar italic and Greek symbols, as in *f(x,y)*= and *Ø(u,v)*= .All MATLAB function names and symbols are typeset in monospace font, as in *fft2(f)*, *logical(A)*, and *roipoly(f, c, r)*.

The first occurrence of a MATLAB or IPT function is highlighted by use of the following icon on the page margin: function name.

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

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

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

1.7 The MATLAB Working Environment

In this section we give a brief overview of some important operational aspects of using MATLAB.

1.7.1 The MATLAB Desktop

The MATLAB *desktop* is the main MATLAB application window. As Fig. 1.1 shows, the desktop contains five subwindows: the Command Window, the Workspace Browser, the Current Directory Window, the Command History Window, and one or more Figure Windows, which are shown only when the user displays a graphic.

The *Command Window* is where the user types MATLAB commands and expressions at the prompt (>>) and where the outputs of those commands are displayed. MATLAB defines the *workspace* as the set of variables that the user creates in a work session. The *Workspace Browser* shows these variables and some information about them. Double-clicking on a variable in the Workspace Browser launches the *Array Editor*, which can be used to obtain information and in some instances edit certain properties of the variable.

The Current Directory tab above the Workspace tab shows the contents of the *current directory*, whose *path* is shown in the *Current Directory Window*. For example, in the Windows operating system the path might be as follows: C:\MATLAB\Work,

indicating that directory “Work” is a subdirectory of the main directory “MATLAB,” which is installed in drive C. Clicking on the arrow in the Current Directory Window shows a list of recently used paths. Clicking on the button to the right of the window allows the user to change the current directory.

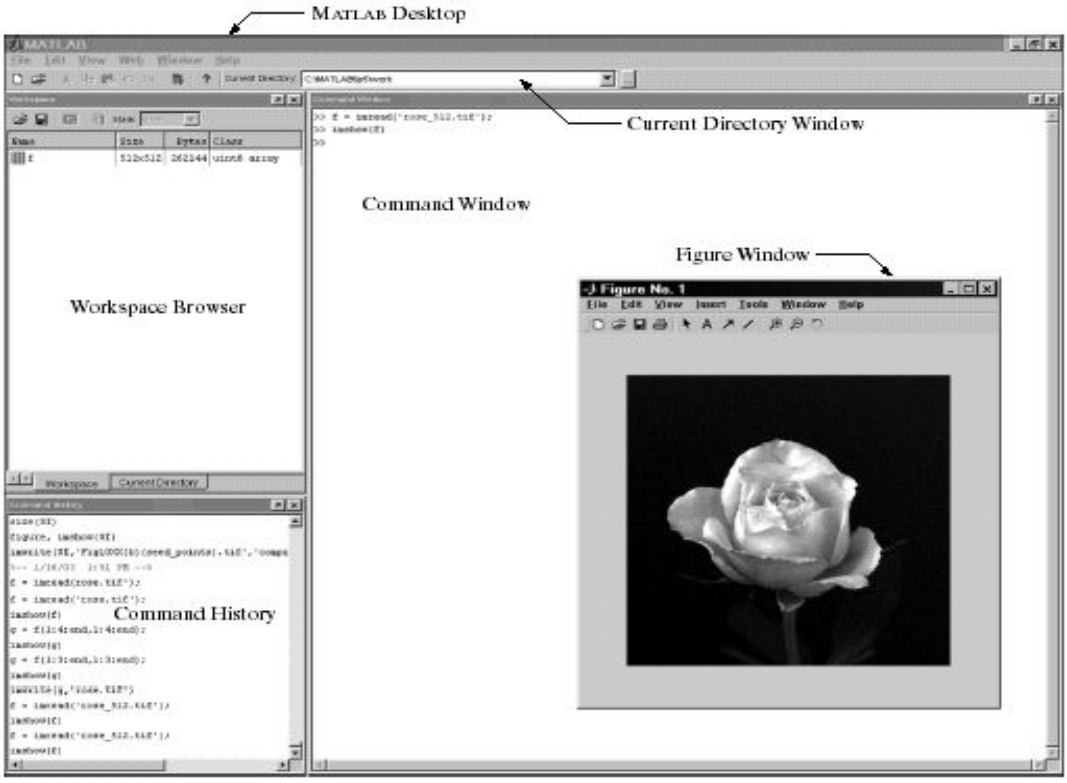


FIGURE 1.1 The MATLAB desktop and its principal components.

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

The *Command History Window* contains a record of the commands a user has entered in the Command Window, including both current and previous MATLAB sessions. Previously entered MATLAB commands can be selected and re-executed from the Command History Window by right-clicking on a command or sequence of

commands. This action launches a menu from which to select various options in addition to executing the commands. This is a useful feature when experimenting with various commands in a work session.

1.7.2 Using the MATLAB Editor to Create M-files

The MATLAB *editor* is both a text editor specialized for creating M-files and a graphical MATLAB debugger. The editor can appear in a window by itself, or it can be a subwindow in the desktop. M-files are denoted by the extension .m, as in `pixeldup.m`. The MATLAB editor window has numerous pull-down menus for tasks such as saving, viewing, and debugging files. Because it performs some simple checks and also uses color to differentiate between various elements of code, this text editor is recommended as the tool of choice for writing and editing M-functions. To open the editor, type edit at the prompt in the Command Window. Similarly, typing edit filename at the prompt opens the M-file filename. m in an editor window, ready for editing. As noted earlier, the file must be in the current directory, or in a directory in the search path.

1.7.3 Getting Help

The principal way to get help online is to use the MATLAB *Help Browser*, opened as a separate window either by clicking on the question mark symbol(?) on the desktop toolbar, or by typing help-browser at the prompt in the Command Window. The Help Browser is a Web browser integrated into the MATLAB desktop that displays Hypertext Markup Language (HTML) documents. The Help Browser consists of two panes, the *help navigator pane*, used to find information, and the *display pane*, used to view the information.

Self-explanatory tabs on the navigator pane are used to perform a search. For example, help on a specific function is obtained by selecting the **Search** tab, selecting **Function Name** as the **Search Type**, and then typing in the function name in the **Search for** field.

1.7.4 Saving and Retrieving a Work Session

There are several ways to save and load an entire work session (the contents of

the Workspace Browser) or selected workspace variables in MATLAB. The simplest is as follows.

To save the entire workspace, simply right-click on any blank space in the Workspace Browser window and select **Save Workspace As** from the menu that appears.

To load saved workspaces and/or variables, left-click on the folder icon on the toolbar of the Workspace Browser window.

It is possible to achieve the same results described in the preceding paragraphs by typing save and load at the prompt, with the appropriate file names and path information.

1.8 How References Are Organized in the Book

All references in the book are listed in the Bibliography by author and date, as in Soille [2003]. Most of the background references for the theoretical content of the book are from Gonzalez and Woods [2002]. References that are applicable to all chapters, such as MATLAB manuals and other general MATLAB references, are so identified in the Bibliography.

Summary

In addition to a brief introduction to notation and basic MATLAB tools, the material in this chapter emphasizes the importance of a comprehensive prototyping environment in the solution of digital image processing problems.

In the following chapter we begin to lay the foundation needed to understand IPT functions and introduce a set of fundamental programming concepts that are used throughout the book. The material in Chapters 3 through 12 spans a wide cross section of topics that are in the mainstream of digital image processing applications. The discussion in those chapters follows the same basic theme of demonstrating how combining MATLAB and IPT functions with new code can be used to solve a broad spectrum of image-processing problems.

Chapter2 Fundamentals

Preview

1. Know Matlab tool power
2. Matlab basic content IPT

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.

$$\text{Image } A = \begin{bmatrix} f(1,1) & f(1,2) \ldots & f(1,N) \\ f(2,1) & f(2,2) \ldots & f(2,N) \\ \ldots & \ldots & \ldots \\ f(M,1) & f(M,2) \ldots & f(M,N) \end{bmatrix}$$

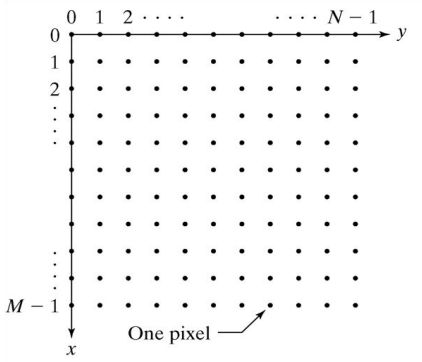
Gray image: The term gray level is used often to refer to the intensity of monochrome images.

Color image: Color images are color system, a color image consists of three (red, green, and blue) individual component images. Monochrome image techniques can be extended to color images.

2- 1

Continuous image can be digitized --- sampling, quantization, digital image.

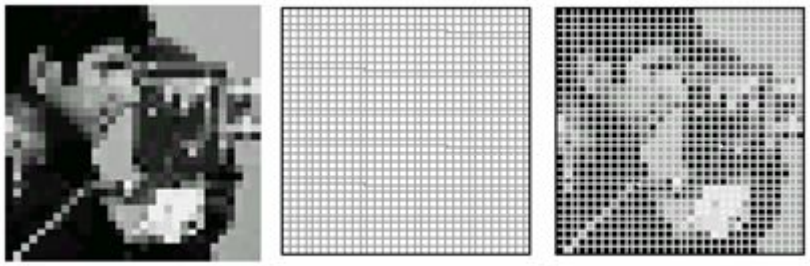
2.1.1Coordinate Conventions



a b

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

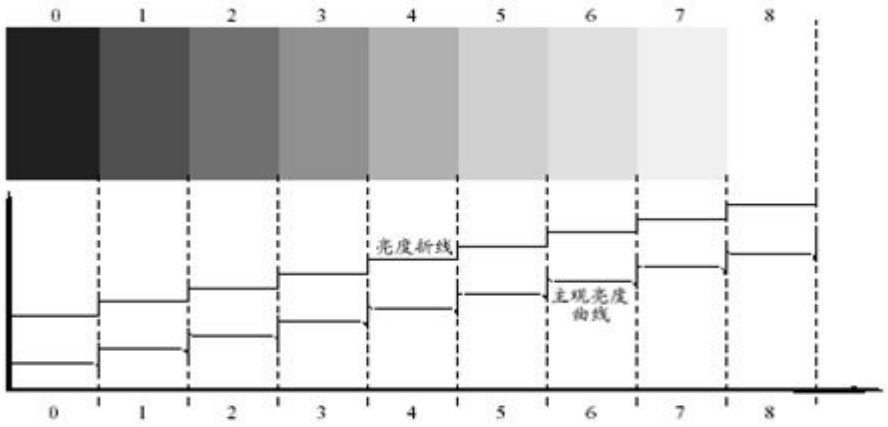
2- 2



2- 3

2.1.2 Images as Matrices

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



2- 4

2.2 Reading Images

Matlab function: imread (‘ *filename* ’) f = imread(‘d:\myimage\chestxray.jpg)

Function: Size(f) [M,N]=size(f)

Format Name	Description	Recognized Extensions
TIFF	Tagged Image File Format	.tif, .tiff
JPEG	Joint Photographic Experts Group	.jpg, .jpeg
GIF	Graphics Interchange Format†	.gif
BMP	Windows Bitmap	.bmp
PNG	Portable Network Graphics	.png
XWD	X Window Dump	.xwd

† GIF is supported by imread, but not by imwrite.

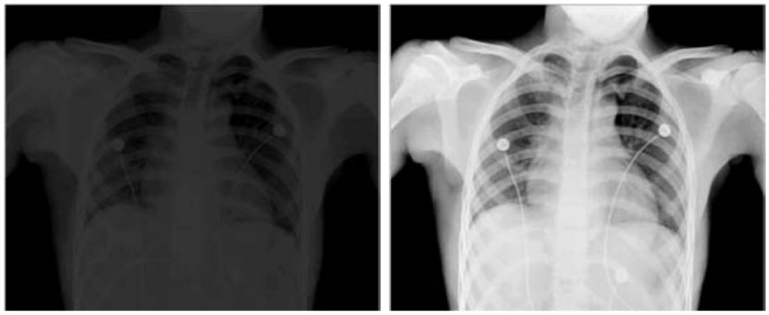
TABLE 2.1 Some of the image/graphics formats supported by imread and imwrite, starting with MATLAB 6.5. Earlier versions support a subset of these formats. See online help for a complete list of supported formats.

2.3 Displaying Images

Matlab function: imshow(f,G)

f is an image array, G is the number of intensity levels used to display it, it defaults to 256 levels. imshow(f, [low high])

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



a b

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

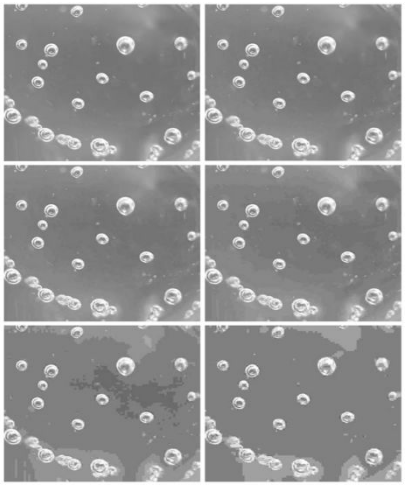
2- 7

2.4 Writing Images

Matlab function: imwrite(f, ‘filename’)

imwrite(f, ‘patient10_run1.tif’)

imwrite(f, ‘filename.jpg’, ‘quality’, q)



a b
c d
e f

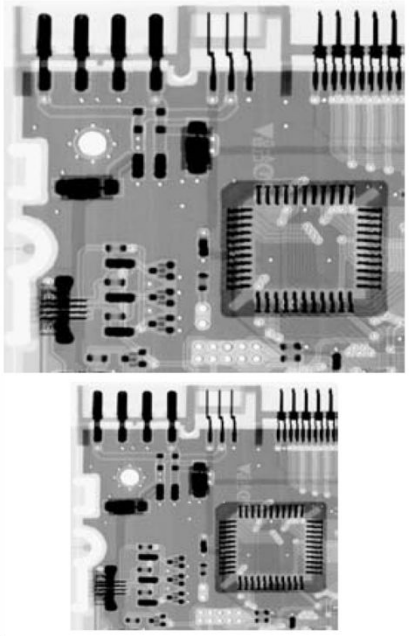
FIGURE 2.4
(a) Original image. (b) through (f) Results of using `jpg` quality values `q = 50, 25, 15, 5,` and `0`, respectively. False contouring begins to be barely noticeable for `q = 15` [image (d)] but is quite visible for `q = 5` and `q = 0`.

2- 8

Function: imfinfo filename

imfinfo lists the image information

name, date, size, format, width, height, bitdepth, color type etc.



a
b

FIGURE 2.5 Effects of changing the dpi resolution while keeping the number of pixels constant. (a) A 450 × 450 image at 200 dpi (size = 2.25 × 2.25 inches). (b) The same 450 × 450 image, but at 300 dpi (size = 1.5 × 1.5 inches). (Original image courtesy of Lixi, Inc.)

2.5 Data Classes

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

TABLE 2.2 Data classes. The first eight entries are referred to as *numeric* classes; the ninth entry is the *character* class, and the last entry is of class *logical*.

2.6 Image Types

- Intensity images
- Binary images
- Indexed images

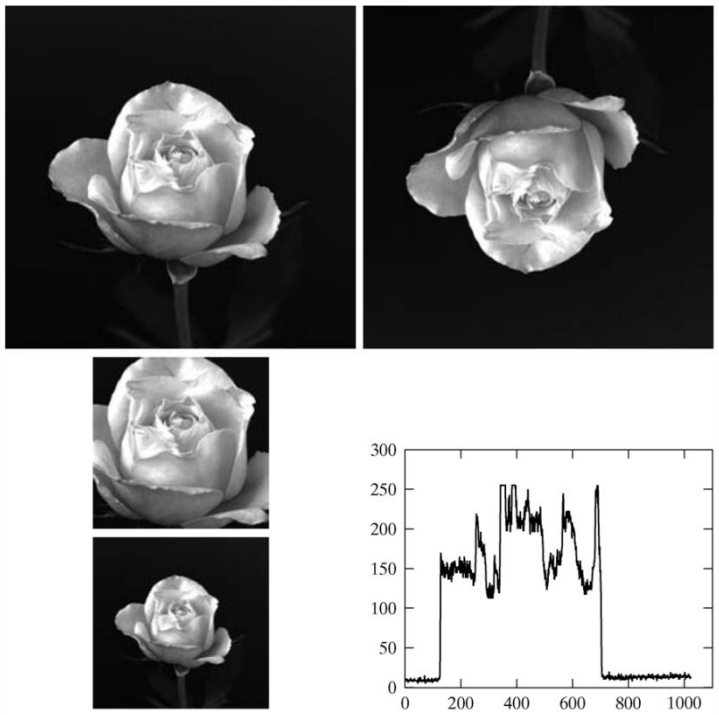
- RGB images

2.7 Converting between Data Classes and Image Types

TABLE 2.3
Functions in IPT for converting between image classes and types. See Table 6.3 for conversions that apply specifically to color images.

Name	Converts Input to:	Valid Input Image Data Classes
im2uint8	uint8	logical, uint8, uint16, and double
im2uint16	uint16	logical, uint8, uint16, and double
mat2gray	double (in range [0, 1])	double
im2double	double	logical, uint8, uint16, and double
im2bw	logical	uint8, uint16, and double

2.8 Array Indexing



a b
c
d e

FIGURE 2.6
Results obtained using array indexing. (a) Original image. (b) Image flipped vertically. (c) Cropped image. (d) Subsampled image. (e) A horizontal scan line through the middle of the image in (a).

2.9 Introduction to M-Function Programming

Operator	Name	MATLAB Function	Comments and Examples
+	Array and matrix addition	plus(A, B)	a + b, A + B, or a + A.
–	Array and matrix subtraction	minus(A, B)	a – b, A – B, A – a, or a – A.
.*	Array multiplication	times(A, B)	C = A.*B, C(I, J) = A(I, J)*B(I, J).
*	Matrix multiplication	mtimes(A, B)	A*B, standard matrix multiplication, or a*A, multiplication of a scalar times all elements of A.
./	Array right division	rdivide(A, B)	C = A./B, C(I, J) = A(I, J)/B(I, J).
.\	Array left division	ldivide(A, B)	C = A.\B, C(I, J) = B(I, J)/A(I, J).
/	Matrix right division	mrdivide(A, B)	A/B is roughly the same as A*inv(B), depending on computational accuracy.
\	Matrix left division	mldivide(A, B)	A\B is roughly the same as inv(A)*B, depending on computational accuracy.
.^	Array power	power(A, B)	If C = A.^B, then C(I, J) = A(I, J)^B(I, J).
^	Matrix power	mpower(A, B)	See online help for a discussion of this operator.
.'	Vector and matrix transpose	transpose(A)	A.'. Standard vector and matrix transpose.
'	Vector and matrix complex conjugate transpose	ctranspose(A)	A'. Standard vector and matrix conjugate transpose. When A is real A.' = A'.
+	Unary plus	uplus (A)	+A is the same as 0 + A.
–	Unary minus	uminus (A)	–A is the same as 0 – A or –1*A.
:	Colon		Discussed in Section 2.8.

TABLE 2.4
Array and matrix arithmetic operators. Computations involving these operators can be implemented using the operators themselves, as in A + B, or using the MATLAB functions shown, as in plus (A, B). The examples shown for arrays use matrices to simplify the notation, but they are easily extendable to higher dimensions.

Function	Description
`imadd`	Adds two images; or adds a constant to an image.
`imsubtract`	Subtracts two images; or subtracts a constant from an image.
`immultiply`	Multiplies two images, where the multiplication is carried out between pairs of corresponding image elements; or multiplies a constant times an image.
`imdivide`	Divides two images, where the division is carried out between pairs of corresponding image elements; or divides an image by a constant.
`imabsdiff`	Computes the absolute difference between two images.
`imcomplement`	Complements an image. See Section 3.2.1.
`imlincomb`	Computes a linear combination of two or more images. See Section 5.3.1 for an example.

2- 14

TABLE 2.5The image arithmetic functions supported by IPT.

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

TABLE 2.6 Relational operators.

2- 15

Operator	Name
&	AND
|	OR
~	NOT

TABLE 2.7 Logical operators.

2- 16

Local Operates and Function

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

TABLE 2.8
Logical functions.

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

TABLE 2.9
Some functions that return a logical 1 or a logical 0 depending on whether the value or condition in their arguments are `true` or `false`. See online help for a complete list.

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

TABLE 2.10 Some important variables and constants.

2- 19

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

TABLE 2.11 Flow control statements.

2- 20

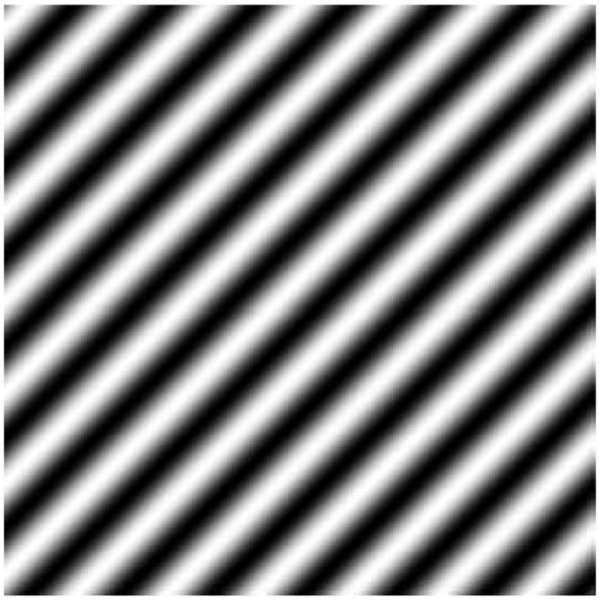


FIGURE 2.7 Sinusoidal image generated in Example 2.13.

2- 21

Summary

- Image representation
- Some functions for image processing
- Image types
- Matlab operations in image

Chapter3 Intensity Transformations and Spatial Filtering

Preview

The term spatial domain refers to the image plane, Methods in this category are based on direct manipulation of pixels in an image. Two important categories of spatial domain processing: *intensity (or gray-level) transformations* and *spatial filtering(neighborhood processing, or spatial convolution)*.

We will develop and illustrate MATLAB formulations representative of processing techniques in these two categories.

In order to carry a consistent theme, most of the examples are related to image enhancement. This is a good way to introduce spatial processing because enhancement is highly intuitive, especially to beginners in the field. These techniques are general in scope and have uses in numerous other branches of digital image processing.

3.1 Background

Spatial domain techniques operate directly on the pixels of an image. The spatial domain processes are denoted by the expression:

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

where f(x,y) is the input image, g(x,y) is the output (processed) image, *T* is an operator on f ,defined over a specified neighborhood about point(x,y). *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 is to use a square or rectangular region centered at as Fig. 3.1 shows.

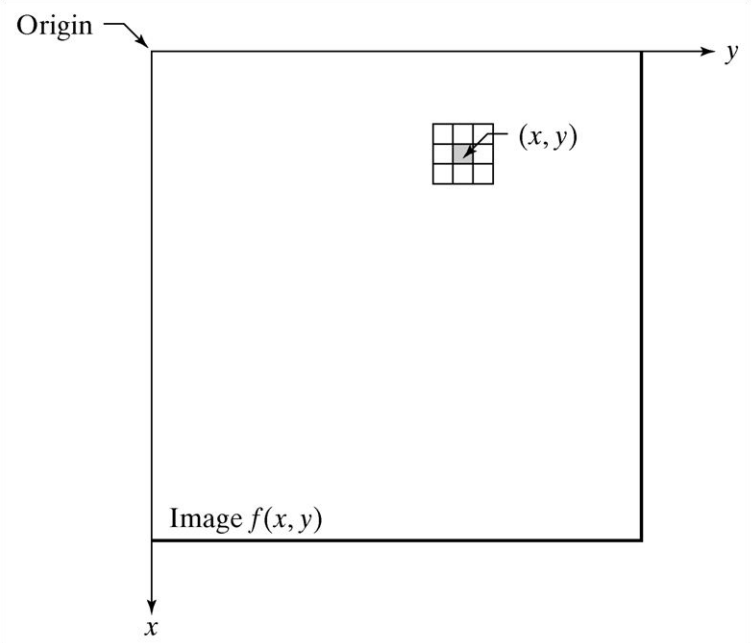
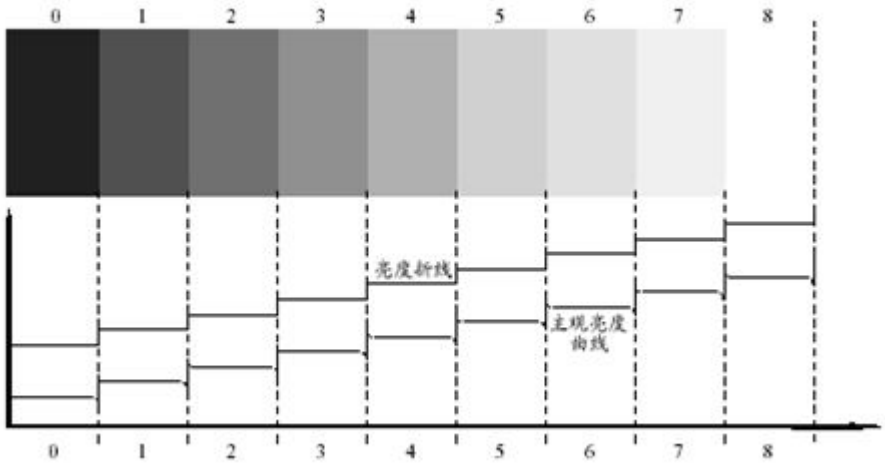


FIGURE 3.1 A neighborhood of size 3 × 3 about a point (x, y) in an image.

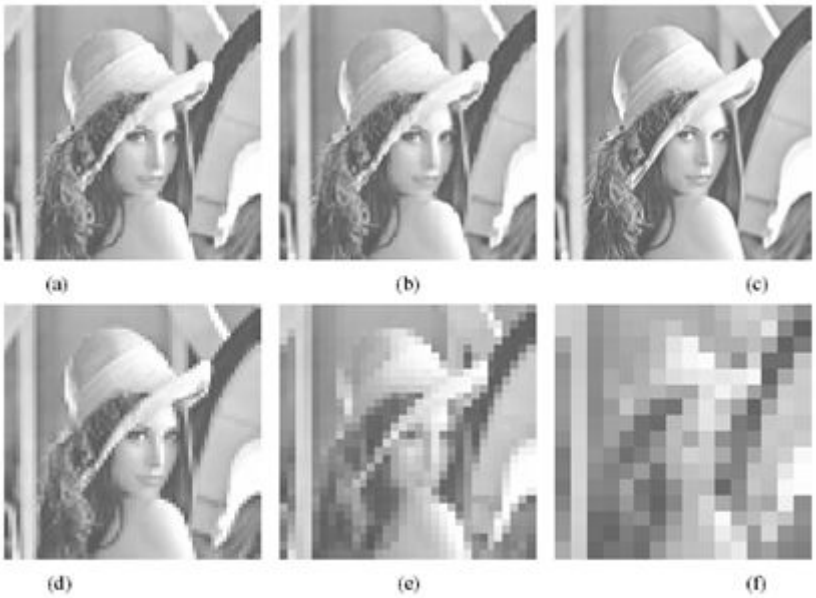
3- 1

Operator T is applied at each location (x,y) to yield the output, g, at that location. Only the pixels in the neighborhood are used in computing the value of g at (x,y) .

Its computational implementation in MATLAB requires that careful attention be paid to data classes and value ranges.



3- 2

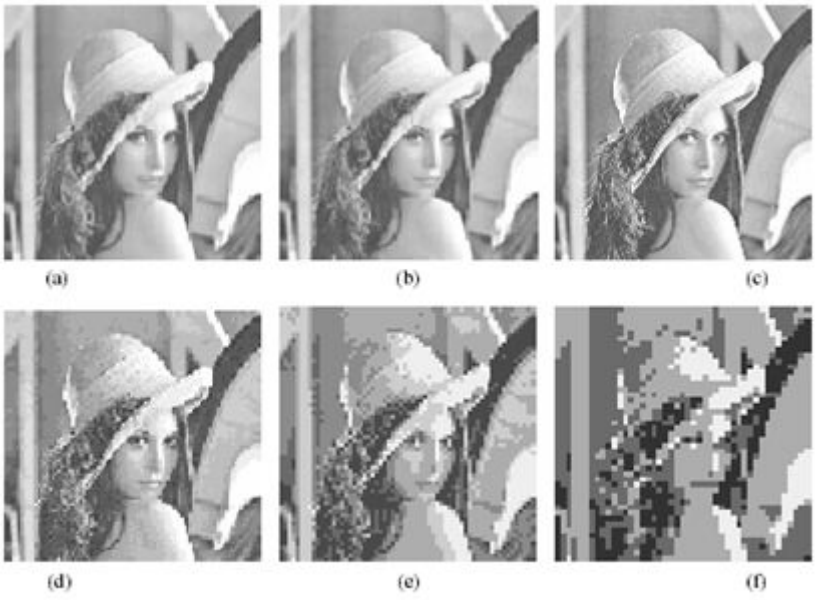


图象空间分辨率变化所产生的效果

3- 3

图象幅度分辨率变化所产生的效果

3- 4



图象空间和幅度分辨率同时变化所产生的效果

3.2 Intensity Transformation Functions

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

Because they depend only on intensity values, and on (*x,y*), intensity transformation functions in simplified form as:

$$s=T(r)$$

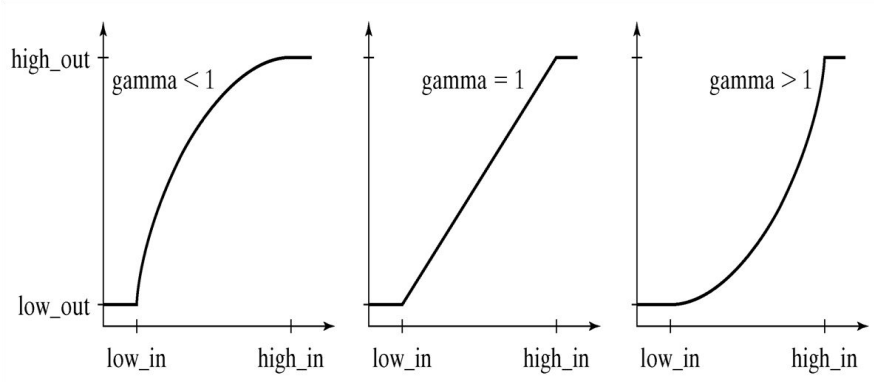
where *r* denotes the intensity of f and *s* the intensity of *g*, both at any corresponding point (*x,y*) in the images.

3.2.1 Function imadjust

It has the syntax

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

As illustrated in Fig. 3.2, this function maps the intensity values in image f to new values in g.



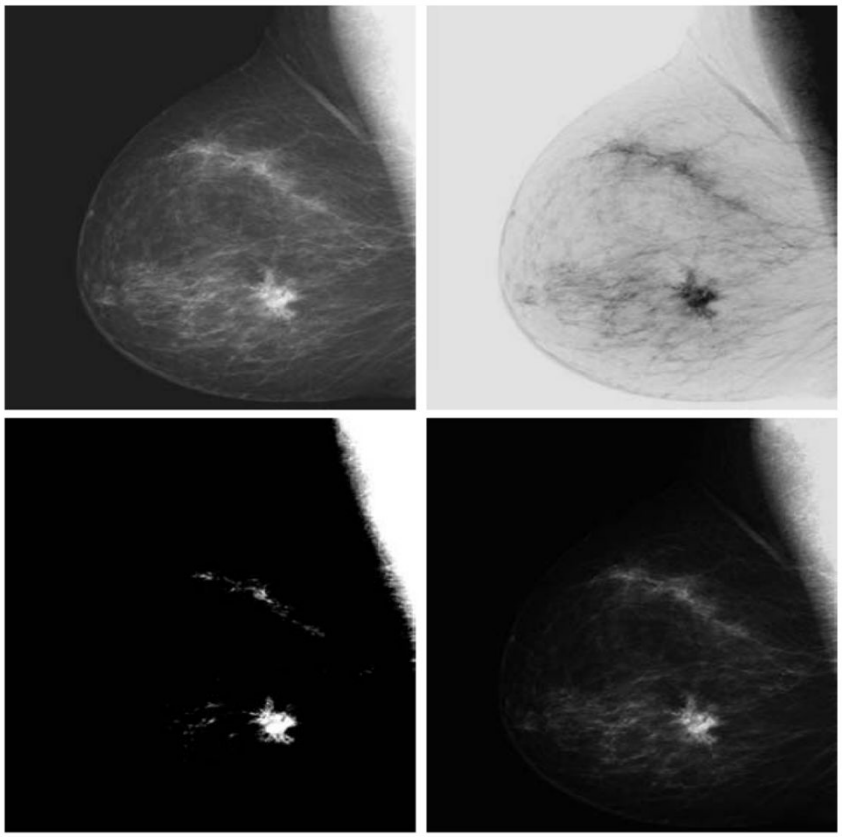
a b c

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

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

Examples:

- g1 = imadjust(f, [0 1], [1 0]);
- g2 = imadjust(f, [0.5 0.75], [0 1]);
- g3 = imadjust(f, [ ], [ ], 2);



a b
c d

FIGURE 3.3 (a) Original digital mammogram. (b) Negative image. (c) Result of expanding the intensity range [0.5, 0.75]. (d) Result of enhancing the image with `gamma = 2`. (Original image courtesy of G. E. Medical Systems.)

3.2.2 Logarithmic and Contrast-Stretching Transformations

Logarithmic and contrast-stretching transformations are basic tools for dynamic range manipulation.

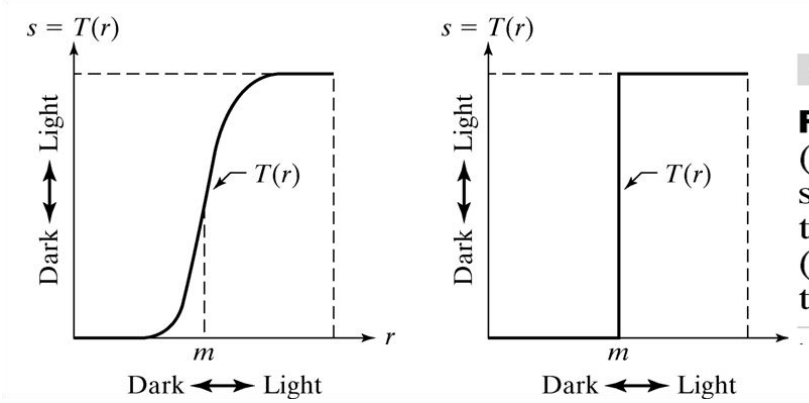
g = c*log(1 + double(f))

where c is a constant. The shape of this transformation is similar to the gamma curve shown in Fig. 3.2(a) that the shape of the gamma curve is variable, whereas the shape of the log function is fixed.

One of the principal uses of the log transformation is to compress dynamic range. By computing the log, a dynamic range on the order of, for example, 1.E6 is reduced to approximately 14, which is much more manageable.

gs = im2uint8(mat2gray(g));

Use of mat2gray brings the values to the range [0, 1] and im2uint8 brings them to the range [0, 255]. Later, in Section 3.2.3, we discuss a scaling function that automatically detects the class of the input and applies the appropriate conversion.



a b

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

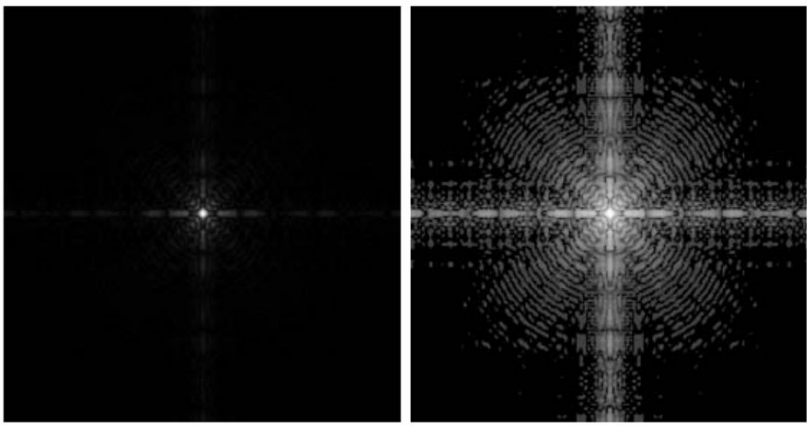
The function shown in Fig. 3.4(a) is called a *contrast-stretching* transformation function because it compresses the input levels lower than *m* into a narrow range of dark levels in the output image. Similarly, it compresses the values above *m* into a narrow band of light levels in the output. The result is an image of higher contrast. In fact, in the limiting case shown in Fig. 3.4(b), the output is a binary image. This limiting function is called a *thresholding* function, which, as we discuss in Chapter 10, is a simple tool used for image segmentation.

The function in Fig. 3.4(a) has the form

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

where *r* represents the intensities of the input image, *s* the corresponding intensity values in the output image, and *E* controls the slope of the function.This equation is implemented in MATLAB for an entire image as:

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



a b

FIGURE 3.5 (a) A Fourier spectrum. (b) Result obtained by performing a log transformation.

Figure 3.5(a) is a Fourier spectrum with values in the range 0 to 1.5*E6 displayed on a linearly scaled, 8-bit system. Figure 3.5(b) shows the result obtained using the commands

>> g = im2uint8(mat2gray(log(1 + double(f))));

>> imshow(g)

The visual improvement of g over the original image is quite evident.

3.2.3 Some Utility M-Functions for Intensity Transformations

Note in the following M-function, which we call *intrans*

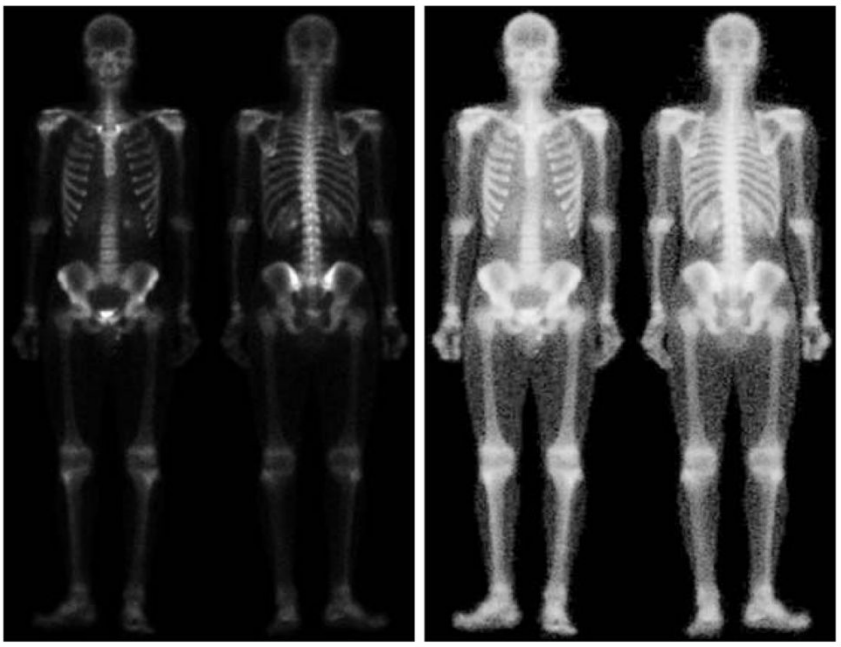
function g = intrans(f, varargin)

As an illustration of function intrans, consider the image in Fig. 3.6(a), Fig. 3.6(b) was obtained with the following call to intrans:

>>g = intrans(f, 'stretch', mean2(im2double(f)), 0.9);

figure, imshow(g)

→ g = 1./(1 + (m./(double(f) + eps)).^E)



a b

FIGURE 3.6 (a) Bone scan image. (b) Image enhanced using a contrast-stretching transformation. (Original image courtesy of G. E. Medical Systems.)

An M-Function for Intensity Scaling

When working with images, results whose pixels span a wide negative to positive range of values are common. While this presents no problems during intermediate computations, it becomes an issue when using an 8-bit or 16-bit format for saving or viewing an image, in which case it often is desirable to scale the image to the full, maximum range, [0, 255] or [0, 65535].

The following M-function, which we call gscale, accomplishes this. In addition, the function can map the output levels to a specified range.

See Appendix C for the listing.

The syntax of function gscale is:

>>**g = gscale(f, method, low, high)**

where f is the image to be scaled. Valid values for method are 'full8' (the default), which scales the output to the full range [0, 255], and 'full16', which scales the output to the full range [0, 65535].

3.3 Histogram Processing and Function Plotting

Intensity transformation functions based on information extracted from image intensity histograms play a basic role in image processing, such as enhancement, compression, segmentation, and description. The focus of this section is on obtaining, plotting, and using histograms for image enhancement. Other applications of

histograms are discussed in later chapters.

3.3.1 Generating and Plotting Image Histograms

The histogram of a digital image with L total possible intensity levels in the range [0,G] is defined as the discrete function:

$$h(r_k) = n_k$$

where r_k is the kth intensity level in the interval [0, G] and n_k is the number of pixels in the image whose intensity level is r_k.The value of G is 255 for images of class uint8, 65535 for images of class uint16, and 1.0 for images of class double.

Often, it is useful to work with *normalized* histograms, obtained simply by dividing all elements of $h(r_k)$ by the total number of pixels in the image, which we denote by n:

$$p(r_k) = \frac{h(r_k)}{n} = \frac{n_k}{n}$$

The core function in the toolbox for dealing with image histograms is imhist, which has the following basic syntax:

h = imhist(f, b)

where f is the input image, h is its histogram h(rk) ,and b is the number of bins used in forming the histogram (b = 256 default).

We obtain the normalized histogram simply by using the expression:

p = imhist(f, b)/numel(f)

Recall from Section 2.10.3 that function numel(f) gives the number of elements in array f (i.e., the number of pixels in the image).

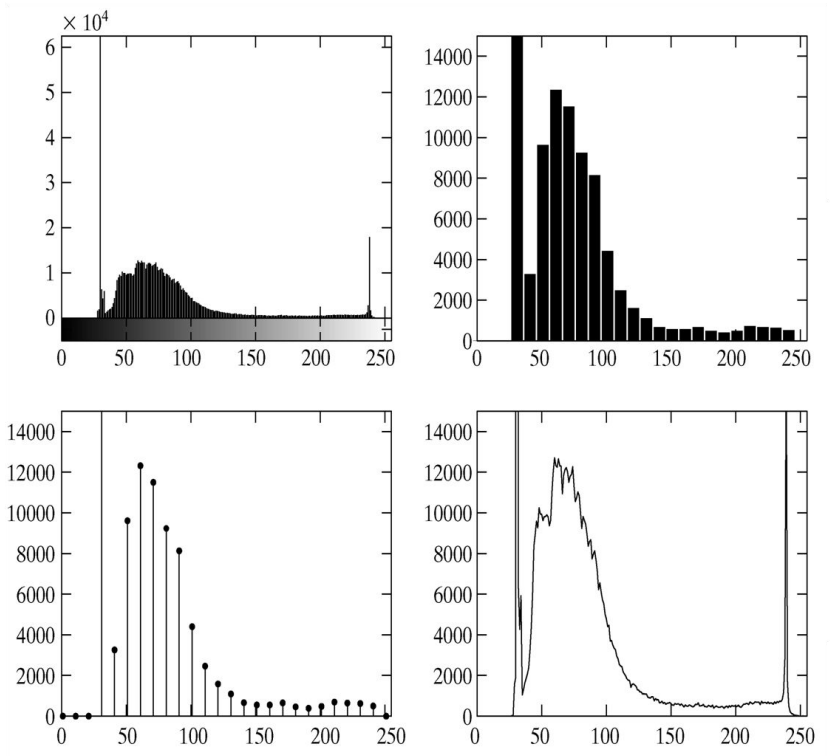
>>imhist(f);

Figure 3.7(a) shows the result.

Histograms often are plotted using bar graphs, we can use the function

bar(horz, v, width)

where v is a row vector containing the points to be plotted, horz is a vector of the same dimension as v that contains the increments of the horizontal scale, and width is a number between 0 and 1.



a b
c d

FIGURE 3.7
Various ways to plot an image histogram.
(a) `imhist`,
(b) `bar`,
(c) `stem`,
(d) `plot`.

SOME FUNCTION

Set, gca, xtick, ytick, axis, xlabel ylabel, text, title, stem, plot, ylim, xlim, hold on

Symbol	Color	Symbol	Line Style	Symbol	Marker
k	Black	–	Solid	+	Plus sign
w	White	--	Dashed	o	Circle
r	Red	:	Dotted	*	Asterisk
g	Green	-.	Dash-dot	.	Point
b	Blue	none	No line	x	Cross
c	Cyan			s	Square
y	Yellow			d	Diamond
m	Magenta			none	No marker

TABLE 3.1
Attributes for functions `stem` and `plot`. The `none` attribute is applicable only to function `plot`, and must be specified individually. See the syntax for function `plot` below.

3.3.2 Histogram Equalization

We perform the following transformation on the input levels to obtain output (processed) intensity levels, s,

$$s = T(r) = \int_0^r p_r(w)dw$$

Where w is a variable of integration. It can be shown (Gonzalez and Woods

[2002]) that the probability density function of the output levels is *uniform*; that is,

$$p_s(s) = \begin{cases} 1 & \text{for } 0 \le s \le 1 \\ 0 & \text{otherwise} \end{cases}$$

In other words, the preceding transformation generates an image whose intensity levels are equally likely, and, in addition, cover the entire range [0, 1]. The result of this intensity-level *equalization* process is an image with increased dynamic range, which will tend to have higher contrast. Note that the transformation function is really nothing more than the cumulative distribution function (CDF).

When dealing with discrete quantities we work with histograms and call the preceding technique *histogram equalization.* With reference to the discussion in Section 3.3.1, the histogram associated with the intensity levels of a given image, recall that the values in a normalized histogram are approximations to the probability of occurrence of each intensity level in the image. For discrete quantities the equalization transformation becomes:

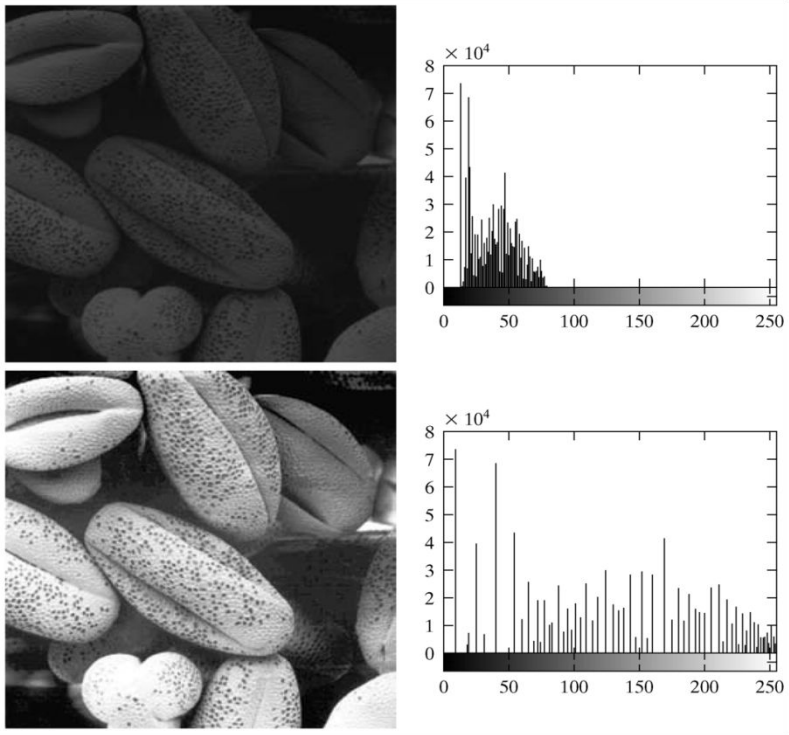
$$s_k = T(r_k) = \sum_{j=1}^{k} p_r(r_j) = \sum_{j=1}^{k} \frac{n_j}{n}$$

for k=1,2,..k where Sk is the intensity value in the output (processed)image corresponding to value rk in the input image.

g = histeq(f, nlev)

- >> imshow(f)
- >> figure, imhist(f)
- >> ylim('auto')

- >> g = histeq(f, 256);
- >> figure, imshow(g)
- >> figure, imhist(g)
- >> ylim('auto')



a b
c d

FIGURE 3.8 Illustration of histogram equalization. (a) Input image, and (b) its histogram. (c) Histogram-equalized image, and (d) its histogram. The improvement between (a) and (c) is quite visible. (Original image courtesy of Dr. Roger Heady, Research School of Biological Sciences, Australian National University, Canberra.)

A plot of cdf, shown in Fig. 3.9, was obtained using the following commands:

- >>hnorm=imhist(f)./numel(f);
- >>cdf=cumsum(hnorm);
- >> x = linspace(0, 1, 256);
- >> plot(x, cdf) % Plot cdf vs. x.
- >> axis([0 1 0 1]) % Scale, settings, and labels:
- >> set(gca, 'xtick', 0:.2:1)
- >> set(gca, 'ytick', 0:.2:1)
- >> xlabel('Input intensity values', 'fontsize', 9)
- >> ylabel('Output intensity values', 'fontsize', 9)
- >> text(0.18, 0.5, 'Transformation function', 'fontsize', 9)

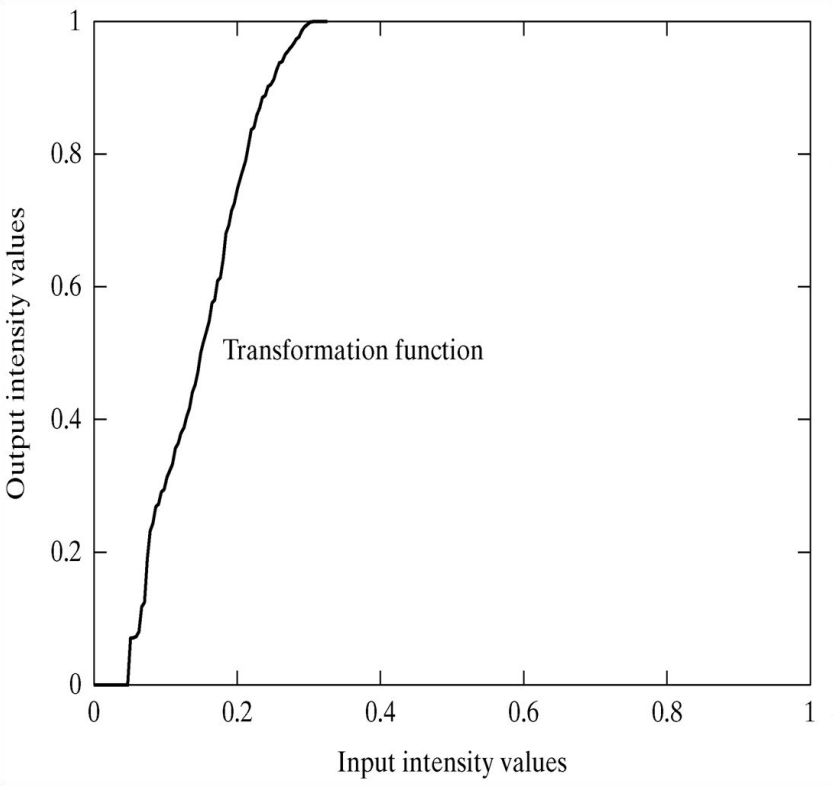


FIGURE 3.9 Transformation function used to map the intensity values from the input image in Fig. 3.8(a) to the values of the output image in Fig. 3.8(c).

3.3.3 Histogram Matching (Specification)

Histogram equalization produces a transformation function that is adaptive, based on histogram of a given image. Transformation function for an image has been computed, it does not change unless the histogram of the image changes.

Histogram equalization achieves enhancement by spreading the levels of the input image over a wider range of the intensity scale. It is useful in some applications to be able to specify the shape of the histogram that we wish the processed image to have. The method used to generate a processed image that has a specified histogram is called *histogram matching* or *histogram specification.*

Equations:

$$s = T(r) = \int_0^r p_r(w)dw$$

$$H(z) = \int_0^z p_z(w)dw$$

$$z = H^{-1}[T(r)]$$

The toolbox implements histogram matching using the following syntax:

g = histeq(f, hspec)

where f is the input image, hspec is the specified histogram (a row vector of

specified values), and g is the output image, its histogram approximates the specified histogram, hspec.

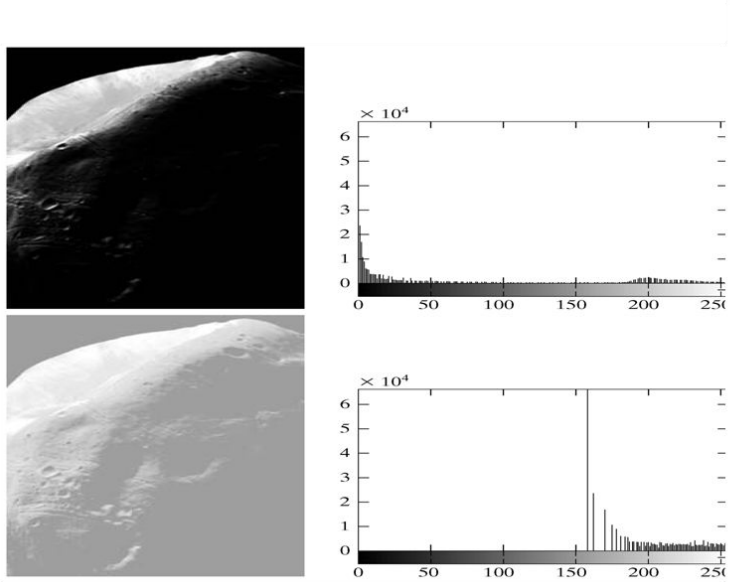


FIGURE 3.10 (a) Image of the Mars moon Phobos. (b) Histogram. (c) Histogram-equalized image. (d) Histogram of (c). (Original image courtesy of NASA).

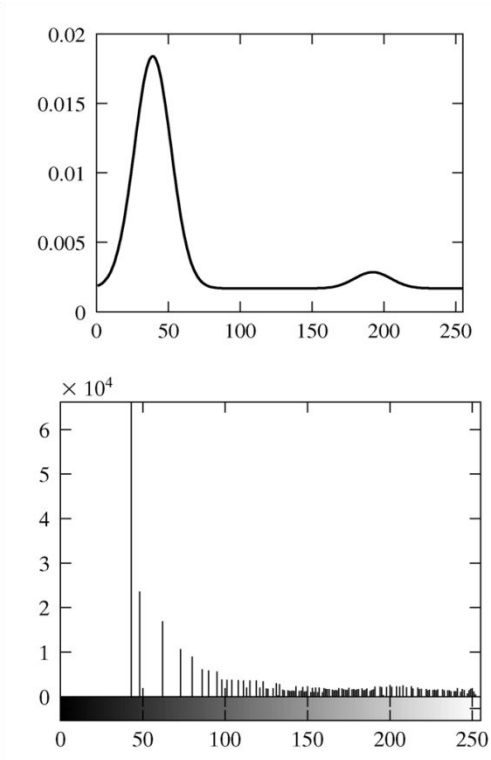
Figure 3.10(a) shows an image, f, of the Mars moon, Fig. 3.10(b) shows its histogram, obtained using imhist(f). The image is dominated by large, dark areas, resulting in a histogram characterized by a large concentration of pixels in the dark end of the gray scale.

At first glance, one might conclude that histogram equalization would be a good approach to enhance this image, so that details in the dark areas become more visible, the result in Fig. 3.10(c), obtained using the command

>> f1 = histeq(f, 256);

The following M-function computes a Gaussian function normalized to unit area, so it can be used as a specified histogram.

- function p = twomodegauss(m1, sig1, m2, sig2, A1, A2, k)
- function p = manualhist
- g = histeq(f, p);



a b
c

FIGURE 3.11
(a) Specified histogram.
(b) Result of enhancement by histogram matching.
(c) Histogram of (b).

Figure 3.11(b) shows the result. The improvement over the histogram equalized result in Fig. 3.10(c) is evident by comparing the two images. It is of interest to note that the specified histogram represents a rather modest change from the original histogram to obtain a significant improvement in enhancement. The histogram of Fig. 3.11(b) is shown in Fig. 3.11(c).The most distinguishing feature of this histogram is how its low end has been moved closer to the lighter region of the gray scale, and thus closer to the specified shape. However, that the shift to the right was not as extreme as the shift in the histogram shown in Fig. 3.10(d), which corresponds to the poorly enhanced image of Fig. 3.10(c).

3.4 Spatial Filtering

As mentioned in Section 3.1 and illustrated in Fig. 3.1, neighborhood processing consists of

(1) defining a center point;

(2) performing an operation that involves only the pixels in a predefined neighborhood about that center point;

(3) letting the result of that operation be the “response” of the process at that point;

(4) repeating the process for every point in the image. The process of moving the center point creates new neighborhoods, one for each pixel in the input image.

The two principal terms used to identify this operation are *neighborhood processing* and *spatial filtering*, with the second term being more prevalent. As explained in the following section, if the computations performed on the pixels of the neighborhoods are linear, the operation is called *linear spatial filtering* (the term *spatial convolution* also used); otherwise it is called *nonlinear spatial filtering.*

3.4.1 Linear Spatial Filtering

The concept of *linear filtering* has its roots in the use of the Fourier transform for signal processing in the frequency domain, a topic discussed in detail in Chapter 4. In chapter 2, filtering operations 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 consist of multiplying each pixel in the neighborhood by a corresponding coefficient and summing the results to obtain the response at each point (*x,y*). If the neighborhood is of size m*n ,mn coefficients are required.

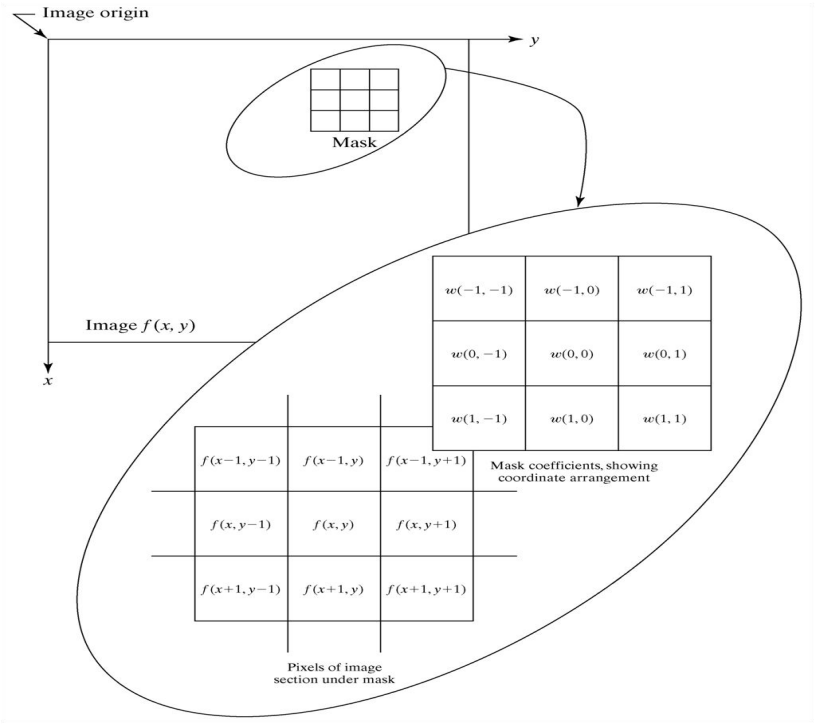


FIGURE 3.12 The mechanics of linear spatial filtering. The magnified drawing shows a 3 × 3 mask and the corresponding image neighborhood directly under it. The neighborhood is shown displaced out from under the mask for ease of readability.

The coefficients are arranged as a matrix, called a *filter*, *mask*, *filter mask*, *kernel*,

template, or *window*, with the first three terms being the most prevalent. For reasons that will become obvious shortly, the terms *convolution filter*, *mask*, or *kernel*, also are used.

There are two closely related concepts that must be understood clearly when performing linear spatial filtering. One is *correlation*; the other is *convolution*. Correlation is the process of passing the mask w by the image array f in the manner described in Fig. 3.12. Mechanically, convolution is the same process, except that w is rotated by 180° prior to passing it by f. These two concepts are best explained by some simple examples.

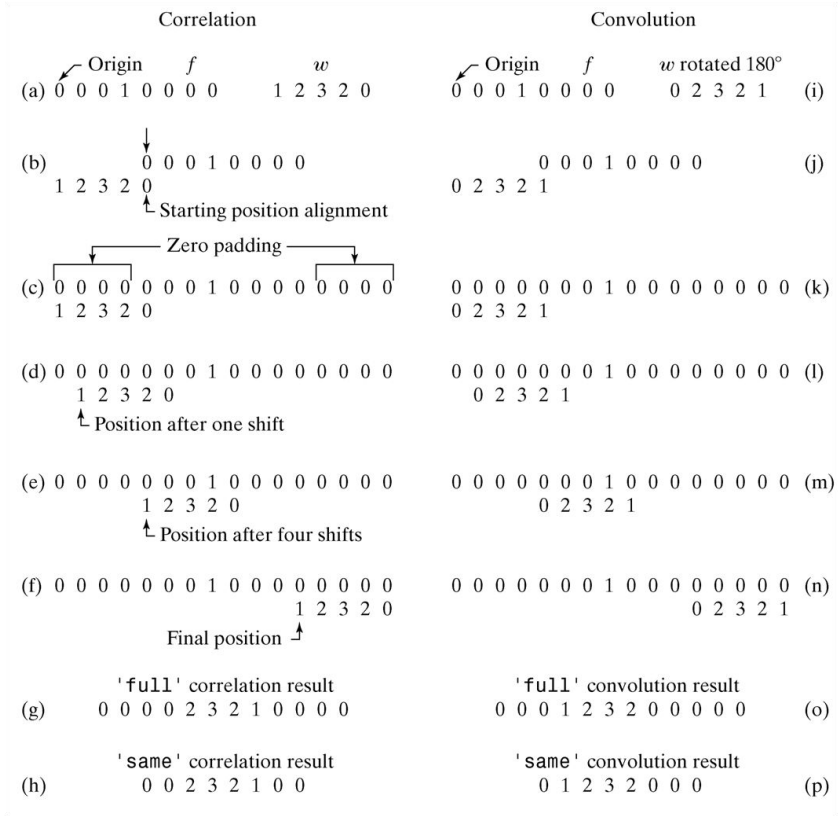


FIGURE 3.13 Illustration of one-dimensional correlation and convolution.

For convolution, we simply rotate w(x,y) by 180° and proceed in the same manner as in correlation [Figs. 3.14(f) through (h)]. As in the one-dimensional example discussed earlier, convolution yields the same result regardless of which of the two functions undergoes translation. In correlation the order does matter, a fact that is made clear in the toolbox by assuming that the filter mask is always the function that undergoes translation. Note also the important fact in Figs. 3.14(e) and (h) that the results of spatial correlation and convolution are rotated by 180° with respect to each other. This is expected because convolution is nothing more than correlation with a rotated filter mask.

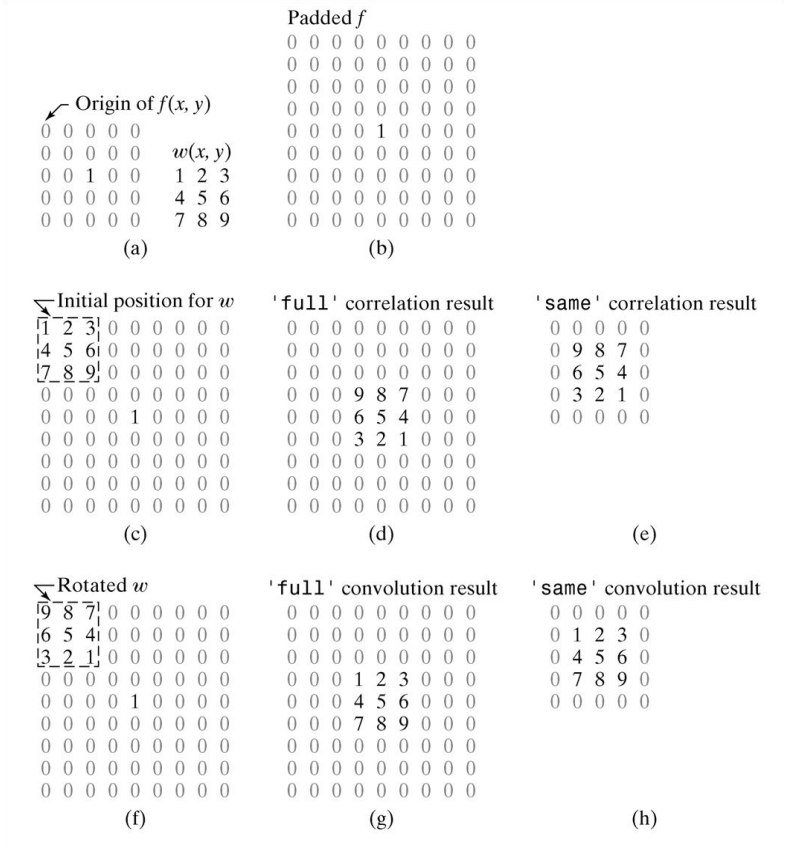


FIGURE 3.14 Illustration of two-dimensional correlation and convolution. The 0s are shown in gray to simplify viewing.

The toolbox implements *linear* spatial filtering using function imfilter:

g = imfilter(f, w, filtering_mode, boundary_options, size_options)

where f is the input image, w is the filter mask, g is the filtered result, and the other parameters are summarized in Table 3.2.

The filtering_mode specifies whether to filter using correlation ('corr') or convolution ('conv'). The boundary_options deal with the border-padding issue, with the size of the border being determined by the size of the filter. These options are explained further in Example 3.7. The size_options are either 'same' or 'full', as explained in Figs. 3.13 and 3.14.

The most common syntax for imfilter is

g = imfilter(f, w, 'replicate')

This syntax is used when implementing IPT standard linear spatial filters.

When working with filters that are neither pre-rotated nor symmetric, and we wish to perform convolution, we have two options. One is to use the syntax:

g = imfilter(f, w, 'conv', 'replicate')

The other approach is to preprocess w by using the function rot90(w, 2) to rotate it 180°, and then use imfilter(f, w, 'replicate'). Of course these two steps can be

combined into one statement. The preceding syntax produces an image g that is of the same size as the input.

Options	Description
Filtering Mode	
`'corr'`	Filtering is done using correlation (see Figs. 3.13 and 3.14). This is the default.
`'conv'`	Filtering is done using convolution (see Figs. 3.13 and 3.14).
Boundary Options	
`P`	The boundaries of the input image are extended by padding with a value, `P` (written without quotes). This is the default, with value 0.
`'replicate'`	The size of the image is extended by replicating the values in its outer border.
`'symmetric'`	The size of the image is extended by mirror-reflecting it across its border.
`'circular'`	The size of the image is extended by treating the image as one period a 2-D periodic function.
Size Options	
`'full'`	The output is of the same size as the extended (padded) image (see Figs. 3.13 and 3.14).
`'same'`	The output is of the same size as the input. This is achieved by limiting the excursions of the center of the filter mask to points contained in the original image (see Figs. 3.13 and 3.14). This is the default.

TABLE 3.2 Options for function `imfilter`.

Figure 3.15(a) is a class double image, f, of size 512*512 pixels. Consider the simple 31*31 filter

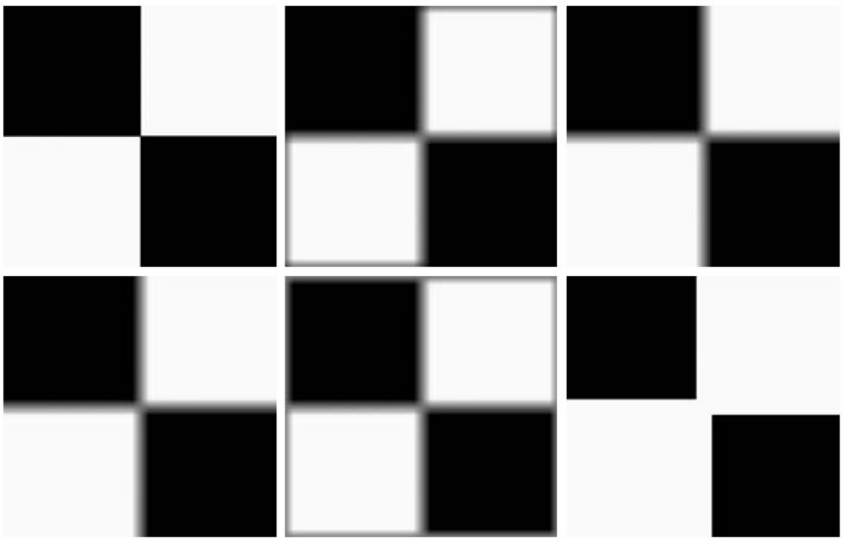
```
>> w = ones(31);
```

which is proportional to an averaging filter.

shows the result of performing the following filtering operation:

```
>> gd = imfilter(f, w);
        imshow(gd, [ ])
>> gr = imfilter(f, w, 'replicate');
      figure, imshow(gr, [ ])
>> gs = imfilter(f, w, 'symmetric');
      figure, imshow(gs, [ ])
>> gc = imfilter(f, w, 'circular');
      figure, imshow(gc, [ ])
>> f8 = im2uint8(f);
```

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



a b c
d e f

FIGURE 3.15
(a) Original image. (b) Result of using `imfilter` with default zero padding. (c) Result with the `'replicate'` option. (d) Result with the `'symmetric'` option. (e) Result with the `'circular'` option. (f) Result of converting the original image to class `uint8` and then filtering with the `'replicate'` option. A filter of size 31 × 31 with all 1s was used throughout.

3.4.2 Nonlinear Spatial Filtering

Nonlinear spatial filtering is based on neighborhood operations also, and the mechanics of defining m*n neighborhoods by sliding the center point through an image are the same as discussed in the previous section. 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 of a neighborhood.

For example, letting the response at each center point be equal to the maximum pixel value in its neighborhood is a nonlinear filtering operation. Another basic difference is that the concept of a mask is not as prevalent in nonlinear processing.

The idea of filtering carries over, but the “filter” should be visualized as a nonlinear function that operates on the pixels of a neighborhood, and whose response constitutes the response of the operation at the center pixel of the neighborhood.

The toolbox provides two functions for performing general nonlinear filtering: nlfilter and colfilt. The former performs operations directly in 2-D,while colfilt organizes the data in the form of columns. Although colfilt requires more memory, it generally executes significantly faster than nlfilter. In most image processing applications speed is an overriding factor, so colfilt is preferred over nlfilt for

implementing generalized nonlinear spatial filtering.

The syntax of function colfilt is

g = colfilt(f, [m n], 'sliding', @fun, parameters)

where m and n are the dimensions of the filter region, 'sliding' indicates that the process is one of sliding the m*n region from pixel to pixel in the input image f, @fun references a function, which we denote arbitrarily as fun, and parameters indicates parameters (separated by commas) that may be required by function fun. The symbol @ is called *a function handle*.

The linear filtering has provisions for padding to handle the border problems inherent in spatial filtering. When using colfilt, however, the input image must be padded explicitly before filtering. For this we use function padarray, which, for 2-D functions

fp = padarray(f, [r c], method, direction)

where f is the input image, fp is the padded image, [r c] gives the number of rows and columns by which to pad f, and method and direction are as explained in Table 3.3. For example, if f = [1 2; 3 4], the command

>> fp = padarray(f, [3 2], 'replicate', 'post')

produces the result

```
fp =
     1     2     2     2
     3     4     4     4
     3     4     4     4
     3     4     4     4
     3     4     4     4
```

nonlinear filter function, call it gmean:

```
function v = gmean(A)
mn = size(A, 1);
    % The length of the columns of A is always mn.
v = prod(A, 1).^(1/mn);
```

To reduce border effects, we pad the input image using, say, the 'replicate‘ option in function padarray:

```
>> f = padarray(f, [m n], 'replicate');
```

Finally, we call colfilt:

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

Options	Description
Method	
`'symmetric'`	The size of the image is extended by mirror-reflecting it across its border.
`'replicate'`	The size of the image is extended by replicating the values in its outer border.
`'circular'`	The size of the image is extended by treating the image as one period of a 2-D periodic function.
Direction	
`'pre'`	Pad before the first element of each dimension.
`'post'`	Pad after the last element of each dimension.
`'both'`	Pad before the first element and after the last element of each dimension. This is the default.

TABLE 3.3 Options for function padarray.

Some commonly used nonlinear filters can be implemented in terms of other MATLAB and IPT functions such as imfilter and ordfilt2 (see Section 3.5.2). Function spfilt in Section 5.3, for example, implements the geometric mean filter in Example 3.8 in terms of imfilter and the MATLAB log and exp functions. Function colfilt, however, remains the best choice for nonlinear filtering operations.

3.5 Image Processing Toolbox Standard Spatial Filters

In this section we discuss linear and nonlinear spatial filters supported by IPT. Additional nonlinear filters are implemented in Section 5.3.

3.5.1 Linear Spatial Filters

The toolbox supports a number of predefined 2-D linear spatial filters, obtained by using function fspecial, which generates a filter mask, w, using the syntax

w = fspecial('type', parameters)

where 'type' specifies the filter type, and parameters further define the specified filter. The spatial filters supported by fspecial are summarized in Table 3.4, including applicable parameters for each filter.

Type	Syntax and Parameters
'average'	fspecial('average', [r c]). A rectangular averaging filter of size r × c. The default is 3 × 3. A single number instead of [r c] specifies a square filter.
'disk'	fspecial('disk', r). A circular averaging filter (within a square of size 2r + 1) with radius r. The default radius is 5.
'gaussian'	fspecial('gaussian', [r c], sig). A Gaussian lowpass filter of size r × c and standard deviation sig (positive). The defaults are 3 × 3 and 0.5. A single number instead of [r c] specifies a square filter.
'laplacian'	fspecial('laplacian', alpha). A 3 × 3 Laplacian filter whose shape is specified by alpha, a number in the range [0, 1]. The default value for alpha is 0.5.
'log'	fspecial('log', [r c], sig). Laplacian of a Gaussian (LoG) filter of size r × c and standard deviation sig (positive). The defaults are 5 × 5 and 0.5. A single number instead of [r c] specifies a square filter.
'motion'	fspecial('motion', len, theta). Outputs a filter that, when convolved with an image, approximates linear motion (of a camera with respect to the image) of len pixels. The direction of motion is theta, measured in degrees, counterclockwise from the horizontal. The defaults are 9 and 0, which represents a motion of 9 pixels in the horizontal direction.
'prewitt'	fspecial('prewitt'). Outputs a 3 × 3 Prewitt mask, wv, that approximates a vertical gradient. A mask for the horizontal gradient is obtained by transposing the result: wh = wv'.
'sobel'	fspecial('sobel'). Outputs a 3 × 3 Sobel mask, sv, that approximates a vertical gradient. A mask for the horizontal gradient is obtained by transposing the result: sh = sv'.
'unsharp'	fspecial('unsharp', alpha). Outputs a 3 × 3 unsharp filter. Parameter alpha controls the shape; it must be greater than 0 and less than or equal to 1.0; the default is 0.2.

TABLE 3.4 Spatial filters supported by function fspecial.

We illustrate the use of fspecial and imfilter by enhancing an image with a Laplacian filter. And Function fspecial('laplacian', alpha) implements a more general Laplacian mask: Enhance the image in Fig. 3.16(a) using the Laplacian. This image is a mildly blurred image of the North Pole of the moon. Enhancement in this case consists of sharpening the image, we generate and display the Laplacian filter:

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

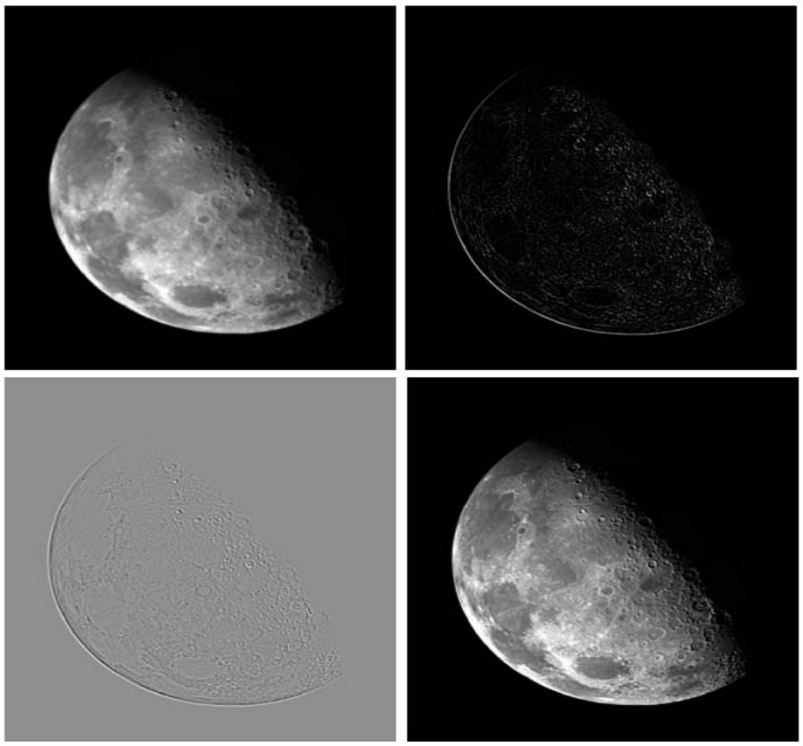
Note that the filter is of class double, and that its shape with alpha = 0 is the Laplacian filter discussed ,easily have specified this shape manually as

```
>> w = [0 1 0; 1 –4 1; 0 1 0];
```

Next we apply w to the input image, f, which is of class uint8:

```
>> g1 = imfilter(f, w, 'replicate');
>> imshow(g1, [ ])
```

Figure 3.16(b) shows the resulting image.



a b
c d

FIGURE 3.16
(a) Image of the North Pole of the moon.
(b) Laplacian filtered image, using `uint8` formats.
(c) Laplacian filtered image obtained using `double` formats.
(d) Enhanced result, obtained by subtracting (c) from (a).
(Original image courtesy of NASA.)


```
>> f2 = im2double(f);
>> g2 = imfilter(f2, w, 'replicate');
>> imshow(g2, [ ])
```

The result, shown in Fig. 3.16(c), is more what a properly processed Laplacian image should look like. Finally, we restore the gray tones lost by using the Laplacian by subtracting (because the center coefficient is negative) the Laplacian image from the original image:

```
>> g = f2 – g2;
>> imshow(g)
```

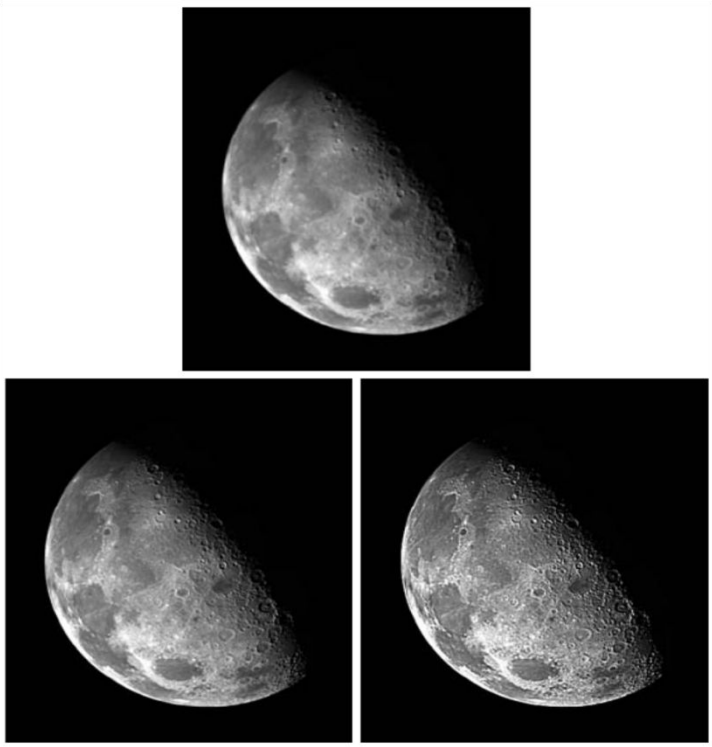
The result, shown in Fig. 3.16(d), is sharper than the original image.

Example 3.10:

Enhancement problems often require the specification of filters beyond those available in the toolbox. The Laplacian is a good example. Usually, sharper enhancement is obtained by using the 3*3 Laplacian filter that has a –8 in the center and is surrounded by 1s, as discussed earlier.

```
>> f = imread('moon.tif');
>> w4 = fspecial('laplacian', 0); % Same as w in Example 3.9.
>> w8 = [1 1 1; 1 –8 1; 1 1 1];
```

```
>> f = im2double(f);
>> g4 = f – imfilter(f, w4, 'replicate');
>> g8 = f – imfilter(f, w8, 'replicate');
>> imshow(f)
>> figure, imshow(g4)
>> figure, imshow(g8)
```



a
b c

FIGURE 3.17 (a) Image of the North Pole of the moon. (b) Image enhanced using the Laplacian filter `'laplacian'`, which has a −4 in the center. (c) Image enhanced using a Laplacian filter with a −8 in the center.

3.5.2 Nonlinear Spatial Filters

A commonly-used tool for generating nonlinear spatial filters in IPT is function ordfilt2, which generates order-statistic filters (also called rank filters). These are nonlinear spatial filters whose response is based on ordering (ranking) the pixels contained in an image neighborhood and then replacing the value of the center pixel in the neighborhood with the value determined by the ranking result.

Attention is focused in this section on nonlinear filters generated by ordfilt2. A number of additional nonlinear filters are developed and implemented in Section 5.3. The syntax of function ordfilt2 is

g = ordfilt2(f, order, domain)

For example, to implement a *min filter* (order 1) of size we use the syntax

g = ordfilt2(f, 1, ones(m, n))

This corresponds to a max filter, which is implemented using the syntax

g = ordfilt2(f, m*n, ones(m, n))

We can use MATLAB function median in ordfilt2 to create a median filter:

g = ordfilt2(f, median(1:m*n), ones(m, n))

Function median has the general syntax

v = median(A, dim)

Because of its practical importance, the toolbox provides a specialized implementation of the 2-D median filter:

g = medfilt2(f, [m n], padopt)

where the [m n] defines a neighborhood of size m × n over which the median is computed, and padopt specifies one of three possible border padding options: 'zeros' (the default), 'symmetric' in which f is extended symmetrically by mirror-reflecting it across its border, and 'indexed', in which f is padded with 1s if it is of class double and with 0s otherwise.

g = medfilt2(f)

which uses a neighborhood to compute the median, and pads the border of the input with 0s.

>> fn = imnoise(f, 'salt & pepper', 0.2);

Figure 3.18(c) is the result of median filtering this noisy image, using the statement:

>> gm = medfilt2(fn);

gms = medfilt2(fn, 'symmetric');

The result, shown in Fig. 3.18(d),

Summary

In addition to dealing with image enhancement, the material is the foundation for numerous topics in subsequent chapters.

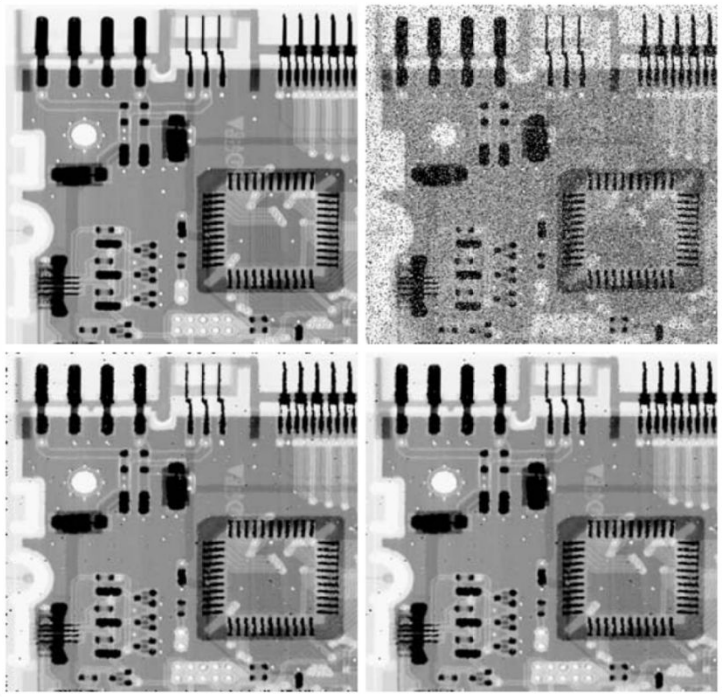
We will encounter spatial processing again in Chapter 5 in connection with image restoration, where we also take a closer look at noise reduction and noise-generating functions in MATLAB.

Some of the spatial masks that were mentioned briefly here are used extensively in Chapter 10 for edge detection in segmentation applications. The concept of

convolution and correlation is explained again in Chapter 4 from the perspective of the frequency domain.

Mask processing and the implementation of spatial filters will surface in various discussions throughout the book.

We extend the discussion begun here and introduce additional aspects of how spatial filters can be implemented efficiently in MATLAB.



a b
c d

FIGURE 3.18 Median filtering, (a) X-ray image. (b) Image corrupted by salt-and-pepper noise. (c) Result of median filtering with `medfilt2` using the default settings. (d) Result of median filtering using the `'symmetric'` image extension option. Note the improvement in border behavior between (d) and (c). (Original image courtesy of Lixi, Inc.)

Chapter4 Frequency Domain Processing

Preview

(1). Filter in frequency domain with Fourier Transform (FF).

(2). FF flexibility for design and implementation of filter.

(3). How to do in MATLAB for filtering

(4). Combine and compare spatial and frequency domain processing.

4.1 The 2-D Discrete Fourier Transform

Define Let f(x,y), x=0,1,…,M-1, y=0,1,…,N-1, 2D discrete Fourier transform (DFT):

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

For u=0,1,…,M-1, v=0,1,…,N-1

The frequency domain is simply the coordinate system spanned by F(u,v) with u and v as (frequency) variables.

Inverse discrete Fourier transform:

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

Denote and DFT properties:

F(0,0) ====> F(1,1)

f(0,0) ====> f(1,1)

$$F(u,v)=|F(u,v)|e^{-j\phi(u,v)}=R(u,v)+jI(u,v)$$

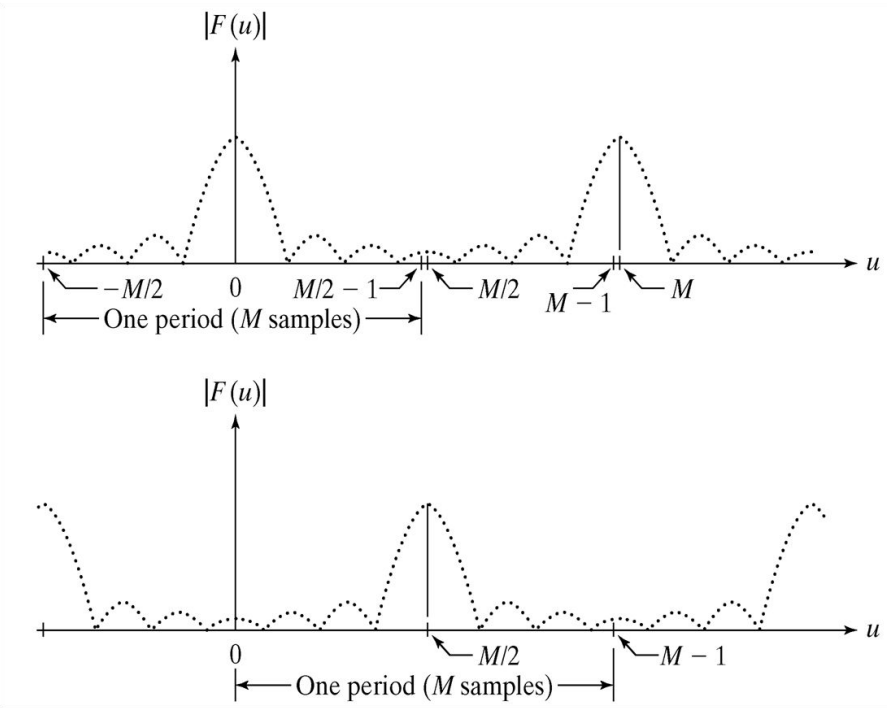
$$|F(u,v)|=[R^2(u,v)+I^2(u,v)]^{1/2}$$

$$\phi(u,v)=arctg\frac{I(u,v)}{R(u,v)}$$

Symmetry property：

$$f^*(x,y)=f(x,y)$$

$$F^*(u,v)=F(-u,-v)$$



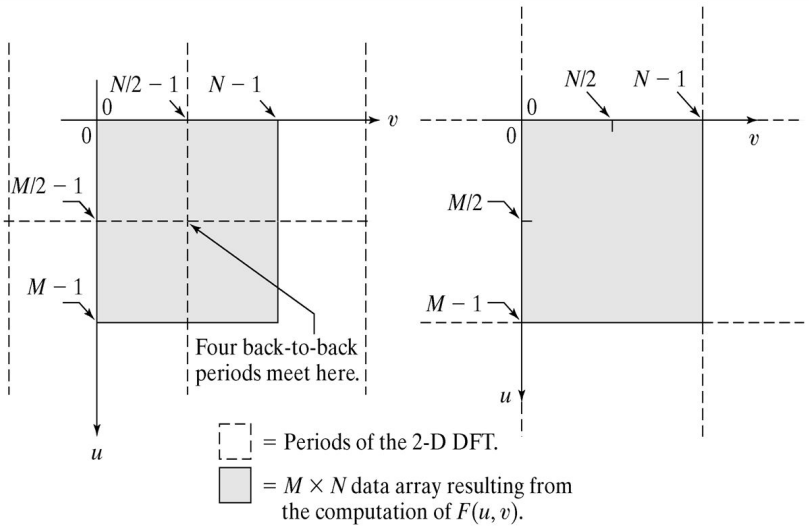
a
b

FIGURE 4.1 (a) Fourier spectrum showing back-to-back half periods in the interval $[0, M-1]$. (b) Centered spectrum in the same interval, obtained by multiplying $f(x)$ by $(-1)^x$ prior to computing the Fourier transform.

$$f(x,y)e^{j2\pi(u_0x+v_0y)/N} \Rightarrow F(u-u_0,v-v_0)$$

$$e^{j2\pi(u_0x+v_0y)/N} = e^{j2\pi(x+y)\frac{N}{2}N} = e^{j\pi(x+y)} = (-1)^{x+y}$$

$$f(x,y)(-1)^{x+y} \Rightarrow F(u-\frac{N}{2},v-\frac{N}{2})$$



a b

FIGURE 4.2 (a) $M \times N$ Fourier spectrum (shaded), showing four back-to-back quarter periods contained in the spectrum data. (b) Spectrum obtained by multiplying $f(x, y)$ by $(-1)^{x+y}$ prior to computing the Fourier transform. Only one period is shown shaded because this is the data that would be obtained by an implementation of the equation for $F(u, v)$.

The preceding discussion for centering the transform by multiplying f(x,y) by (-1).^(x+y) is an important concept that is included here for completeness.

When working in MATLAB, the approach is to compute the transform without multiplication by (-1).^(x+y) and then to rearrange the data afterwards using function fftshift. This function and its use are discussed in the following section.

4.2 Computing and Visualizing the 2-D DFT in MATLAB

(1). fft2 F=fft2(f, P, Q)

(2). abs S=abs(F)

F=fft2(f))

S=abs(F)

imshow(S,[ ])

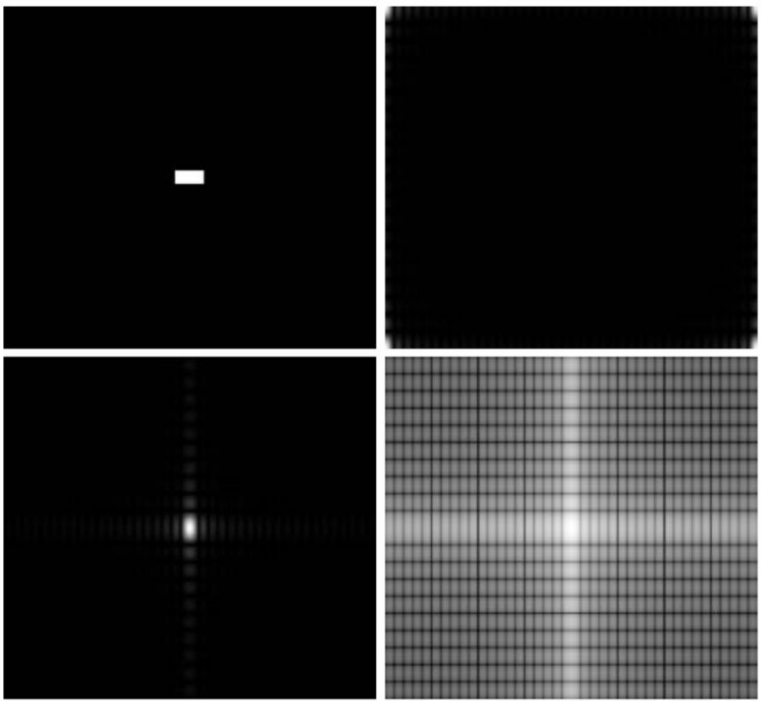
Fc=fftshift(F)

Fc=fftshift(F)

imshow(abs(Fc),[ ])

S2=log(1+abs(Fc))

imshow(S2,[ ])



a b
c d

FIGURE 4.3
(a) A simple image.
(b) Fourier spectrum.
(c) Centered spectrum.
(d) Spectrum visually enhanced by a log transformation.

Floor, ceil

f=ifft2(F)

f=real(ifft2(F))

4.3 Filtering in the Frequency Domain

Filtering in the frequency domain is quite simple conceptually. We give a brief overview of the concepts involved in frequency domain filtering and its implementation in MATLAB.

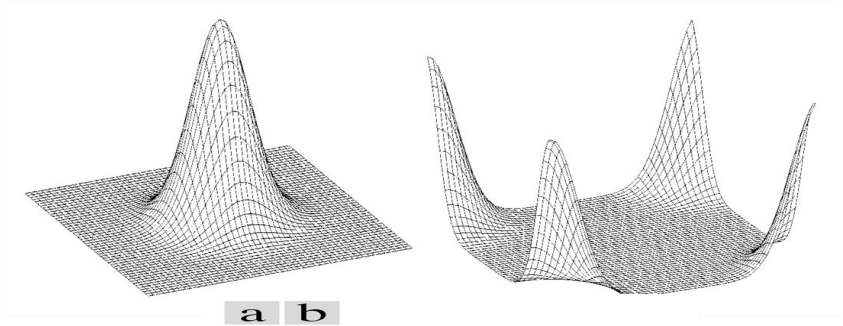
4.3.1 Fundamental Concepts

The foundation for linear filtering in both the spatial and frequency domains is the convolution theorem,which are:

f(x,y)*h(x,y) ====> H(u,v)F(u,v)

f(x,y)h(x,y) ===> H(u,v)*F(u,v)

* indicates convolution of two functions， h the filter, H the filter transfer function。



a b

FIGURE 4.4 Transfer functions of (a) a centered lowpass filter, and (b) the format used for DFT filtering. Note that these are frequency domain filters.

Based on the convolution theorem, to obtain the corresponding filtered image in the spatial domain, we simply compute the inverse Fourier transform of the product H(*u,v*)F(*u,v*). It is important to keep in mind that the process just described is identical to what we could obtain by using convolution in the spatial domain,as long as the filter mask,h(*x,y*) is the inverse Fourier transform of H(*u,v*).

In practice,spatial convolution generally is simplified by using small masks that

attempt to capture the features of their frequency domain counterparts.

Wraparound error:

Images and their transforms are automatically considered periodic if we elect to work with DFTs to implement filtering.

It is not difficult to visualize that convolving periodic functions can cause interference between adjacent periods if the periods are close with respect to the duration of the nonzero parts of the functions.

This interference called wraparound error, can be avoided by padding the functions with zeros in the following manner.

Function paddedsize( ):

Assume that functions f(x,y) and h(x,y) are of size A*B and C*D,then we form two extended (padded) functions both of size P*Q by appending zeros to f and h.

P>=A+C-1

Q>=B+D-1

Function PQ=paddedsize(AB,CD,PARAM)

F=fft2(f, PQ(1), PQ(2))

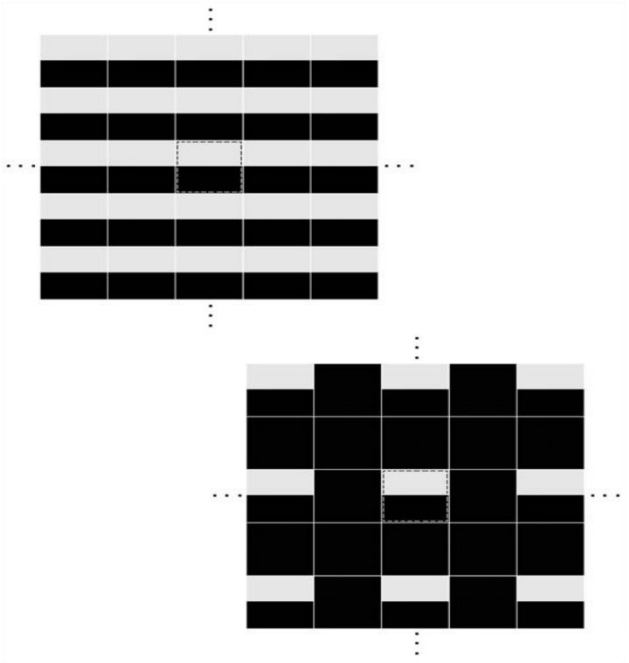
Example 4.1 padding and without padding

- [M,N]=size(f);
- F=fft2(f);
- sig=10;
- H=lpfilter('gaussian',M,N,sig);
- G=H.*F
- g=real(ifft2(G));
- imshow(g,[ ])

a b c

FIGURE 4.5 (a) A simple image of size 256 × 256. (b) Image lowpass-filtered in the frequency domain without padding. (c) Image lowpass-filtered in the frequency domain with padding. Compare the light portion of the vertical edges in (b) and (c).

4- 5



a
b

FIGURE 4.6
(a) Implied, infinite periodic sequence of the image in Fig. 4.5(a). The dashed region represents the data processed by `fft2`. (b) The same periodic sequence after padding with 0s. The thin white lines in both images are shown for convenience in viewing; they are not part of the data.

4- 6

FIGURE 4.7 Full padded image resulting from `ifft2` after filtering. This image is of size 512 × 512 pixels.

4.3.2 Basic steps in DFT filtering

(1).obtain the padding parameters

PQ=paddedsize(size(f));

(2).obtain the Fourier transform with padding

F=fft2(f, PQ(1), PQ(2));

(3).before using filter, let H=fftshift(H)

(4).multiply the transform by the filter G=H.*F

(5).obtain the real part of the inverse FFT of image

g=real(ifft2(G));

(6).crop the top ,left rectangle to the original size

g=g(1:size(f,1), 1:size(f,2));

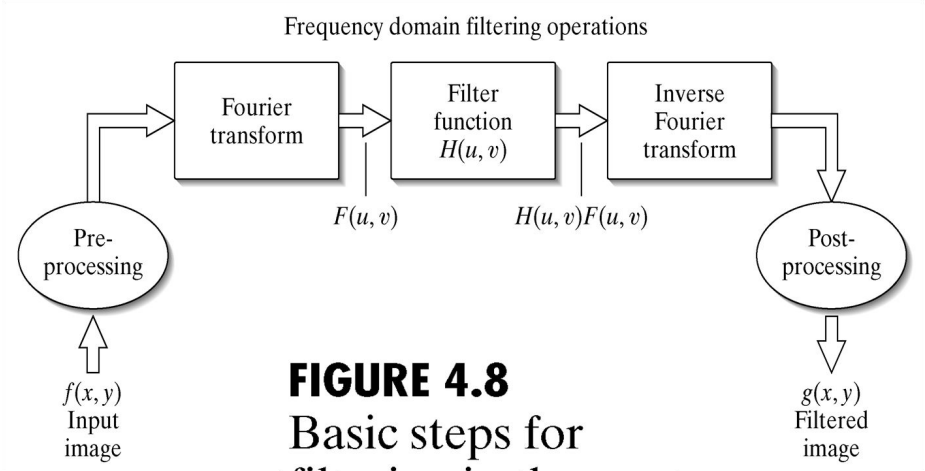


FIGURE 4.8 Basic steps for filtering in the frequency domain.

4.3.3 M-function

Use Function g=dftfilt(f, H) to finish the filtering image with above 6 steps

4.4 obtaining frequency domain filters from spatial filters

We are interested on two major topics:

(1). How to convert spatial filters into equivalent frequency domain filters;

(2). How to compare the results between spatial domain filtering using function imfilter, and frequency domain filtering using the techniques discussed in the pervious section.

Function freqz2 computes the frequency response of FIR filters,which as mentioned at the end of section,are the only linear filters considered in this book.

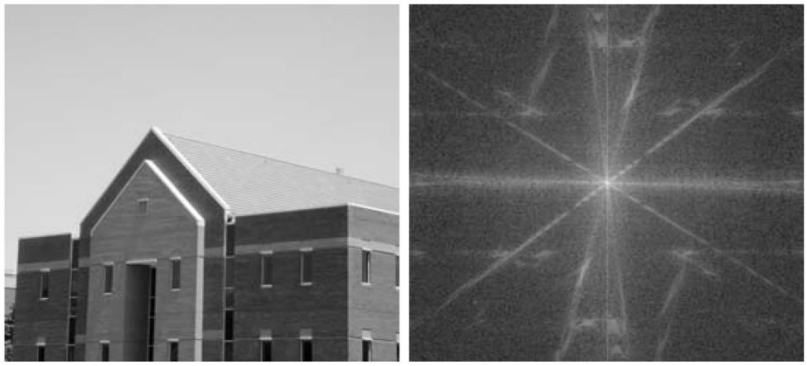
H=freqz(h, R, C)

Where h is a 2-D spatial filter and H is the corresponding 2-D frequency domain filter. R is the number of rows,and C the number of columns that we wish filter H to have.

Example 4.2

- F=fft2(f);
- S=fftshift(log(1+abs(F)));

- s=gscale(S);
- imshow(s)



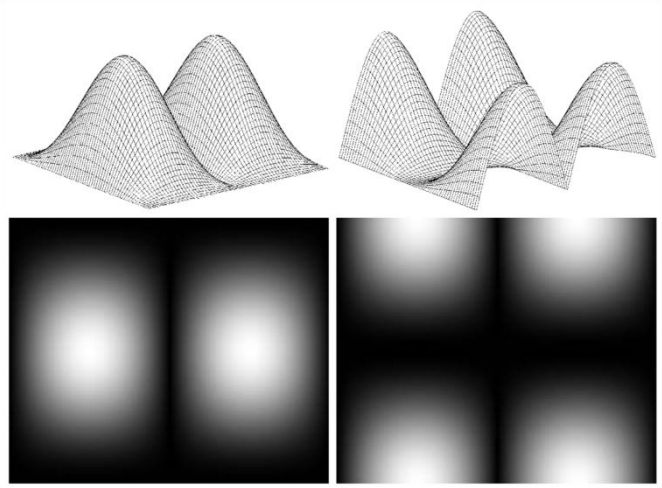
a b

FIGURE 4.9
(a) A gray-scale image. (b) Its Fourier spectrum.

4- 9

Generate the spatial filter using function fspecial:

- h=fspecial('sobel')
- h= 1 0 -1
  2 0 -2
  1 0 -1



a b
c d

FIGURE 4.10
(a) Absolute value of the frequency domain filter corresponding to a vertical Sobel mask. (b) The same filter after processing with function `fftshift`. Figures (c) and (d) are the filters in (a) and (b) shown as images.

4- 10

- Gs=imfilter(double(f),h);

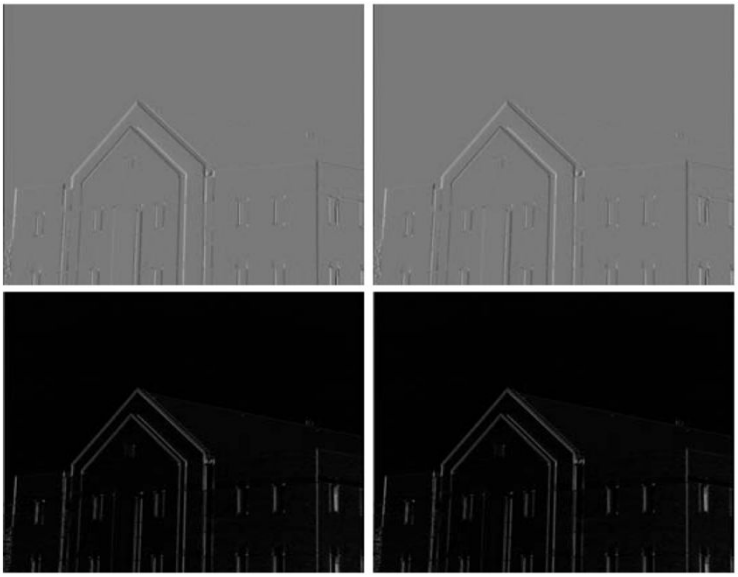
imshow(Gs, [ ])

- PQ=paddedsize(size(f));

  H=freqz2(h, PQ(1), PQ(2));

  H1=ifftshift(H);

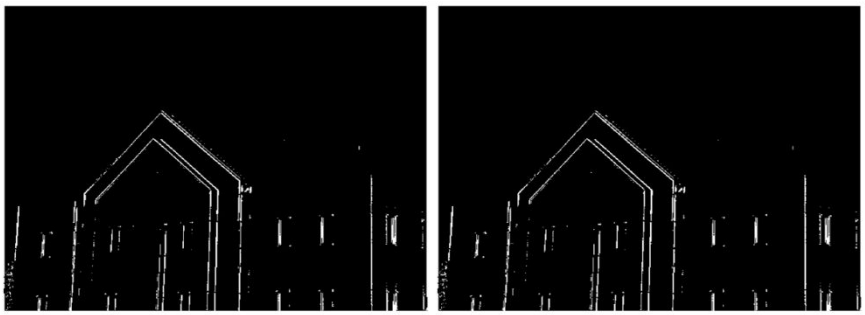
- Gf=dftfilt(f, H1);
  imshow(Gf, [ ])



a b
c d

FIGURE 4.11
(a) Result of filtering Fig. 4.9(a) in the spatial domain with a vertical Sobel mask. (b) Result obtained in the frequency domain using the filter shown in Fig. 4.10(b). Figures (c) and (d) are the absolute values of (a) and (b), respectively.

- imshow(abs(Gs), [ ])
- imshow(abs(Gf), [ ])
- imshow(abs(gs)>0.2*abs(max(gs(:))))
- imshow(abs(gf)>0.2*abs(max(gf(:))))



a b

FIGURE 4.12
Thresholded versions of Figs. 4.11(c) and (d), respectively, to show the principal edges more clearly.

4.5 Generating filters directly in the frequency domain

(1).How to implement filter functions directly in the frequency domain

(2).Several well-known smoothing (lowpass)filters, sharpening(highpass) filters

4.5.1 Creating meshgrid arrays for use in implementing filters in the frequency domain

Because FFT computations in MATLAB assume that the origin of the transform is at the top, left of the frequency rectangle, our distance computations are with respect to that point. The data can be rearranged for visualization purposes by using function fftshift.

Function dftuv (fftshift)

dftuv provides the necessary meshgrid array for use in distance computations and other similar applications, the meshgrid arrays generated by dftuv are in the order required for processing with fft2 of ifft2,so no rearranging of the data is required.

Example 4.3

[u,v]=dftuv(8,5);

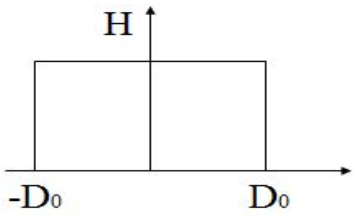
D=U.^2+V.^2 fftshift(D)

D=										
	0	1	4	4	1	20	17	16	17	20
	1	2	5	5	2	13	10	9	10	13
	4	5	8	8	5	8	5	4	5	8
	9	10	13	13	10	5	2	1	2	5
	16	17	20	20	17	4	1	0	1	4
	9	10	13	13	10	5	2	1	2	5
	4	5	8	8	5	8	5	4	5	8
	1	2	5	5	2	13	10	9	10	13

4.5.2 Lowpass frequency domain filter

Ideal lowpass filter(ILPF)

$$H(u,v)=\begin{cases}1 & D(u,v)\le D_0\\ 0 & D(u,v)>D_0\end{cases}$$



Butterworth lowpass filter(BLPF)

$$H(u,v)=\frac{1}{1+\left[\dfrac{D(u,v)}{D_0}\right]^{2n}}$$

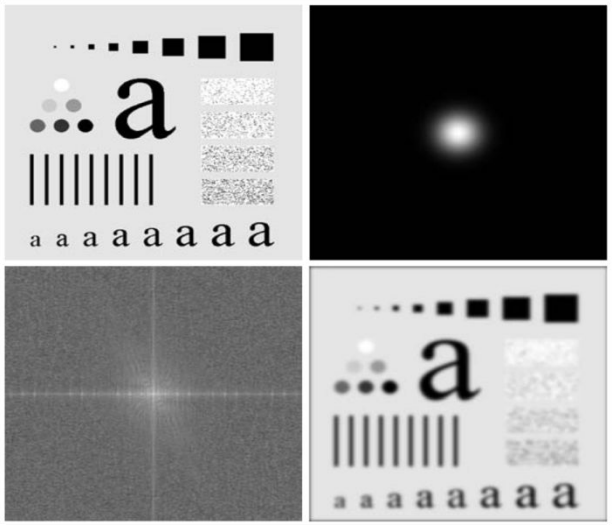
$$H(u,v)=\frac{1}{1+[\sqrt{2}-1][D(u,v)/D_0]^{2n}}$$

Gaussian lowpass filter(GLPF)

$$H(u,v)=e^{-D^2(u,v)/2\sigma^2}$$

Example 4.4 lowpass filtering

- PQ=paddedsize(size(f));
- [U,V]=dftuv(PQ(1), PQ(2));
- D0=0.05*PQ(2);
- F=fft2(f, PQ(1), PQ(2));
- H=exp(-(U.^2+V.^2)/(2*(D0^2)));
- g=dftfilt(f, H);
- imshow(fftshift(H), [ ]);
- imshow(log(1+abs(fftshift(F))), [ ])
- imshow(g, [ ])



a b
c d

FIGURE 4.13
Lowpass filtering. (a) Original image. (b) Gaussian lowpass filter shown as an image. (c) Spectrum of (a). (d) Processed image.

Function lpfilter (type-ideal,btw,gaussian):

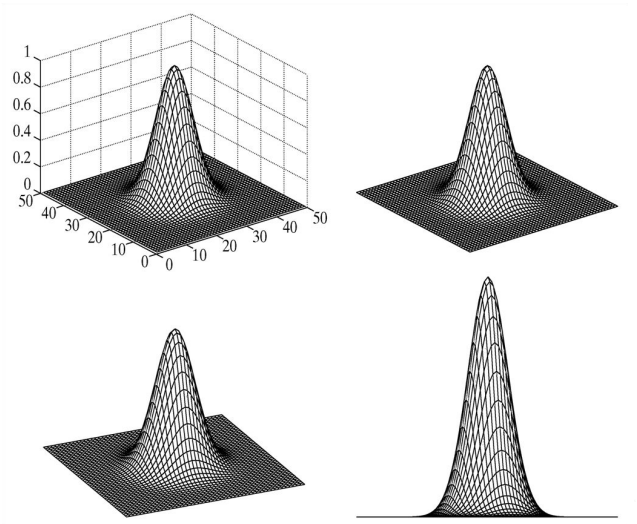
Function[H,D]=lpfilter(type, M, N, D0, n)

4.5.3 Wireframe and surface plotting

Function mesh,colormap, grid,view

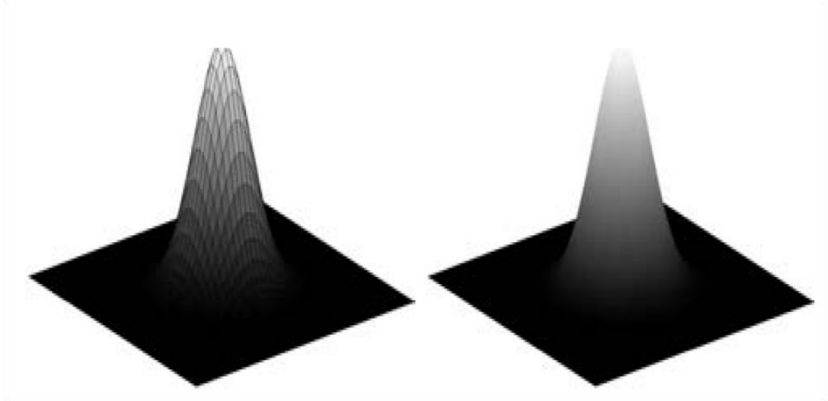
Example 4.5 wireframe plotting

- H=fftshift(lpfilter('gaussian', 500,500,50))



a b
c d

FIGURE 4.15
(a) A plot obtained using function `mesh`. (b) Axes and grid removed. (c) A different perspective view obtained using function `view`. (d) Another view obtained using the same function.



a b

FIGURE 4.16
(a) Plot obtained using function `surf`. (b) Result of using the command `shading interp`.

4.6 Sharpening frequency domain filters

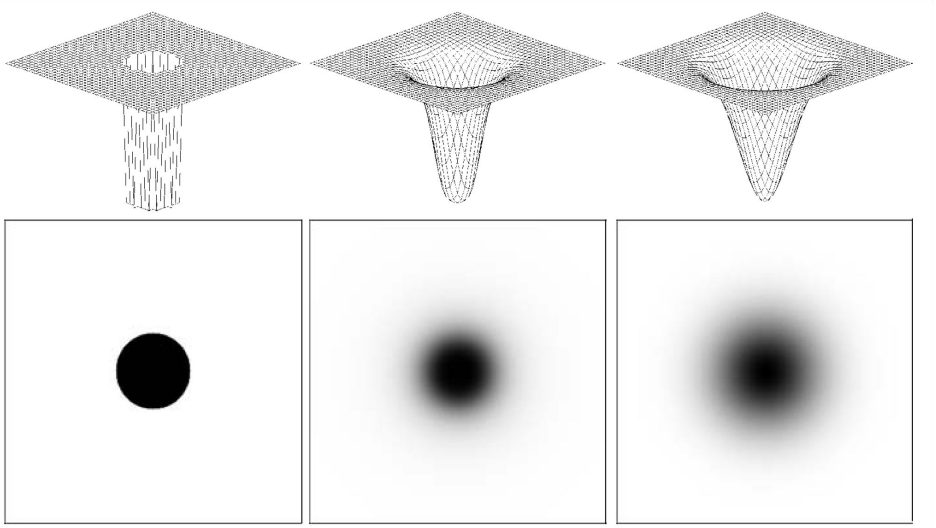
Just as lowpass filtering blurs an image,the opposite process, highpass filtering sharpens the image by attenuating the low frequencies and leaving the high frequencies of the Fourier transform relatively unchanged.

4.6.1 Basic highpass filtering

Hhp(u,v)=1-Hlp(u,v)

Function hpfilter (type-ideal, btw, gaussian)

function H=hpfilter(type, M, N, D0, n)

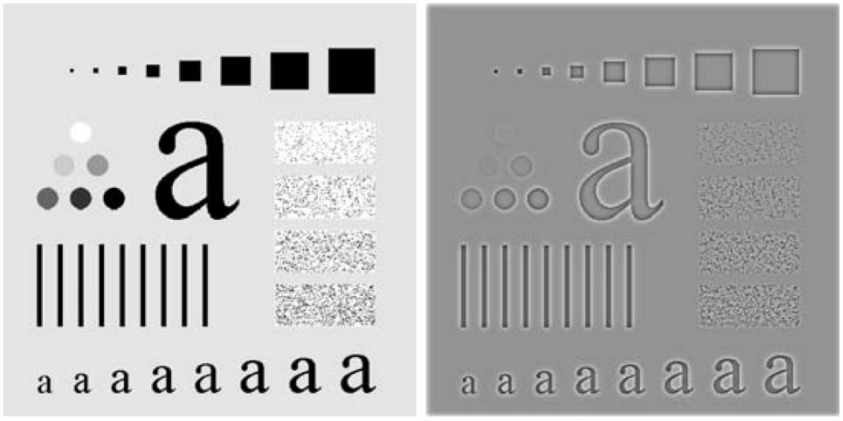


a b c
d e f

FIGURE 4.17 Top row: Perspective plots of ideal, Butterworth, and Gaussian highpass filters. Bottom row: Corresponding images.

Example 4.7 highpass filtering

- PQ=paddedsize(size(f));
- D0=0.05*PQ(1);
- H=hpfilter('gaussian', PQ(1), PQ(2), D0);
- g=dftfilt(f, H);
- imshow(g, [ ])



a b

FIGURE 4.18
(a) Original image.
(b) Result of Gaussian highpass filtering.

4.6.2 High frequency emphasis filtering

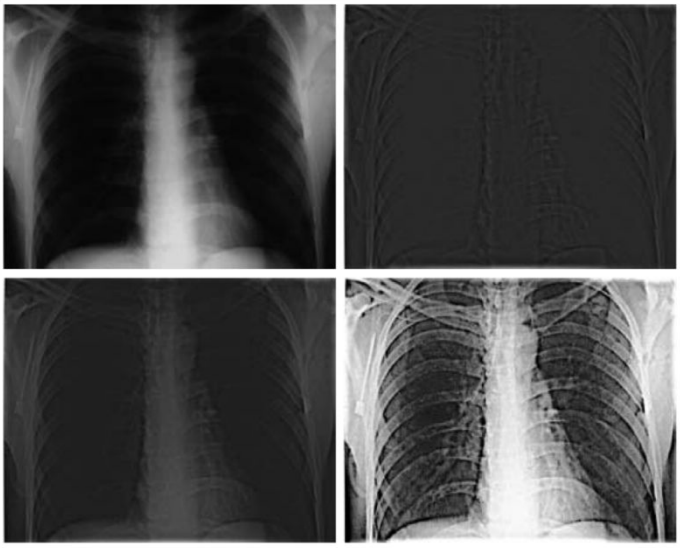
Hhfe(u,v)=a+b*Hhp(u,v)

where a is the offset, b is the multiplier,and Hhp is the transfer function of a highpass filter.

Example 4.8

Combining high frequency emphasis and histogram equalization.

- PQ=paddedsize(size(f));
- D0=0.05*PQ(1);
- HBW=hpfilter('btw', PQ(1), PQ(2), D0, 2)
- gbw=dftfilt(f, HBW);
- ghw=gscale(ghw);
- H=0.5+2*HBW;
- ghf=dftfilt(f, H);
- ghf=gscale(ghf);
- ghe=histeq(ghf, 256)



a b
c d

FIGURE 4.19 High-frequency emphasis filtering. (a) Original image. (b) Highpass filtering result. (c) High-frequency emphasis result. (d) Image (c) after histogram equalization. (Original image courtesy of Dr. Thomas R. Gest, Division of Anatomical Sciences, University of Michigan Medical School.)

Summary

In addition to the image enhancement applications in this and the preceding chapter, the concepts and techniques developed in these two chapters provide the basis for other areas of image processing addressed in subsequent discussions in the book.

Intensity transformations are used frequently for intensity scaling, and spatial filtering is used extensively for image restoration in the next chapter, for color processing in Chapter 6, for image segmentation in Chapter 10, for extracting descriptors from an image in Chapter 11.

The Fourier techniques developed in the chapter are used extensively in the next chapter for image restoration,

- Chapter 8 for image compression,
- Chapter 11 for image description.

Chapter5 Image Restoration

1. Objective of restoration

_improve a given image

_objective process

2. Compare enhancement with it

_ subjective process

Restoration attempts to reconstruct or recover an image that has been degrade by using a priori knowledge of the degradation phenomenon. Restoration techniques are oriented toward modeling the degradation and applying the inverse process in order to recover the original image.

3. Explore how to use MATLAB and IPT capabilities to model degradation phenomena and to formulate restoration solutions.

5.1 A Model of the Image Degradation/Restoration Process

1. The degradation process is modeled

$$g(x,y)=H[f(x,y)]+n(x,y)$$

g(x,y) output image,f(x,y) input image,n(x,y) noise,H the degradation system

2. If H is a linear,spatially invariant process,it can be given in spatial domain g(x,y)=h(x,y)*f(x,y)+n(x,y)
3. In frequency domain G(u,v)=H(u,v)F(u,v)+N(u,v)

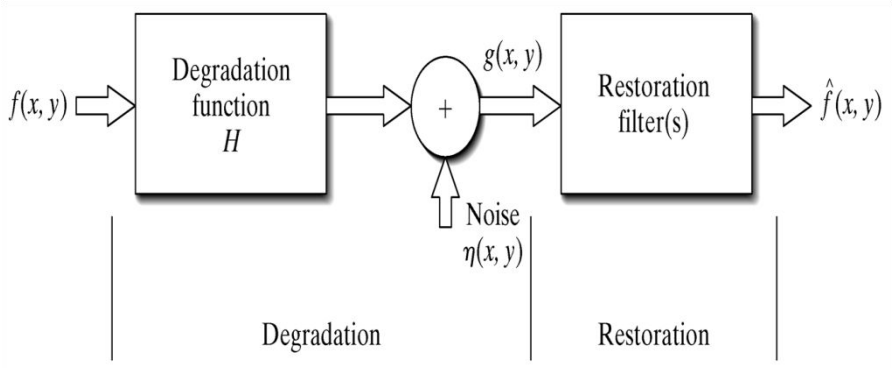


FIGURE 5.1
A model of the image degradation/ restoration process.

The degradation function H(u,v) is called the *optical transfer function* (OTF) from the Fourier analysis of optical systems.

In spatial domain, h(x,y) is the *point spread function* (PSF) from a point of light to obtain the characteristics of the degradation for any type of input.

The OTF and PSF are a Fourier transform pair.

5.2 Noise Models

Two basic types of noise models: noise in the spatial domain and noise in the frequency domain, described by various Fourier properties of the noise.

5.2.1 Adding Noise with Function imnoise

g=imnoise(f, type, parameters)

5.2.2 Generating Spatial Random Noise with a Specified Distribution PDF,CDF

Example 5.1: Using uniform random numbers to generate random numbers with a specified distribution.

>>R=a+sqrt(b*log(1-rand(M, N)));

Table 5.1 lists the random variables of interest in the present discussion, along with their PDFs, CDFs, and random number generator equations.

TABLE 5.1 Generation of random variables.

Name	PDF	Mean and Variance	CDF	Generator†
Uniform	$p_z(z) = \begin{cases} \frac{1}{b-a} & \text{if } a \le z \le b \\ 0 & \text{otherwise} \end{cases}$	$m = \frac{a+b}{2}, \quad \sigma^2 = \frac{(b-a)^2}{12}$	$F_z(z) = \begin{cases} 0 & z < a \\ \frac{z-a}{b-a} & a \le z \le b \\ 1 & z > b \end{cases}$	MATLAB function `rand`
Gaussian	$p_z(z) = \frac{1}{\sqrt{2\pi}b} e^{-(z-a)^2/2b^2}$ $-\infty < z < \infty$	$m = a, \quad \sigma^2 = b^2$	$F_z(z) = \int_{-\infty}^{z} p_z(v)\, dv$	MATLAB function `randn`
Salt & Pepper	$p_z(z) = \begin{cases} P_a & \text{for } z = a \\ P_b & \text{for } z = b \\ 0 & \text{otherwise} \end{cases}$ $b > a$	$m = aP_a + bP_b$ $\sigma^2 = (a-m)^2 P_a + (b-m)^2 P_b$	$F_z(z) = \begin{cases} 0 & \text{for } z < a \\ P_a & \text{for } a \le z < b \\ P_a + P_b & \text{for } b \le z \end{cases}$	MATLAB function `rand` with some additional logic
Lognormal	$p_z(z) = \frac{1}{\sqrt{2\pi}bz} e^{-[\ln(z)-a]^2/2b^2}$ $z > 0$	$m = e^{a+(b^2/2)}, \sigma^2 = [e^{b^2} - 1]e^{2a+b^2}$	$F_z(z) = \int_0^z p_z(v)\, dv$	$z = ae^{bN(0,1)}$
Rayleigh	$p_z(z) = \begin{cases} \frac{2}{b}(z-a)e^{-(z-a)^2/b} & z \ge a \\ 0 & z < a \end{cases}$	$m = a + \sqrt{\pi b/4}, \sigma^2 = \frac{b(4-\pi)}{4}$	$F_z(z) = \begin{cases} 1 - e^{-(z-a)^2/b} & z \ge a \\ 0 & z < a \end{cases}$	$z = a + \sqrt{b \ln[1 - U(0,1)]}$
Exponential	$p_z(z) = \begin{cases} ae^{-az} & z \ge 0 \\ 0 & z < 0 \end{cases}$	$m = \frac{1}{a}, \quad \sigma^2 = \frac{1}{a^2}$	$F_z(z) = \begin{cases} 1 - e^{-az} & z \ge 0 \\ 0 & z < 0 \end{cases}$	$z = -\frac{1}{a} \ln[1 - U(0,1)]$
Erlang	$p_z(z) = \frac{a^b z^{b-1}}{(b-1)!} e^{-az}$ $z \ge 0$	$m = \frac{b}{a}, \quad \sigma^2 = \frac{b}{a^2}$	$F_z(z) = \left[1 - e^{-az} \sum_{n=0}^{b-1} \frac{(az)^n}{n!}\right]$ $z \ge 0$	$z = E_1 + E_2 + \cdots + E_b$ (The E's are exponential random numbers with parameter a.)

† $N(0, 1)$ denotes normal (Gaussian) random numbers with mean 0 and a variance of 1. $U(0, 1)$ denotes uniform random numbers in the range $(0, 1)$.

A=randn(M, N)

I = find(A) % The syntax forms of function imnoise2

[r,c] = find(A)

[r, c, v] = find(A)

>> I = find(A < 128); % To find and set to 0 all pixels in an image

>> A(I) = 0; % whose values are less than 128

>> I = find(A >=64 & A <= 192); % To set to 128 all pixels in the closed

>> A(I) = 128; % interval [64, 192]

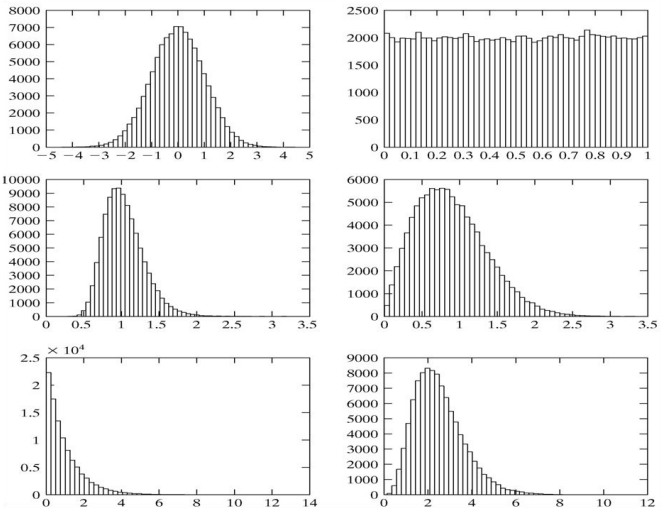
function R = imnoise2(type, M, N, a, b)

(1). generates an M*N noise array, (2). produces the noise pattern itself compared with imnoise(outputs a noise image)

Example 5.2: Histograms of data generated using the function imnoise2.

p=hist(r,bins)

Figure 5.2 shows histograms of all the random number types in Table 5.1.



a b
c d
e f

FIGURE 5.2 Histograms of random numbers: (a) Gaussian, (b) uniform, (c) lognormal, (d) Rayleigh, (e) exponential, and (f) Erlang. In each case the default parameters listed in the explanation of function `imnoise2` were used.

>> r= imnoise2(‘gaussian’, 100000, 1, 0, 1);% to generate the data for Fig.5.2(a)

p = hist(r, bins) % bins = 50 to generate the histograms in Fig.5.2.

5.2.3 Periodic Noise

r(x,y)=Asin[2pi*u0(x+Bx)/M+2pi*V0(y+By)/N

R(u,v)=jA/2[exp(j2pi*u0*Bx/M)delter(u+u0,v+v0)

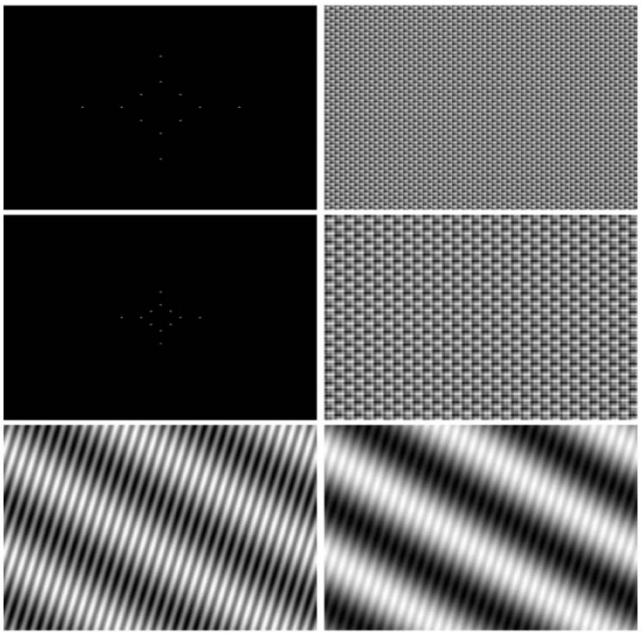
-exp(j2pi*v0*By/N)delter(u-u0,v-v0)]

function [r, R, S] = imnoise3(M, N, C, A, B)

C is a k-by-2 matrix containing k pairs of frequency domain coordinates(u,v) indicating the locations of impulses in the frequency domain.

Example 5.3: Using function *imnoise3*

```
>> C = [0 64; 0 128;    32 32; 64,0; 128 0; -32 32];
>> [r, R, S] = imnoise3(512, 512, C);
>>imshow(S, [ ])                    % Fig. 5.3(a)
>> figure,    imshow(r, [ ])         % Fig.5.3(b)
>>    C = [0 32; 0 64; 16 16; 32 0; 64 0; -16 16] ;      % Fig.5.3(c,d)
>>    C = [6 32; -2 2] ;              %      Fig.5.3(e)
>>    A = [1    5];
>>    [r, R, S] = imnoise3( 512, 512, C, A);
```



a b
c d
e f

FIGURE 5.3
(a) Spectrum of specified impulses. (b) Corresponding sine noise pattern. (c) and (d) A similar sequence. (e) and (f) Two other noise patterns. The dots in (a) and (c) were enlarged to make them easier to see.

5.2.4 Estimating Noise Parameters

Function statmoments computes the mean and central moments up to order n, and returns them in row vector v,

v(1)=mean,v(2)=variance.

[v, unv] = statmoments(p, n) % p is the histogram vector and n is the number of moments to compute.

To select a region of interest(ROI), using function roipoly

B = roipoly(f, c, r)

% f is the image of interest, c and r are the column and row coordinates of the vertices of the polygon, B is a binary image the same size as f with 0's outside the region of interest and 1's inside. B = roipoly(f) displays the image f on the screen and lets the user specify the polygon using the mouse.

[B, c, r] = roipoly(. . .) % indicates any valid syntax for this function

Example 5.4: Estimating noise parameters.

>> [B, c, r] = roipoly(f); % Fig 5.4(b)

function [p,npix] = histroi(f, c, r) computes the histogram of an image within a polygonal region . P is histogram, npix is the number of pixels in the polygonal region. C and r are the coordinates of polygon points.

>> [P, npix] = histroi(f, c, r); % Fig 5.4(c)

>> Figure, Bar(p, 1)

>> [V , unv] = statmoments(h, 2); % The Mean and Variance of the region masked by B

>> V =

0.5794 0.0063

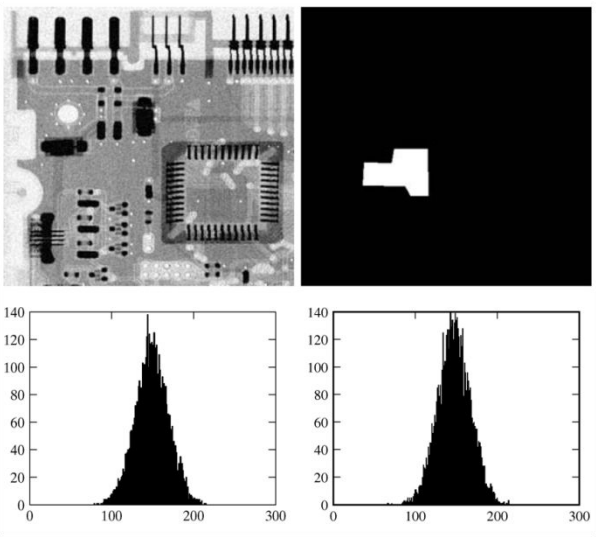
>> unv

147.7430 410.9313

>> x= imnoise2('gaussian', npix, 1, 147, 20);

>> figure, hist(x, 130)

>> axis([0 300 0 140])



a b
c d

FIGURE 5.4
(a) Noisy image.
(b) ROI generated interactively.
(c) Histogram of ROI.
(d) Histogram of Gaussian data generated using function `imnoise2`.
(Original image courtesy of Lixi, Inc.)

5.3 Restoration in the Presence of Noise Only—Spatial Filtering

When the only degradation present is noise, the model

$$g(x,y) = f(x,y) + n(x,y)$$

In this section we summarize and implement several spatial filers for noise reduction.

5.3.1 Spatial Noise Filters

- B = imlincomb(c1, A1, c2, A2, . . . , ck, Ak)
- % function *imlincomb* to compute the linear combination of the inputs.
- Which implements the equation

  B = c1*A1 + c2*A2 + . . . + ck*Ak
- warning('message')
- % To be suppressed if the argument of the *log* function becomes 0.

TABLE 5.2 Spatial filters. The variables m and n denote respectively the number of rows and columns of the filter neighborhood.

Filter Name	Equation	Comments
Arithmetic mean	$\hat{f}(x,y) = \frac{1}{mn}\sum_{(s,t)\in S_{xy}} g(s,t)$	Implemented using IPT functions `w = fspecial('average', [m, n])` and `f = imfilter(g, w)`.
Geometric mean	$\hat{f}(x,y) = \left[\prod_{(s,t)\in S_{xy}} g(s,t)\right]^{\frac{1}{mn}}$	This nonlinear filter is implemented using function `gmean` (see custom function `spfilt` in this section).
Harmonic mean	$\hat{f}(x,y) = \frac{mn}{\sum_{(s,t)\in S_{xy}} \frac{1}{g(s,t)}}$	This nonlinear filter is implemented using function `harmean` (see custom function `spfilt` in this section).
Contraharmonic mean	$\hat{f}(x,y) = \frac{\sum_{(s,t)\in S_{xy}} g(s,t)^{Q+1}}{\sum_{(s,t)\in S_{xy}} g(s,t)^{Q}}$	This nonlinear filter is implemented using function `charmean` (see custom function `spfilt` in this section).
Median	$\hat{f}(x,y) = \underset{(s,t)\in S_{xy}}{\text{median}}\{g(s,t)\}$	Implemented using IPT function `medfilt2`: `f = medfilt2(g, [m n])`.
Max	$\hat{f}(x,y) = \max_{(s,t)\in S_{xy}}\{g(s,t)\}$	Implemented using IPT function `ordfilt2`: `f = ordfilt2(g, m*n, ones(m, n))`.
Min	$\hat{f}(x,y) = \min_{(s,t)\in S_{xy}}\{g(s,t)\}$	Implemented using IPT function `ordfilt2`: `f = ordfilt2(g, 1, ones(m, n))`.
Midpoint	$\hat{f}(x,y) = \frac{1}{2}\left[\max_{(s,t)\in S_{xy}}\{g(s,t)\} + \min_{(s,t)\in S_{xy}}\{g(s,t)\}\right]$	Implemented as 0.5 times the sum of the max and min filtering operations.
Alpha-trimmed mean	$\hat{f}(x,y) = \frac{1}{mn-d}\sum_{(s,t)\in S_{xy}} g_r(s,t)$	The $d/2$ lowest and $d/2$ highest intensity levels of $g(s,t)$ in S_{xy} are deleted, $g_r(s,t)$ denotes the remaining $mn-d$ pixels in the neighborhood. Implemented using function `alphatrim` (see custom function `spfilt` in this section).

Function f = spfilt(g, type, m, n, parameter) (P-159)

% spfilt performs filtering in the spatial domain with any of the filters

% listed in Table 5.2.

Example 5.5: Using function *spfilt*.

>> [M, N] = size(f); % To generate Fig.5.5(a)

>> R = imnoise2(‘salt & pepper’, M, N, 0.1,0);

>> c = find(R = = 0);

>> gp = f;

>> gp(c) = 0;

>> R = imnoise2(‘salt & pepper’, M, N, 0, 0.1); % To generate Fig.5.5(b)

>> c = find(R = = 1);

>> gs = f;

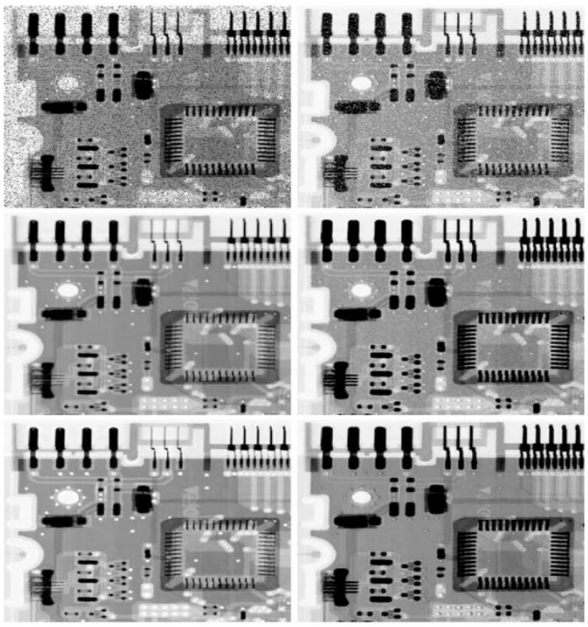
>> gs(c) = 255;

>> fp = spfilt(gp, ‘chmean’, 3, 3, 1.5); % To generate Fig.5.5(c) To filter pepper noise

>> fs = spfilt(gs, ‘chmean’, 3, 3, -1.5); % To filter salt noise

>> fpmax = spfilt(gp, ‘max’, 3, 3);

>> fsmin = spfilt(gs, ‘ min’, 3, 3); % To generate Figs.5.5 (e) and (f).



a b
c d
e f

FIGURE 5.5
(a) Image corrupted by pepper noise with probability 0.1. (b) Image corrupted by salt noise with the same probability. (c) Result of filtering (a) with a 3×3 contraharmonic filter of order $Q = 1.5$. (d) Result of filtering (b) with $Q = -1.5$. (e) Result of filtering (a) with a 3×3 max filter. (f) Result of filtering (b) with a 3×3 min filter.

5.3.2 Adaptive Spatial Filters

In some applications, results can be improved by using filters capable of adapting their behavior depending on the characteristics of the image in the area being filtered.

As an illustration, adaptive median filter and median filtering algorithm

Zmin Zmax Zmed Zxy, Level A, Level B

- Matlab function

f = adpmedian(g, Smax)

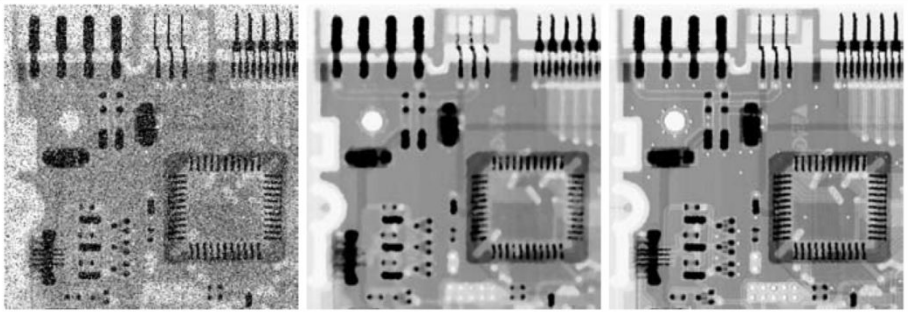
- % Smax is the maximum allowed size of the
- adaptive filter window.

Example 5.6: Adaptive median filtering.

>> g = imnoise(f, 'salt & pepper', .25);

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

>> f2 = adpmedian(g, 7);



a b c

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

5.4 Periodic Noise Reduction by Frequency Domain Filtering

H(u,v)= once H has been obtained, filtering is done using function dftfilt explained in section 4.3.3

5.5 Modeling the Degradation Function

PSF point spread function, a term that arises from letting h(x,y) operate on a point of light to obtain the characteristics of the degradation for any type of input.

When no information is available about the PSF, we can resort to "blind deconvolution" for inferring the PSF.

One of the principal degradations encountered in image restoration problems is image blur.

Blur that occurs with the scene and sensor at rest with respect to each other can be modeled by spatial or frequency domain lowpass filters.

Another important degradation model is image blur due to uniform linear motion between the sensor and scene during image acquisition

Image blur can be modeled using IPT function fspecial

PSF = fspecial('motion', len, theta)

Len pixels, theta is in degrees.

To create a degraded image with a PSF

```
>> g = imfilter(f, PSF, 'circular');
>> g = g + noise;
```

It is useful to use the same image or test pattern so that comparisons are meaningful. The test pattern generated by

```
C = checkerboard( NP, M, N)
```

To generate a checkerboard in which all light squares are white we use

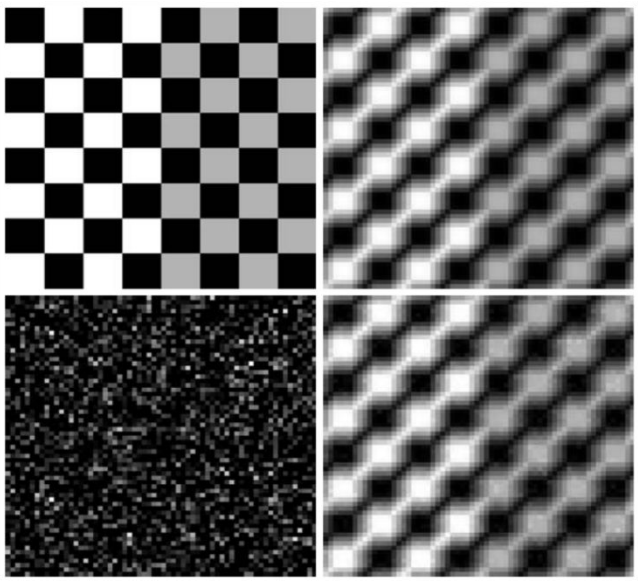
```
>> K = im2double( checkerboard( NP, M, N)) > 0.5;
```

To zoom an image by pixel replication(Appendix C for the code)

```
B = pixeldup(A, m, n)
```

Example 5.7: Modeling a blurred, noisy image.

```
>> f = checkerboard (8);                          % Figure 5.7(a)
>> PSF = fspecial('motion', 7, 45);          % Fig.5.7(b)
>> gb = imfilter(f, PSF, 'circular');
>>  noise = imnoise(zeros(size(f)), 'gaussian', 0, 0.001);   %  Fig.5.7(c)
>>  g = gb + noise;                                % Fig. 5.7(d)
>>  imshow(pixeldup(f, 8), [ ])
```



a b
c d

FIGURE 5.7
(a) Original image. (b) Image blurred using `fspecial` with `len = 7`, and `theta = –45` degrees. (c) Noise image. (d) Sum of (b) and (c).

5.6 Direct Inverse Filtering

degradation process is modeled

$$g(x,y)=H[f(x,y)]+n(x,y)$$

In frequency domain

$$G(u,v)=H(u,v)F(u,v)+N(u,v)$$

An estimate of the form

$$F(u,v)=G(u,v)/H(u,v)$$

Or

$$F(u,v)=F(u,v)+N(u,v)/H(u,v)$$

5.7 Wiener Filtering

Wiener filter seeks an estimate that minimizes the statistical error function

$$e^2 = E\{(f-\hat{f})^2\}$$

In frequency domain

$$\hat{F}(u,v) = [\frac{1}{H(u,v)} \frac{|H(u,v)|^2}{|H(u,v)|^2 + S_\eta(u,v)/S_f(u,v)}]G(u,v)$$

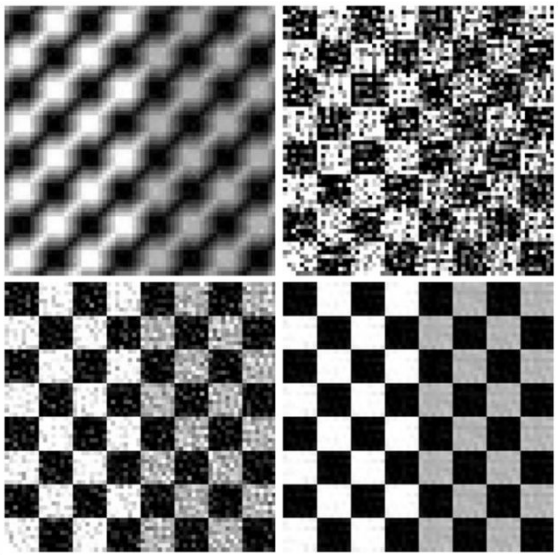
Defined R, replacing it by assumes that is constant. P-171

1. fr = deconvwnr(g, PSF) % The noise-to-signal ratio is zero.
2. fr = deconvwnr(g, PSF, NSPR) % The noise to-signal ratio is known.
3. fr = deconvwnr(g, PSF, NACORR, FACORR)

Before using them, use the J = edgetaper(I, PSF) **reduce ringing by FFT**

Example 5.8: Using function deconvwnr to restore a blurred, noisy image.

```
>> fr1 = deconvwnr(g, PSF);                    % Fig.5.7(b)
>> Sn = abs(fft2(noise)) .^2;                  % noise power spectrum
>> nA = sum(Sn(:))/prod(size(noise));          % noise average power
>> Sf = abs(fft2(f)) .^2;                      % image power spectrum
>> fA = sum(Sf(:))/prod(size(f));              % image average power
>> R = nA/fA                                   % The ratio, R
>> fr2 = deconvwnr(g, PSF, R);        % To restore the image using this ratio
>> NCORR = fftshift(real(ifft2(Sn)));          % Restoration using the
>> ICORR = fftshift(real(ifft2(Sf)));          % autocorrelation functions
>> fr3 = deconvwnr(g, PSF, NCORR, ICORR);
```



a b
c d

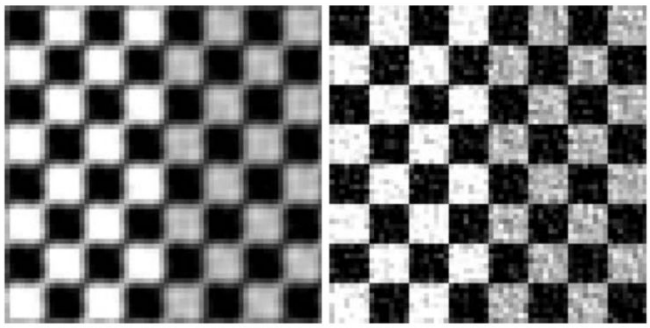
FIGURE 5.8
(a) Blurred, noisy image. (b) Result of inverse filtering. (c) Result of Wiener filtering using a constant ratio. (d) Result of Wiener filtering using autocorrelation functions.

5.8 Constrained Least Squares (Regularized) Filtering

fr = deconvreg(g, PSF, NOISEPOWER, RANGE)

Example 5.9: Using function *deconvreg* to restore a blurred, noisy image.

```
>> fr = deconvreg(g,  PSF,  4);     %  Fig.5.9(a)
>> fr = deconvreg(g,  PSF,  0.4,  [1e-7  1e7]);     %  Fig.5.9(b)
```



a b

FIGURE 5.9
(a) The image in Fig. 5.7(d) restored using a regularized filter with `NOISEPOWER` equal to 4. (b) The same image restored with `NOISEPOWER` equal to 0.4 and a `RANGE` of `[1e–7 1e7]`.

5.9 Iterative Nonlinear Restoration Using the Lucy-Richardson Algorithm

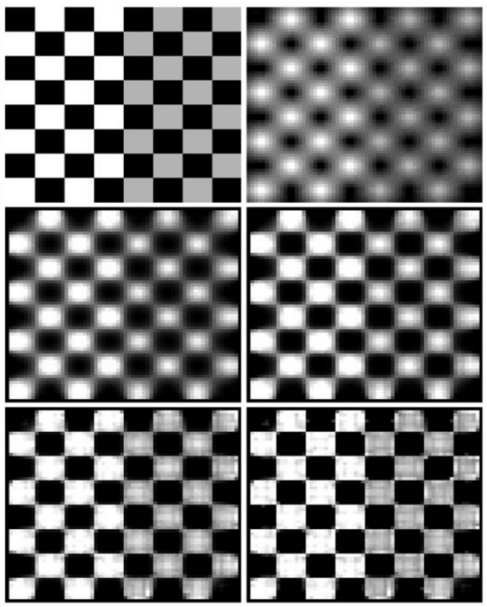
L_R algorithm arises from a maximum-likelihood formulation in which the image is modeled with Poisson statistics. The Model and following iteration converges:

$$\hat{f}_{k+1}(x,y) = \hat{f}_k(x,y)[h(-x,-y) * \frac{g(x,y)}{h(x,y) * \hat{f}_k(x,y)}]$$

fr = deconvlucy(g, PSF, NUMIT, DAMPAR, WEIGHT)

Example 5.10: Using function deconvlucy to restore a blurred, noisy image.

```
>> f = checkerboard(8);
>> imshow(pixeldup(f,    8));
>> PSF = fspecial('gaussian', 7, 10);
>> SD = 0.01;
>> g = imnoise( imfilter(f, PSF), 'gaussian', 0, SD^2);        % Fig 5.10(b)
>> DAMPAR = 10*SD;
>> LIM = ceil(size(PSF, 1)/2);                    % To create array WEIGHT
>> WEIGHT = zeros(size(g));
>> WEIGHT(LIM + 1:end – LIM,      LIM +1:end –LIM) = 1;
>> NUMIT = 5;
>> fr = deconvlucy(g, PSF, NUMIT, DAMPAR, WEIGHT);
>> imshow(pixeldup(fr, 8))                          %    Fig. 5.10(c)
```



a b
c d
e f

FIGURE 5.10
(a) Original image. (b) Image blurred and corrupted by Gaussian noise. (c) through (f) Image (b) restored using the L-R algorithm with 5, 10, 20, and 100 iterations, respectively.

5.10 Blind Deconvolution

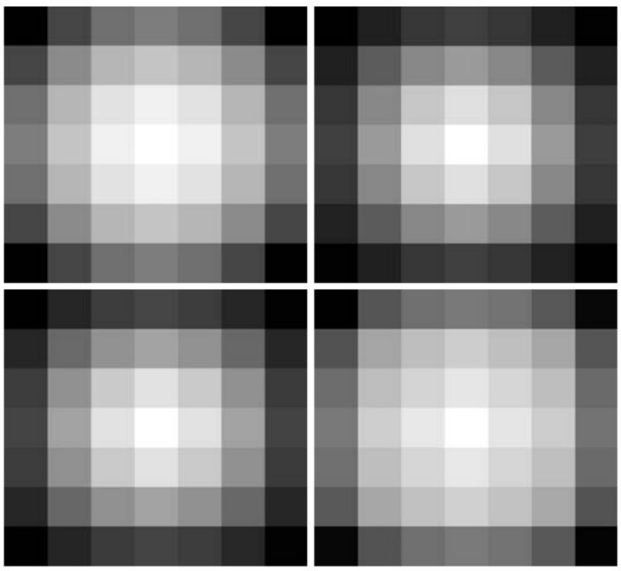
One of most difficult problems in image restoration is obtaining a suitable estimate of the PSF to use in restoration algorithms. Image restoration methods that are not based on specific knowledge of the PSF are called blind deconvolution.

In blind deconvolution the optimization problem is solved iteratively with specified constraints and, assuming convergence, the specific f(x,y) and h(x,y) [section 5.1] that result in a maximum are the restored image and the PSF.

1. [fr, PSFe] = deconvblind(g, INITPSF)
2. [fr, PSFe] = deconvblind(g, INITPSF, NUMIT, DAMPAR, WEIGHT)

Example 5.11: Using function *deconvblind* to estimate a PSF.

```
>> PSF = fspecial('gaussian', 7, 10);
>> imshow(pixeldup(PSF, 73), [ ]);
>> SD = 0.01;
>> g = imnoise(imfilter(f, PSF), 'gaussian', 0, SD^2);
>> INITPSF = ones(size(PSF));
>> NUMIT = 5;
>> [fr, PSFe] = deconvblind(g, INITPSF, NUMIT, DAMPAR, WEIFHT);
>> imshow(pixeldup(PSFe, 73), [ ]);
```



a b
c d

FIGURE 5.11
(a) Original PSF. (b) through (d) Estimates of the PSF using 5, 10, and 20 iterations in function `deconvblind`.

5.11 Geometric Transformations and Image Registration

An introduction to geometric transformations for image restoration. Geometric transformations modify the spatial relationship between pixels in an image. Geometric transformations are used frequently to perform image registration, a process that takes two images of the same scene and aligns them so they can be merged for visualization, or for quantitative comparison.

We discuss

(1). Spatial transformations and how to define and visualize them in MATLAB;

(2). How to apply spatial transformations to images;

(3). How to determine spatial transformations for use in image registration.

5.11.1 Geometric Spatial Transformations

(x, y) = T{(w,z)}

For example, if (x, y) = T{(w, v)} = (w/2, z/2), the "distortion" is simply a shrinking of *f* by half in both spatial dimensions, as illustrated in Fig.5.12.

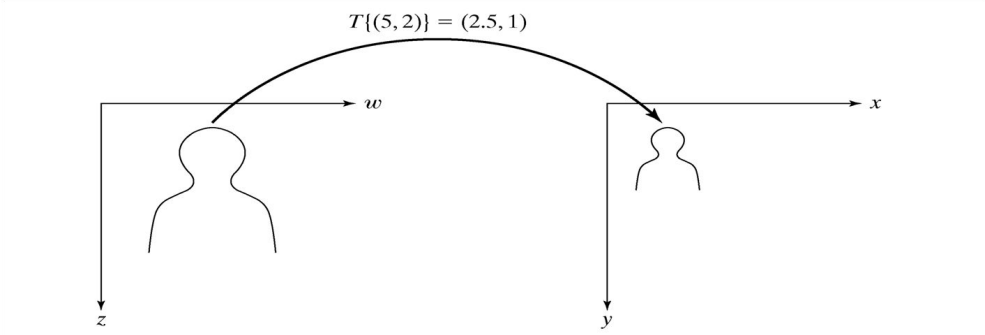


FIGURE 5.12 A simple spatial transformation. (Note that the *xy*-axes in this figure do not correspond to the image axis coordinate system defined in Section 2.1.1. As mentioned in that section, IPT on occasion uses the so-called spatial coordinate system in which *y* designates rows and *x* designates columns. This is the system used throughout this section in order to be consistent with IPT documentation on the topic of geometric transformations.)

One of the most commonly used forms of spatial transformations is the affine transform

$$[x \quad y \quad 1] = [w \quad z \quad 1]\mathbf{T} = [w \quad z \quad 1]\begin{bmatrix} t_{11} & t_{12} & 0 \\ t_{21} & t_{22} & 0 \\ t_{31} & t_{32} & 1 \end{bmatrix}$$

This transformation can scale, rotate, translate, or shear a set of points, depending on the values chosen for the elements of T.

Table 5.3 shows how to choose the values of the elements to achieve different transformations.

One way to create such a structure is by using function maketform

tform = maketform(transform_type, transform_parameters)

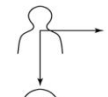
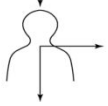
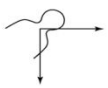
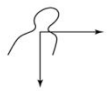
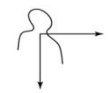
Type	Affine Matrix, T	Coordinate Equations	Diagram
Identity	$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$	$x = w$ $y = z$	
Scaling	$\begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{bmatrix}$	$x = s_x w$ $y = s_y z$	
Rotation	$\begin{bmatrix} \cos\theta & \sin\theta & 0 \\ -\sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}$	$x = w\cos\theta - z\sin\theta$ $y = w\sin\theta + z\cos\theta$	
Shear (horizontal)	$\begin{bmatrix} 1 & 0 & 0 \\ \alpha & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$	$x = w + \alpha z$ $y = z$	
Shear (vertical)	$\begin{bmatrix} 1 & \beta & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$	$x = w$ $y = \beta w + z$	
Translation	$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ \delta_x & \delta_y & 1 \end{bmatrix}$	$x = w + \delta_x$ $y = z + \delta_y$	

TABLE 5.3
Types of affine transformations.

IPT provides two functions for applying a spatial transformation to points:

Tformfwd computes the forward transformation T{(w,z)}

Tforminv computes the inverse transformation 1/T{(x,y)}

Example

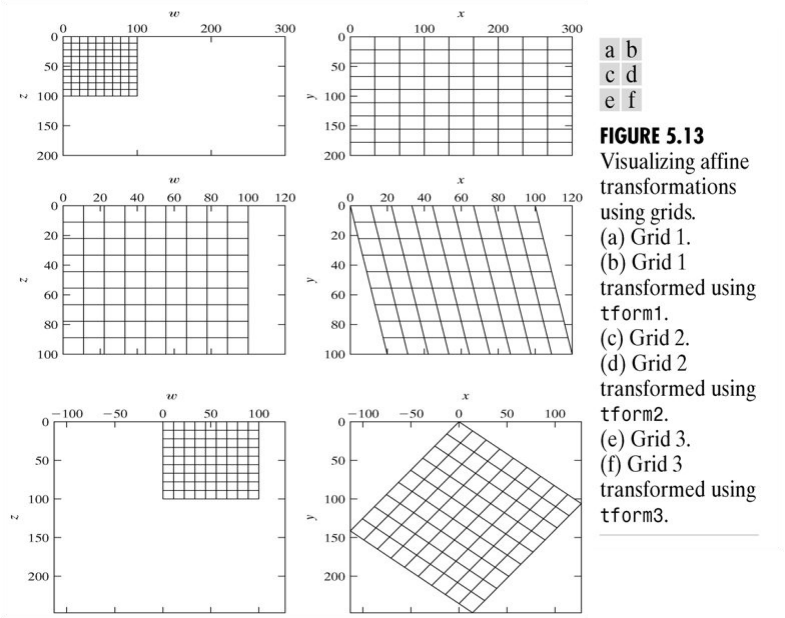
Vistformfwd constructs a grid of points, transforms the grid using tformfwd, and then plots the grid and the transformed grid side by side for comparison.

Function vistformfwd(tform, wdata, zdata, N)

Example 5.12: Visualizing affine transforms using v*istformfwd*.

```
>> T1 = [3    0    0; 0    2    0; 0    0    1]
>> tform1 = maketform('affine', T1);
>> vistformfwd(tform1,[0 100],[0 100]);% Fig.5.13(a) and (b) show the result.
>> T2 = [1    0    0;    .2    1    0;    0    0    1];
>>    tform2 = maketform('affine', T2);
>>    vistformfwd(tform2,    [0      100],    [0    100]);
          % Fig.5.13(c) and (d) show the effect of theshearing transform on a grid.
>> Tscale = [1.5    0    0;    0    2    0;    0    0    1];
>> Trotation = [cos(pi/4)    sin(pi/4)    0    -sin(pi/4)    cos(pi/4)    0 0      0      1];
>> Tshear = [1    0    0;    .2    1    0;    0    0    1];
>> T3 = Tscale * Trotation * Tshear;
>> tform3 = maketform('affine', T3);
>>vistformfwd(tform3, [0     100], … [0     100]) % Fig.5.13(e) and (f) show the
```

results.



a b
c d
e f

FIGURE 5.13 Visualizing affine transformations using grids. (a) Grid 1. (b) Grid 1 transformed using `tform1`. (c) Grid 2. (d) Grid 2 transformed using `tform2`. (e) Grid 3. (f) Grid 3 transformed using `tform3`.

5.11.2 Applying Spatial Transformations to Images

Most computational methods for spatially transforming an image fall into one of two categories:

Methods that use forward mapping

Methods that use inverse mapping

One problem with the forward mapping procedure is that two or more different pixels in the input image could be transformed into the same pixel in the output image, raising the question of how to combine multiple input pixel values into a single output pixel value.

Another potential problem is that some output pixels may not be assigned a value at all.

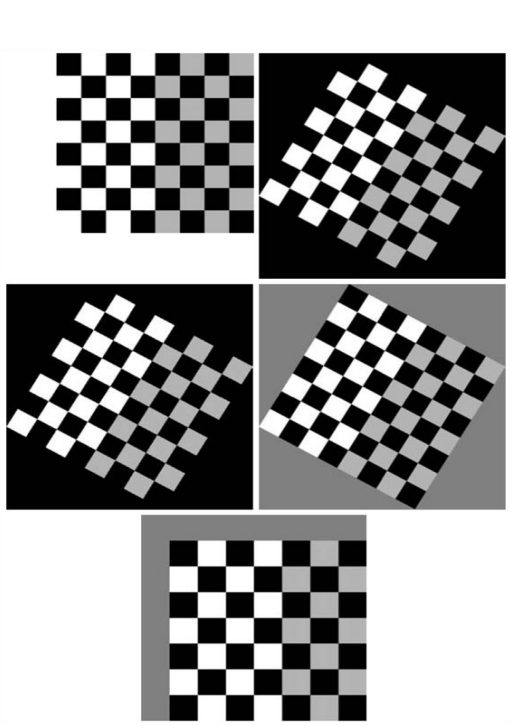
IPT USE g = imtransform(f, tform, interp) for inverse mapping instead.

A linear conformal transformation is a type of affine transformation that preserves shapes and angles.

Example 5.13: Spatially transforming images.

```
>> f = checkerboard(50);
>> s = 0.8;
>> theta = pi/6;
>> T = [s*cos(theta)    s*sin(theta)    0
```

```
        -s*sin(theta)    s*cos(theta)    0
          0                  0             1];
>> tform = maketform('affine',    T);
>> g = imtransform(f,      tform);
>> g2 = imtransform(f,    tform,    'nearest');
>> g3 = imtransform(f,    tform, 'FillValue', 0.5);
>>T2 = [1    0    0;    0    1    0; 50    50    1];
>> tform2 = maketform('affine,…T2);
>> g4 = imtransform(f, tform2);
>> g5 = imtransform(f, tform2,…
'Xdata', [1    400], 'Ydata', …          [1    400], 'FillValue', 0.5);
```



a b
c d
e

FIGURE 5.14 Affine transformations of the checkerboard image. (a) Original image. (b) Linear conformal transformation using the default interpolation (bilinear). (c) Using nearest neighbor interpolation. (d) Specifying an alternate fill value. (e) Controlling the output space location so that translation is visible.

5.11.3 Image Registration

Image registration methods seek to align two images of the same scene, the figure illustrates the idea of control points using a test pattern and a version of the test pattern that has undergone projective distortion.

IPT function cp2tform can be used to fit a specified type of spatial transformation to the control points (support the transformation types in Table 5.4.

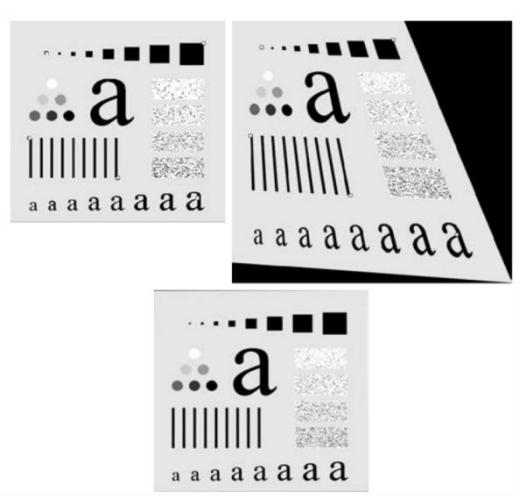
Example

>> basepoints =[83 81; 450 56; 43 293; 249 392; 436 442];

>> inputpoints = [68 66; 375 47; 42 286; 275 434; 523 532];

>> tform = cp2tform(inputpoints, basepoints, ‘projective’);

>> gp = imtransform(g, tform, ‘Xdata”, [1 502], ‘Ydata’, [1 502]);



a b
c

FIGURE 5.15
Image registration based on control points.
(a) Original image with control points (the small circles superimposed on the image).
(b) Geometrically distorted image with control points.
(c) Corrected image using a projective transformation inferred from the control points.

5- 16

Transformation Type	Description	Functions
Affine	Combination of scaling, rotation, shearing, and translation. Straight lines remain straight and parallel lines remain parallel.	`maketform` `cp2tform`
Box	Independent scaling and translation along each dimension; a subset of affine.	`maketform`
Composite	A collection of spatial transformations that are applied sequentially.	`maketform`
Custom	User-defined spatial transform; user provides functions that define T and T^{-1}.	`maketform`
Linear conformal	Scaling (same in all dimensions), rotation, and translation; a subset of affine.	`cp2tform`
LWM	Local weighted mean; a locally-varying spatial transformation.	`cp2tform`
Piecewise linear	Locally varying spatial transformation.	`cp2tform`
Polynomial	Input spatial coordinates are a polynomial function of output spatial coordinates.	`cp2tform`
Projective	As with the affine transformation, straight lines remain straight, but parallel lines converge toward vanishing points.	`maketform` `cp2tform`

TABLE 5.4
Transformation types supported by `cp2tform` and `maketform`.

5- 17

The toolbox includes a graphical user interface designed for the interactive selection of control points on a pair of images.

Figure 5.16 shows a screen capture of this tool, which is invoked by the command cpselect.

FIGURE 5.16
Interactive tool for choosing control points.

Summary

A good overview of how MATLAB and IPT functions used for image restoration and as the basis for generating models that help explain the degradation to which an image has been subjected.

- imnoise, imnoise2, imnoise3
- spfilt for denoising
- Adaptive and blind filter for (PSF and noise)
- Geometric transformations and image registration

Chapter6 Color Image Processing

Preview

In this chapter

we discuss fundamentals of color image processing using the image processing toolbox ,

extend some of its functionality by developing additional color generation and transformation functions.

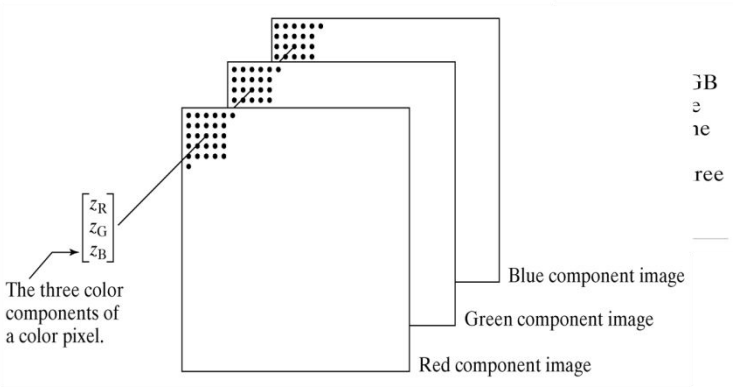
The discussion in this chapter assumes familiarity on the part of the reader with the principles and terminology of color image processing at an introductory level.

6.1 Color Image Representation in MATLAB

Two types: indexed images or RGB(red, green, blue) images

6.1.1 RGB Images

an M×N×3 array of color pixels as a ”stack” of three gray-scale images



6- 1

Let fR, fG, fB represent three RGB component images. An RGB image is formed from these images by using the cat(concatenate) operator to stack the image.

rgb_image = cat(3 , fR , fG , fB) % fR,fG,fB represent three

RGB component images

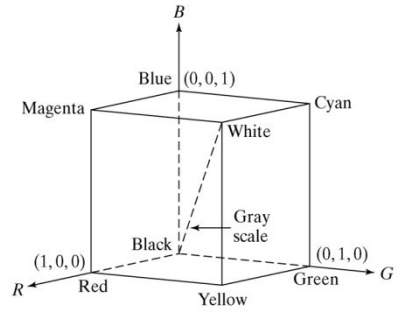
The following commands extract the three component image:

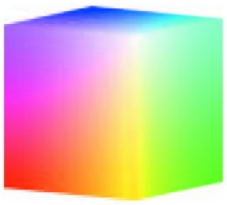
fR = rgb_image(: , : , 1);

fG = rgb_image(: , : , 2);

fB = rgb_image(: , : , 3);

The color space usually is shown graphically as an RGB color cube:





a b

FIGURE 6.2
(a) Schematic of the RGB color cube showing the primary and secondary colors of light at the vertices. Points along the main diagonal have gray values from black at the origin to white at point (1, 1, 1). (b) The RGB color cube.

Function rgbcube (vx,vy,vz) is used to view the

color cube from any perspective.

```
function rgbcube(vx,vy,vz)
vertices_matrix=[000;001;010;011;100;101;110;111];
faces_matrix=[1562;1375;1243;2486;3784;5687];
colors=vertices_matrix;
path ('Vertices', vertices_matrix,' Faces', faces_matrix,…
        'FaceVertexCData',colors,'FaceColor','interp',…
            'EdgeAlpha',0)
if nargin=0
    vx=10;vy=10; vz=4;
elseif nargin ~=3
    error('Wrong number of inputs.')
end
axis off
view([vx,vy,vz])
axis square
```

6.1.2 Indexed Images

Indexed Images include two components: a data matrix of integers, x, and a

colormap matrix , map. Matrix map is an m×3 array of class double containing floating-point values in the range[0,1]. The length, m, of the map is equal to the number of colors it defines. Each row of map specifies the red, green, and blue components of a single color.

An indexed image uses “direct mapping” of pixel intensity values to colormap value.

These concepts can illustrate in following Fig:

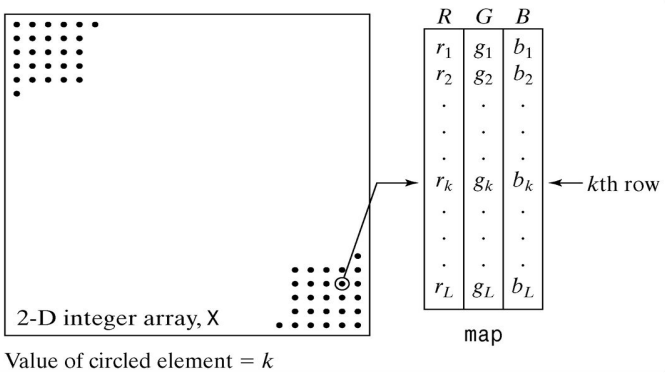


FIGURE 6.3 Elements of an indexed image. Note that the value of an element of integer array X determines the row number in the colormap. Each row contains an RGB triplet, and L is the total number of rows.

To display an indexed image we write:

>>imshow(x , map) or, alternatively,

>>image(x)

>>colormap(map)

A colormap is stored with an indexed image and is automatically loaded with the image when function imread is used to load the image.

Sometime, we use funtion imapporx to approximate an indexed image, whose syntax is:

[Y , newmap] = imapprox(x , map , n)

Return an array Y with colormap newmap which has at most n colors.

One approach to specify a color map:

>>map(k , :) = [r(k) g(k) b(k)]

% r(k) g(k) b(k) are RGB values that specify one row of a colormap

Table lists the RGB values for some basis color.Any of the three formats shown in the table can be used to specify color.

For example: the background color of a figure can be changed to green by using any of the following three statements:

>>whitebg(‘g’)

>>whitebg('green')

>>whitebg([0 1 0])

Long name	Short name	RGB values
Black	k	[0 0 0]
Blue	b	[0 0 1]
Green	g	[0 1 0]
Cyan	c	[0 1 1]
Red	r	[1 0 0]
Magenta	m	[1 0 1]
Yellow	y	[1 1 0]
White	w	[1 1 1]

TABLE 6.1 RGB values of some basic colors. The long or short names (enclosed by quotes) can be used instead of the numerical triplet to specify an RGB color.

MATLAB provides several predefined color maps, accessed using the command:

colormap(map_name)

>>colormap(copper)

>>imshow(x , copper)

Following table 6.2 lists some of the colormaps available in MATLAB. The length (number of colors) of these colormaps can be specified by enclosing the number in parentheses ( ). For example, gray (16) generates a colormap with 16 shades of gray.

Name	Description
autumn	Varies smoothly from red, through orange, to yellow.
bone	A gray-scale colormap with a higher value for the blue component. This colormap is useful for adding an "electronic" look to gray-scale images.
colorcube	Contains as many regularly spaced colors in RGB color space as possible, while attempting to provide more steps of gray, pure red, pure green, and pure blue.
cool	Consists of colors that are shades of cyan and magenta. It varies smoothly from cyan to magenta.
copper	Varies smoothly from black to bright copper.
flag	Consists of the colors red, white, blue, and black. This colormap completely changes color with each index increment.
gray	Returns a linear gray-scale colormap.
hot	Varies smoothly from black, through shades of red, orange, and yellow, to white.
hsv	Varies the hue component of the hue-saturation-value color model. The colors begin with red, pass through yellow, green, cyan, blue, magenta, and return to red. The colormap is particularly appropriate for displaying periodic functions.
jet	Ranges from blue to red, and passes through the colors cyan, yellow, and orange.
lines	Produces a colormap of colors specified by the ColorOrder property and a shade of gray. Consult online help regarding function ColorOrder.
pink	Contains pastel shades of pink. The pink colormap provides sepia tone colorization of grayscale photographs.
prism	Repeats the six colors red, orange, yellow, green, blue, and violet.
spring	Consists of colors that are shades of magenta and yellow.
summer	Consists of colors that are shades of green and yellow.
white	This is an all white monochrome colormap.
winter	Consists of colors that are shades of blue and green.

TABLE 6.2 Some of the MATLAB predefined colormaps.

6.1.3 IPT functions for Manipulating RGB and Indexed Images

Function	Purpose
dither	Creates an indexed image from an RGB image by dithering.
grayslice	Creates an indexed image from a gray-scale intensity image by multilevel thresholding.
gray2ind	Creates an indexed image from a gray-scale intensity image.
ind2gray	Creates a gray-scale intensity image from an indexed image.
rgb2ind	Creates an indexed image from an RGB image.
ind2rgb	Creates an RGB image from an indexed image.
rgb2gray	Creates a gray-scale image from an RGB image.

TABLE 6.3 IPT functions for converting between RGB, indexed, and gray-scale intensity images.

Function dither is applicable both to gray-scale and color images. Dithering is to give the visual impression of shade variations on a printed page that consists of dots. the syntax used by function dither for gray-scale image is:

bw = dither(gray_image)

% bw is the binary image

bw is the dithered result(a binary image).

x = grayslice(gray_image , n)

This function produces an indexed image by thresholding gray_image with threshhold values

1/n,2/n,……n-1/n

an alternate syntax is:

x = grayslice(gray_image , v)

v is a vector whose values are used to threshold gray_image.

Function gray2ind, syntax is:

[x , map] = gray2ind(gray_image , n)

Function ind2gray,b syntax is:

gray_image = ind2gray(x , map)

converts an indexed image, composed of x and map, to a gray-scale image.

Function rgb2ind:

[x , map] = rgb2ind(rgb_image , n , dither_option)

Function ind2rgb: (converts the matrix X and corresponding colormap map to RGB format)

rgb_image = ind2rgb(x , map)

Function rgb2gray: (coverts an RGB image to a gray-scale image)

gray_image = rgb2gray(rgb_image)

Example 6.1: Illustration of some of the functions in Table 6.3

Function rgb2ind is quite useful for reducing the number of colors in an RGB image.

Figure6.4(a) is a 24-bit RGB Image, f. Figure6.4(b) and Figure6.4(c) show the results of using the commands

```
>>[x1 , map1] = rgb2ind(f , 8 , 'nodither');
>>imshow(x1 , map1)        and
>> [x2 , map1] = rgb2ind(f , 8 , 'dither');
>>figure , imshow(x2 , map2)
```

The effects of dithering are usually better illustrated with gray-scale images. Figure6.4(d) and Figure6.4(e) were obtained using the commands

```
>>g = rgb2gray(f)
>>g1 = dither(g)
>>figure , imshow(g) ; figure , imshow(g1)
```

a
b c
d e

FIGURE 6.4
(a) RGB image. (b) Number of colors reduced to 8 without dithering. (c) Number of colors reduced to 8 with dithering. (d) Gray-scale version of (a) obtained using function `rgb2gray`. (e) Dithered gray-scale image (this is a binary image).

6.2 Converting to Other Color Spaces

Other color spaces include the NTSC,YCbCr,HSV,CMY,CMYK,and HSI color spaces.

6.2.1 NTSC Color Space

used in television in United States. NTSC formats image data consist of three component: luminance(Y), hue(I),and saturation(Q). YIQ components are obtained from the RGB components of an image using the transformation:

$$\begin{bmatrix} Y \\ I \\ Q \end{bmatrix} = \begin{bmatrix} 0.299 & 0.587 & 0.114 \\ 0.596 & -0.274 & -0.322 \\ 0.211 & -0.523 & 0.312 \end{bmatrix} \begin{bmatrix} R \\ G \\ B \end{bmatrix}$$

yiq_image = rgb2ntsc(rgb_image)

where the input RGB image can be of class unit8,unit16,or double. The output image is an M×N×3 array of class double.

the RGB components are obtained from the YIQ components using the transformation:

$$\begin{bmatrix} R \\ G \\ B \end{bmatrix} = \begin{bmatrix} 1.000 & 0.956 & 0.621 \\ 1.000 & -0.272 & -0.647 \\ 1.000 & -1.106 & 1.732 \end{bmatrix} \begin{bmatrix} Y \\ I \\ Q \end{bmatrix}$$

rgb_image = ntsc2rgb(yiq_image)

6.2.2 The YCbCr Color Space

used in digital video.

$$\begin{bmatrix} Y \\ Cb \\ Cr \end{bmatrix} = \begin{bmatrix} 16 \\ 128 \\ 128 \end{bmatrix} + \begin{bmatrix} 65.484 & 128.553 & 24.966 \\ -37.797 & -74.203 & 112.000 \\ 112.000 & -93.786 & -18.214 \end{bmatrix} \begin{bmatrix} R \\ G \\ B \end{bmatrix}$$

ycbcr_image = rgb2ycbcr(rgb_image)

Transformation converts from YCbCr back to RGB:

rgb_image = ycbcr2rgb(ycbcr_image)

6.2.3 The HSV Color Space

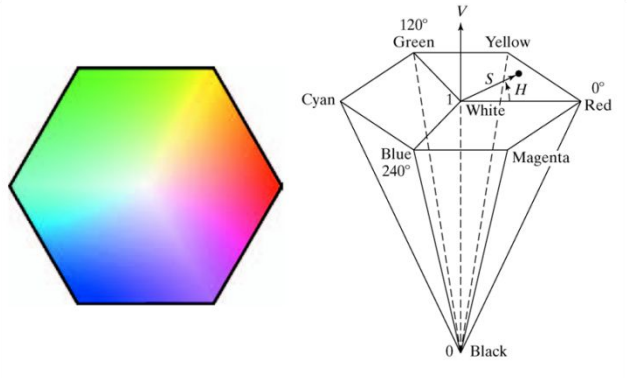
HSV(hue,saturation,value) is one of color system used by people to select color from a color wheel or palette.

The MATLAB function for converting from RGB to HSV is rgb2hsv:

hsv_image = rgb2hsv(rgb_image)

The function for converting from HSV to RGB is hsv2rgb :

rgb_image = hsv2rgb(hsv_image)



a b

FIGURE 6.5
(a) The HSV color hexagon.
(b) The HSV hexagonal cone.

6.2.4 The CMY and CMYK Color Space

the CMY s are obtained from the RGB using the transformation:

$$\begin{bmatrix} C \\ M \\ Y \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} - \begin{bmatrix} R \\ G \\ B \end{bmatrix}$$

Function imcomplement can be used to convert from RGB to CMY

my_image = imcomplement(rgb_image)

Also can convert from CMY to RGB:

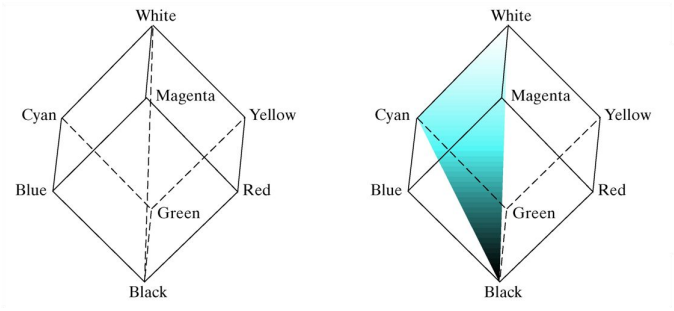
rgb_image = imcomplement(cmy_image)

6.2.5 The HSI Color Space

HSI (hue, saturation, intensity)

Hue is an attribute that describe a pure color (e.g., pure yellow, orange, or red), Saturation gives a measure of the degree to which a pure color is diluted by white light. Brightness is a subjective descriptor that is practically impossible to measure. It embodied the achromatic notion of intensity (gray level) is a most useful descriptor of monochromatic images.

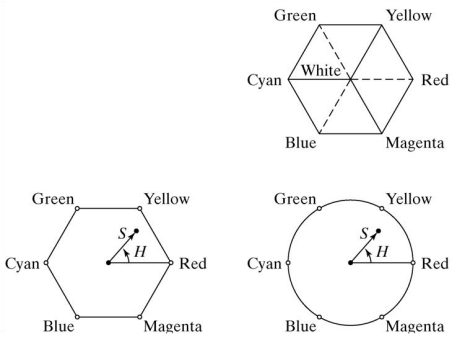
In fig6.6(b) we can see how hue can be determined from a given RGB point, which shows a plane defined by three points, (black, white and cyan) .



a b

FIGURE 6.6 Relationship between the RGB and HSI color models.

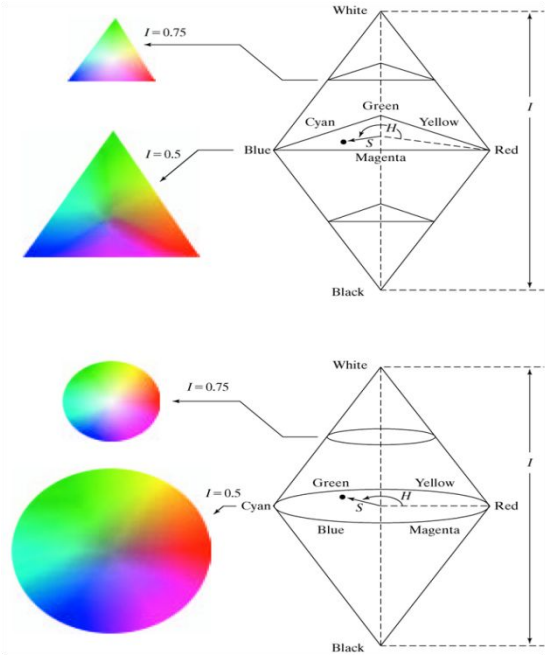
6- 9



a
b c d

FIGURE 6.7 Hue and saturation in the HSI color model. The dot is an arbitrary color point. The angle from the red axis gives the hue, and the length of the vector is the saturation. The intensity of all colors in any of these planes is given by the position of the plane on the vertical intensity axis.

6- 10



a
b

FIGURE 6.8 The HSI color model based on (a) triangular and (b) circular color planes. The triangles and circles are perpendicular to the vertical intensity axis.

6- 11

Fig6.8 shows the HSI model based on color triangles and also on circles.

As the HSI plane moves up and down the intensity axis, the boundary defined by the intersection of the plane with the faces of the cube have either a triangular or hexagonal shape. This can be visualized more readily. By looking at the cube down its gray-scale axis, as shown in fig6.7(a). fig 6.7(b) show hexagonal shape and an

arbitrary color point(show as a dot). fig 6.7(c) , (d) describe as a triangle or a circle.

Converting Color from RGB to HIS

$$H = \begin{cases} \theta & if\ B \le G \\ 360 - \theta & if\ B > G \end{cases} \qquad \theta = \cos^{-1}\left\{\frac{\frac{1}{2}[(R-G)+(R-B)]}{[(R-G)^2+(R-B)(G-B)]^{1/2}}\right\}$$

$$S = 1 - \frac{3}{(R+G+B)}[\min(R,G,B)]$$

I = (R+G+B)/3

Converting Color from HSI to RGB

$0° \le H < 120°$ $\qquad 120° \le H < 240°$ $\qquad 240° \le H \le 360°$

$R = I[1 + \frac{S\cos H}{\cos(60° - H)}]$ $\qquad$ R=I(1-S) $\qquad$ R=3I-(R+G)

G=3I-(R+B) $\qquad$ $G = I[1 + \frac{S\cos H}{\cos(60° - H)}]$ $\qquad$ G=I(1-S)

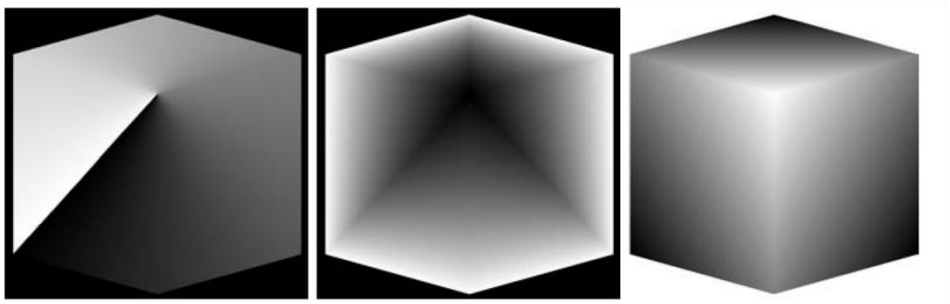
B=I(1-S) $\qquad$ B=3I-(R+G) $\qquad$ $B = I[1 + \frac{S\cos H}{\cos(60° - H)}]$

An M-function for Converting from RGB to HSI:

hsi = rgb2hsi(rgb)

An M-function for Converting from HSI to RGB:

rgb = hsi2rgb(hsi)



a b c

FIGURE 6.9 HSI component images of an image of an RGB color cube. (a) Hue, (b) saturation, and (c) intensity images.

Example 6.2: Converting from RGB to HSI

Fig6.9 show the hue,saturation,and intensity components of an image of an RGB cube on a white background. Fig6.7(a) is the hue image. Fig6.9(b) is the saturation image. It show progressively darker values toward the white vertex of the RGB cube. Fig6.9(c) is the intensity image. It show the average of the RGB values at the

corresponding pixel in fig6.2(b).

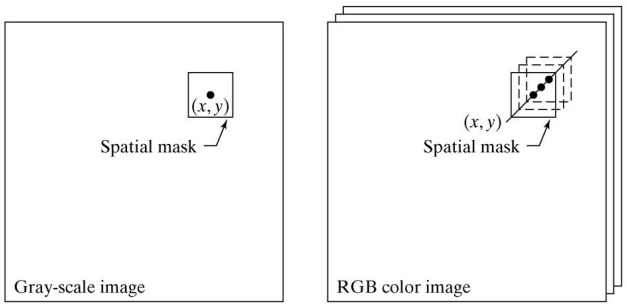
6.3 The Basis of Color Image Processing

Three principle areas

(1) Color transformations (also called color mapping);

(2) Spatial processing of individual color planes;

(3) Color vector processing.



a b

FIGURE 6.10
Spatial masks for gray-scale and RGB color images.

Fig6.10 shows spatial neighborhood processing of gray-scale and full-color image. In fig(a) averaging would be accomplished by summing the gray levels of all the pixels in the neighborhood and dividing by the total number of pixels in the neighborhood. In fig(b) averaging would be done by summing all the vectors in the neighborhood and dividing each component by the total number of vectors in the neighborhood.

6.4 Color Transformation

The technique in this section are based on processing the color components of a color image or intensity component of a monochrome image within the context of a single color image.

A color image c=[Cr; Cg; Cb]=[R; G; B]

C(x,y)=[Cr(x,y); Cg(x,y); Cb(x,y)]=[R(x,y); G(x,y); B(x,y)]

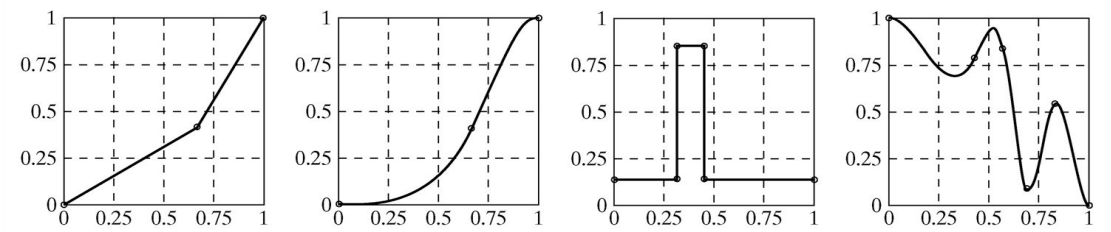
Transformation form

$S_i=T_i(r)$, i=1,2,…,n

r denotes gray-level values, Si and Ti are as color input/output images, n is the dimension of the color space (number of color components).

Fig 6.11 show a simple but powerful way to specify mapping functions graphically. Fig 6.11(a) shows a transformation that is formed by linearly

interpolating three control points (the circled coordinates in the figure);Fig 6.11(b) shows the transformation that results from a cubic spline interpolation of the same three points. Fig 6.11(c), (d) provide more complex linear and cubic spline interpolation, respectively.



a b c d

FIGURE 6.11 Specifying mapping functions using control points: (a) and (c) linear interpolation, and (b) and (d) cubic spline interpolation.

Linear interpolation is implemented by using:

z = interp1q(x , y , xi)

It returns a column vector containing the values of linearly interpolated 1-D function z at points xi. Column vectors x and y specify the horizontal and vertical coordinate pairs of the underlying control points. The elements of x must increase monotonically. The length of z is equal to the length of xi. For example:

>> z = interp1q([0 255] ‘, [0 255]’ , [0 : 255]’)

Cubic spline interpolation is implemented using the spline function

z = spline(x , y , xi)

The development of function ice, given in Appendix B, is a comprehensive illustration of how to design a graphical user interface(GUI) in MATLAB. Its syntax is:

g = ice(‘Property Name’ , ‘Property value’ , …)

Table 6.4 lists the valid pairs for use in function ice.

Property Name	Property Value
`'image'`	An RGB or monochrome input image, `f`, to be transformed by interactively specified mappings.
`'space'`	The color space of the components to be modified. Possible values are `'rgb'`, `'cmy'`, `'hsi'`, `'hsv'`, `'ntsc'` (or `'yiq'`), and `'ycbcr'`. The default is `'rgb'`.
`'wait'`	If `'on'` (the default), `g` is the mapped input image. If `'off'`, `g` is the handle of the mapped input image.

TABLE 6.4 Valid inputs for function `ice`.

To obtain the properties of an image with handle g we use the get function.

h = get(g)

This function returns all properties and applicable current values of the graphics objects identified by the handle g.

Example of the syntax of function ice:

ice

g = ice('image',f);

g = ice('image',f,'wait','off');

g = ice('image',f,'space','off');

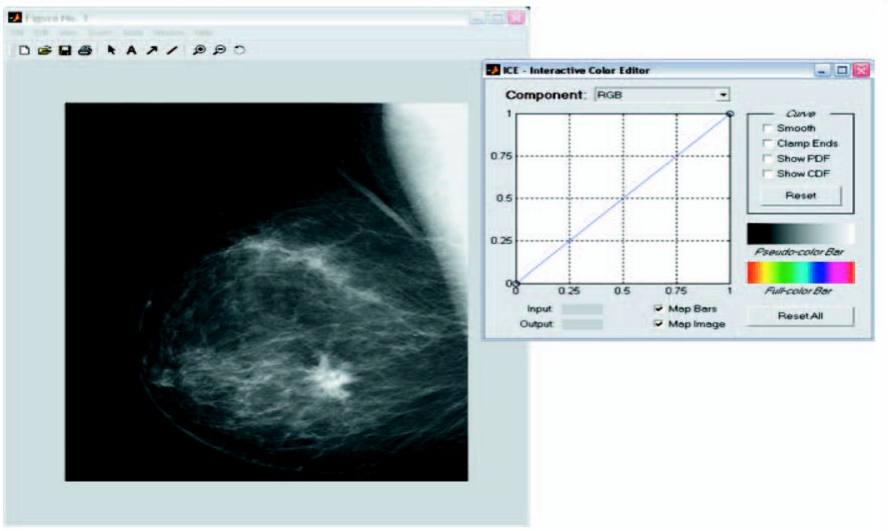


FIGURE 6.12 The typical opening windows of function `ice`. (Image courtesy of G. E. Medical Systems.)

Control points are manipulated with the mouse, as summarized in table 6.5. Table 6.6 lists the function of the other GUI components.

Mouse Action†	Result
Left Button	Move control point by pressing and dragging.
Left Button + Shift Key	Add control point. The location of the control point can be changed by dragging (while still pressing the Shift Key).
Left Button + Control Key	Delete control point.

† For three button mice, the left, middle, and right buttons correspond to the move, add, and delete operations in the table.

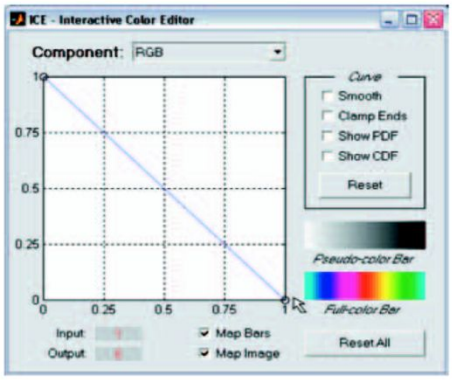
TABLE 6.5
Manipulating control points with the mouse.

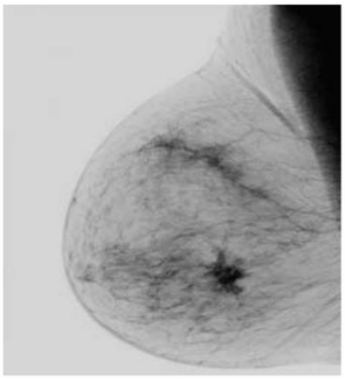
GUI Element	Function
Smooth	Checked for cubic spline (smooth curve) interpolation. If unchecked, piecewise linear interpolation is used.
Clamp Ends	Checked to force the starting and ending curve slopes in cubic spline interpolation to 0. Piecewise linear interpolation is not affected.
Show PDF	Display probability density function(s) [i.e., histogram(s)] of the image components affected by the mapping function.
Show CDF	Display cumulative distribution function(s) instead of PDFs. (Note: PDFs and CDFs cannot be displayed simultaneously.)
Map Image	If checked, image mapping is enabled; otherwise it is not.
Map Bars	If checked, pseudo- and full-color bar mapping is enabled; otherwise the unmapped bars (a gray wedge and hue wedge, respectively) are displayed.
Reset	Initialize the currently displayed mapping function and uncheck all curve parameters.
Reset All	Initialize all mapping functions.
Input/Output	Shows the coordinates of a *selected* control point on the transformation curve. Input refers to the horizontal axis, and Output to the vertical axis.
Component	Select a mapping function for interactive manipulation. In RGB space, possible selections include R, G, B, and RGB (which maps all three color components). In HSI space, the options are H, S, I, and HSI, and so on.

TABLE 6.6
Function of the checkboxes and pushbuttons in the ice GUI.

The following example show typical applications of function ice

Example 6.3:Inverse mappings: monochrome negatives and color complements Figure 6.13(a) shows the ice interface after the default RGB curve of Fig 6.12 is modified to produce an inverse or negative mapping function figure 6.13(b) shows the monochrome negative that results from the inverse mapping.





a b
FIGURE 6.13
(a) A negative mapping function, and (b) its effect on the monochrome image of Fig. 6.12.

Inverse or negative mapping functions also are useful in color processing. As can be seen in fig. 6.14(a) and (b), the result of the mapping is reminiscent of conventional color film negatives. Fig.6.14(a) is transformed to cyan in fig.6.14(b)---the color complement of red.

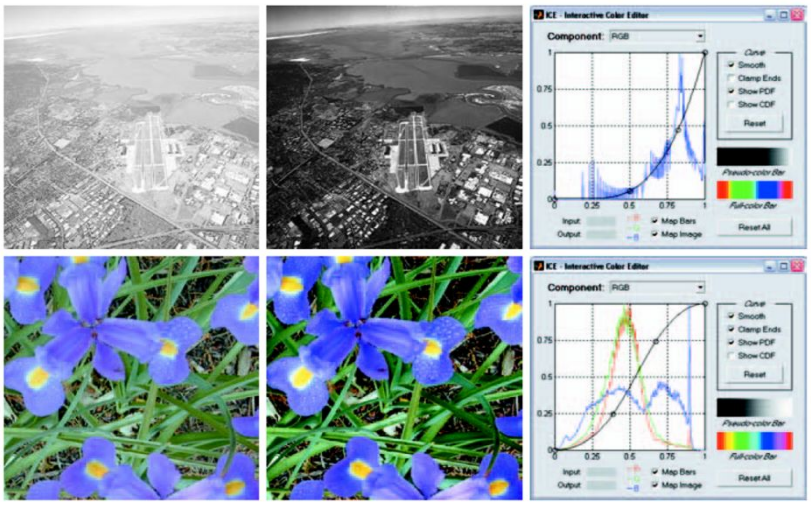
Consider next the use of function ice for monochrome and color contrast manipulation. Figure 6.15(a) through (c) demonstrate the effectiveness of ice in processing monochrome image. Figure 6.15(d) through (f) show similar effectiveness for color inputs.

a b

FIGURE 6.14
(a) A full color image, and (b) its negative (color complement).

6- 17

Example 6.4: Monochrome and color contrast enhancement



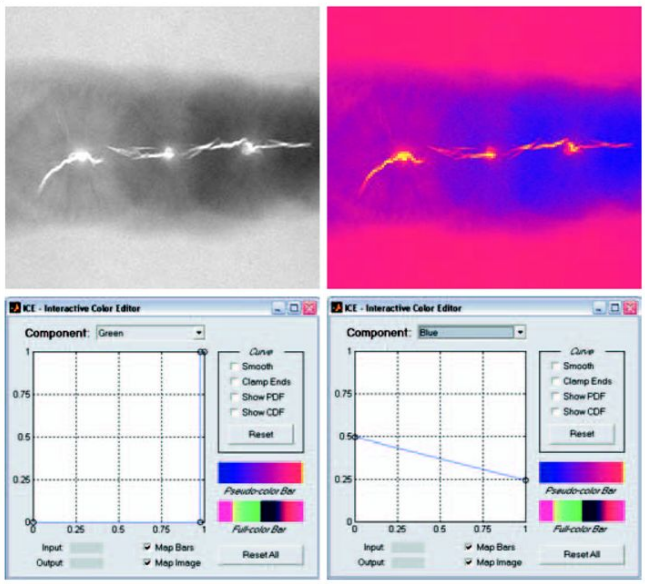
a b c
d e f

FIGURE 6.15 Using function `ice` for monochrome and full color contrast enhancement: (a) and (d) are the input images, both of which have a "washed-out" appearance; (b) and (e) show the processed results; (c) and (f) are the `ice` displays. (Original monochrome image for this example courtesy of NASA.)

6- 18

Example 6.5: Pseudocolor mappings.

Figure 6.16(a) is an X-ray image of a weld(the horizontal dark region) containing several cracks and porosities (the bright white streaks running through the middle of the image). Figure 6.16(b) show a pseudocolor version of the image,it was generated by mapping the green and blue components of the RGB-converted input using the mapping function in Fig.6.16(c) and (d). The interactively specified mapping function transform the black-to-white gray scale to hues between blue and red, with yellow reserved for white.



a b
c d

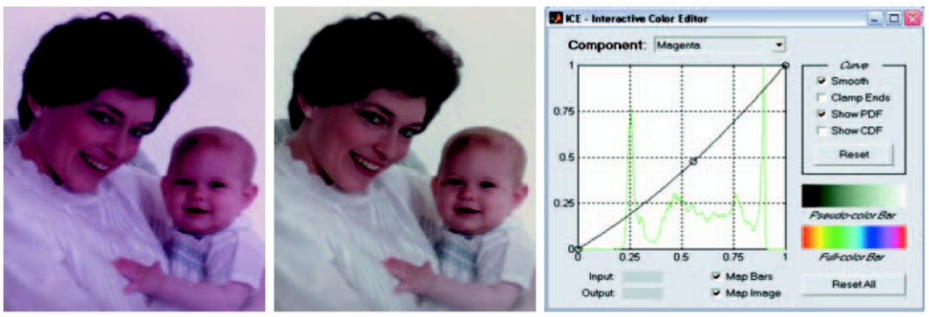
FIGURE 6.16
(a) X-ray of a defective weld; (b) a pseudo-color version of the weld; (c) and (d) mapping functions for the green and blue components. (Original image courtesy of X-TEK Systems, Ltd.)

6- 19

Example 6.6: Color balancing.

Figure 6.17 shows an application involving a full-color image,in which it is advantageous to map an image's color components independently. figure 6.17(a) shows a CMY scan of a mother and her child with an excess of magenta (keep in mind that only an RGB version of the image can displayed by MATLAB). to interactively modify the CMY components of RGB image f1, for Example, the appropriate ice call is:

```
>>f2 = ice('image' , f1 , 'space' , 'CMY');
```



a b c

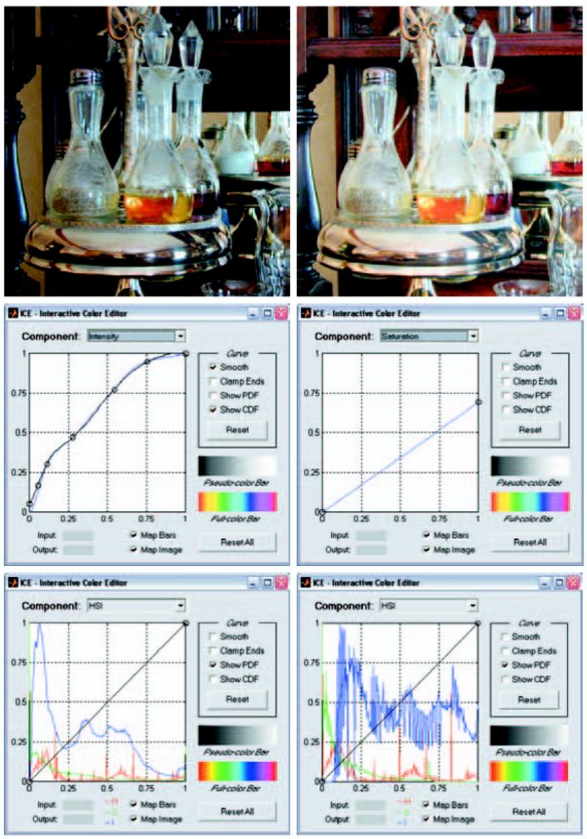
FIGURE 6.17 Using function `ice` for color balancing: (a) an image heavy in magenta; (b) the corrected image; and (c) the mapping function used to correct the imbalance.

6- 20

Example 6.7: Histogram based Mappings.

Figure 6.18 (a) shows a color image of a caster stand containing cruets and shakers. The transformed image in fig.6.18(b),which was produce using the HSI transformations in fig.6.18(c) and (d), is significant image. Note that the histograms of the input and output image's hue, saturation, and intensity components are shown in

fig.6.18(e) and (f).



a b
c d
e f

FIGURE 6.18 Histogram equalization followed by saturation adjustment in the HSI color space: (a) input image; (b) mapped result; (c) intensity component mapping function and cumulative distribution function; (d) saturation component mapping function; (e) input image's component histograms; and (f) mapped result's component histograms.

6.5 Spatial Filter of Color Image

6.5.1 Color Image Smoothing

Smoothing an RGB color image, fc, with a linear spatial filter consists of the following steps:

1. Extract the three component images:

```
>>fR = fc( : , : , 1); fG = fc( : , : , 2); fB = fc( : , : , 3);
```

2. Filter each component images individually. Letting w represent a smoothing filter generated using fspecial , we smooth the red component images as follows:

```
>> fR_filtered = imfilter(fR , w);
```

3. Reconstruct the filter RGB images:

```
>>fc_filtered = cat(3 , fR_filtered , fG_filtered , fB_filtered );
```

Or combine three steps into one:

>>fc_filtered = imfilter(fR , w);

Example 6.8: Color image smoothing.

Figure 6.19(a) shows an RGB image of size 1197 ×1197 pixels and figs 6.19(b) through (d) are its RGB component images, extracted using the procedure described in the previous paragraph.

a b
c d

FIGURE 6.19
(a) RGB image; (b) through (d) are the red, green and blue component images, respectively.

Fig 6.20 (a) through (c) show the three HSI component images of fig.6.19(a), obtained using function rgb2hsi.

a b c
FIGURE 6.20 From left to right: hue, saturation, and intensity components of Fig. 6.19(a).

Fig. 6.21(a) shows the result of smoothing the image in fig.6.19(a) using function imfilter with the 'replicate' option and an 'average' filter of size 25 ×25 pixels.fig. 6.21(b) was obtained using the command:

```
>>h = rgb2hsi(fc);
>>H = h(: , : , 1 ); S = h(: , : , 2 ); I = h(: , : , 3 );
>>w = fspecial('average' , 25);
>>I_filtered = imfilter(I , w , 'replicate');
>>h = cat(3 , H , S , I_filtered );
```

>>f = hsi2rgb(h);

>>f = min(f , 1);

>>imshow(f)

a b c

FIGURE 6.21 (a) Smoothed RGB image obtained by smoothing the *R*, *G*, and *B* image planes separately. (b) Result of smoothing only the intensity component of the HSI equivalent image. (c) Result of smoothing all three HSI components equally.

6.5.2 Color Image Sharpening

Sharpening an RGB color image with a linear spatial filter follows the same procedure outlined in the previous section, but using a sharpening filter instead.

Example 6.9: color image sharpening

To sharpen the image we use the Laplacian filter mask

>>lapmark = [1 1 1 ; 1 –8 1 ; 1 1 1]

The enhanced image was computed and displayed using the command

>>fen = imsubtract(fb , imfilter(fb , lapmask , 'replicate'));

>>imshow(fen)

a b

FIGURE 6.22 (a) Blurred image. (b) Image enhanced using the Laplacian, followed by contrast enhancement using function `ice`.

6.6 Working Directly in RGB Vector Space

6.6.1 Color Edge Detection Using the Gradient

Figure 6.23(a) shows a neighborhood of 3 ×3, where the z's indicate pixel value. The image separated two masks shown in Figure 6.23(b) and (c).

z_1	z_2	z_3
z_4	z_5	z_6
z_7	z_8	z_9

−1	−2	−1
0	0	0
1	2	1

−1	0	1
−2	0	2
−1	0	1

a b c

FIGURE 6.23 (a) A small neighborhood. (b) and (c) Sobel masks used to compute the gradient in the x (vertical) and y (horizontal) directions, respectively, with respect to the center point of the neighborhood.

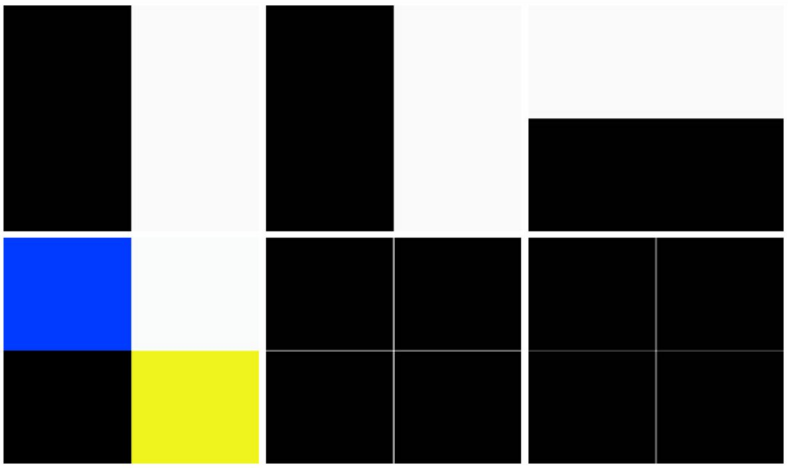
6- 26

The following function implements the color gradient for RGB images:

[VG , A , PPG] = colorgrad(f, T)

Where f is an RGB image, T is an optional threshold in the range[0,1]; VG is the RGB vector gradient Fθ(x,y); A is the angle image θ(x,y), in radians; and PPG is the gradient formed by summing the 2-D gradients of the individual color planes (generated for comparison purposes).

Example 6.10: RGB edge detection using function colorgrad.



a b c
d e f

FIGURE 6.24 (a) through (c) RGB component images (black is 0 and white is 255). (d) Corresponding color image. (e) Gradient computed directly in RGB vector space. (f) Composite gradient obtained by computing the 2-D gradient of each RGB component image separately and adding the results.

6- 27

Figure 6.24(a) through (c) show three simple monochrome image which,when used as RGB planes, produced the color image in fig.6.24(d). The objectives of this example are

(1) to illustrate the use of function colorgrad

(2) to show that computing the gradient of a color image by combining the gradient of its individual color planes is quite different from computing the

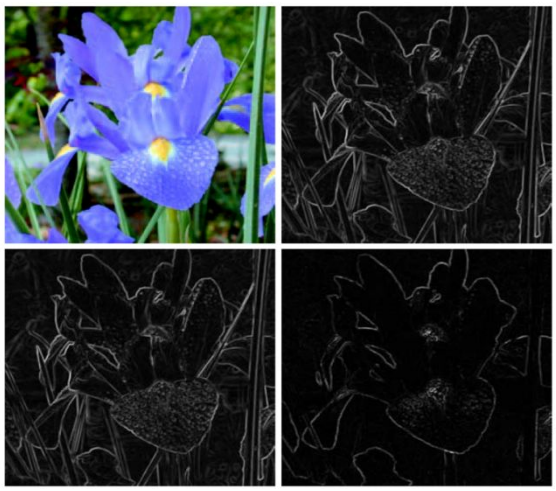
gradient directly in RGB vector space using the method just explained.

Let the f represent the RGB image in fig.6.24(d), the command:

>>[VG , A , PPG] = colorgrad(f)

Produced the image VG and PPG shown in fig.6.24(e) and (f).

For example: Fig.6.25(b) and (c) are analogous to fig.6.24(e) and (f). They were obtained by applying function colorgrad to the image in fig.6.25(a). Fig.6.25(d) is the difference of the two gradient images, scaled to the range [0,1]

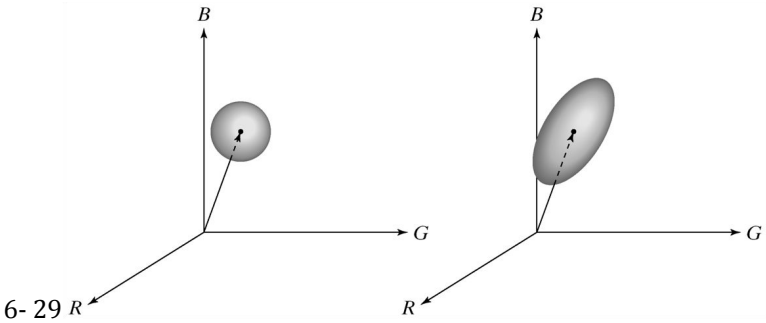


a b
c d

FIGURE 6.25
(a) RGB image.
(b) Gradient computed in RGB vector space.
(c) Gradient computed as in Fig. 6.24(f).
(d) Absolute difference between (b) and (c), scaled to the range [0, 1].

6.6.2 Image Segmentation in RGB Vector Space

Segmentation is a process that partitions an image into region. We consider here briefly for the sake of continuity.



a b

FIGURE 6.26 Two approaches for enclosing data in RGB vector space for the purpose of segmentation.

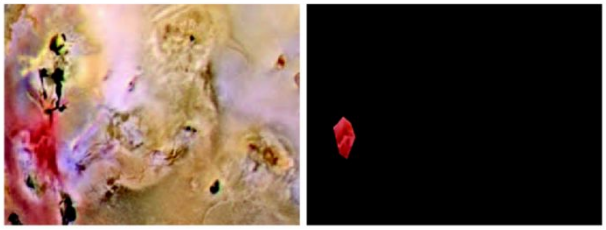
Segmentation in the manner just described is implemented by function colorseg ,which has the syntax (see Appendix C for the code)

S = colorseg(method , f , T , parameters)

Where method is either ‘euclidean’ or ‘mahalanobis’, f is the RGB image to be

segmented, and T is the threshold described above.

Example 6.11: RGB color image segmentation



a b

FIGURE 6.27 (a) Pseudocolor of the surface of Jupiter's Moon Io. (b) Region of interest extracted interactively using function `roipoly`. (Original image courtesy of NASA.)

Letting f denote the color image in fig.6.27(a),the region in fig.6.27(b) was obtained using the commands

```
>>mask = roipoly(f);
>>red = immultiply(mask , f( ; , : , 1));
>>green = immultiply(mask , f( ; , : , 2));
>>blue = immultiply(mask , f( ; , : , 3));
>>g = cat(3 , red , green , blue);
>>figure , imshow(g)
```

Where mask is a binary image (the same size as f) with 0s in the background and 1s in the region selected interactively.

Next we compute the mean vector and covariance matrix of the points in the ROI(region of interest), but first the coordinates of the points in the ROI must be extracted.

```
>>[m , n , k] = size(g);
>>I = reshape(g ,M*N , 3);
>>I = double(I(idx , 1:3));
>>[c , m] = covmatrix(I);
```

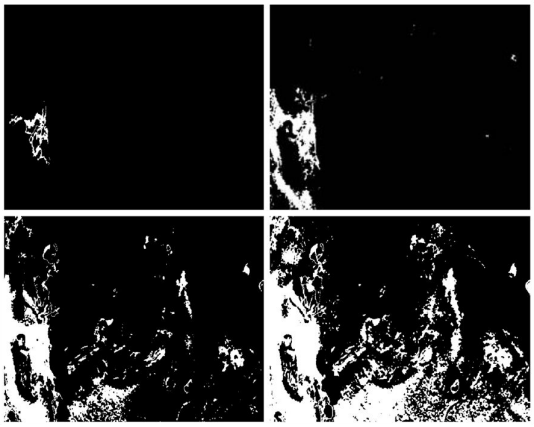
There are the non-background pixels of the masked image in fig.6.27(b).The main diagonal of C contains the variances of the RGB component , all we have to do is extract these elements and compute their square roots:

```
>>d = diag(c);
>>sd = sqrt(d);
22.0643      24.2442      16.1806
```

For the 'euclidean' option with T=25, we use:

```
>>E25 = colorseg('euclidean' , f , 25 , m);
```

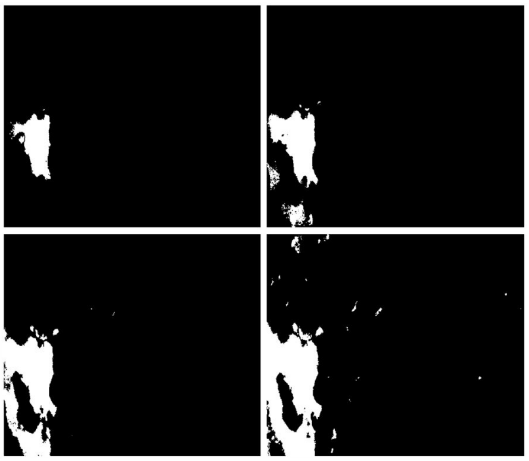
Figure 6.28(a) show the result, and fig.6.28(b) through (d) show the segmentation result with T=50,75,100.



a b
c d

FIGURE 6.28
(a) through (d) Segmentation of Fig. 6.27(a) using option `'euclidean'` in function `colorseg` with $T = 25, 50, 75,$ and 100, respectively.

Fig.6.29(a) though (d) show the results obtained using the ‘mahalanobis’ option with the same sequence of threshold values.



a b
c d

FIGURE 6.29
(a) through (d) Segmentation of Fig. 6.27(a) using option `'mahalanobis'` in function `colorseg` with $T = 25, 50, 75,$ and 100, respectively. Compare with Fig. 6.28.

Summary

Basic topics in the application and use of color in image processing.

Implement of these concept using MATLAB, IPT and new functions in the preceding sections.

Chapter7 Wavelets

Preview

Digital images are processed at multiple resolutions, DWT is a good choice. It is an efficient, highly intuitive framework for the representation and storage of multiresolution images. DWT provides powerful insight into an image's spatial and frequency characteristics, but FF only an image's frequency attributes.

Explore both the computation and use of the discrete wavelet transform.

Introduce the Wavelet Toolbox, a collection of MathWorks' functions designed for wavelet analysis but not included in MATLAB's Image Processing Toolbox(IPT)

Develop a compatible set of routines that allow basic wavelet-based processing using IPT, without the Wavelet Toolbox.

These custom functions, in combination with IPT, provide the tools needed to implement all the concepts discussed in Chapter 7 of Digital Image Processing by Gonzalez and Woods[2002].

They are applied in much the same way—and provide a similar range of capabilities—as IPT functions fft2 and ifft2 in Chapter 4.

7.1 Background

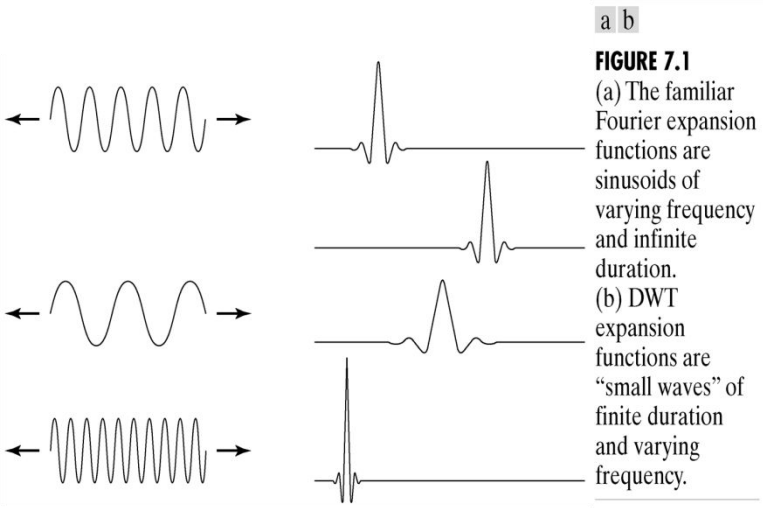
Consider an image *f(x,y)* of size M×N whose forward, discrete transform, T(u, v,…), can be expressed in terms of the general relation (page 242-243). The $g_{u,v,\ldots}$ and $h_{u,v,\ldots}$ in these equations are called forward and inverse transformation kernels, respectively. Transform coefficients T(u,v,…) can be viewed as the expansion coefficients of a series expansion of *f* with respect to {h u,v,…}.

The discrete Fourier transform (DFT) fits this series expansion formulation well. (Page 243) The Kernels are separable and orthonormal.

Unlike the discrete Fourier transform, which can be completely defined by two straightforward equations that revolve around a single pair of transformation kernels，the term DWT refers to a class of transformations that differ not only in the

transformation kernels employed, but also in the fundamental nature of those functions and in the way in which they are applied.

Since the DWT encompasses a variety of unique but related transformations, We cannot write a single equation that complete describes them all. Instead, we characterize each DWT by a transform kernel pair or set of parameters that defines the pair. The various transforms are related by the fact that their expansion functions are "small waves" of varying frequency and limited duration [see Fig.7.1(b)].



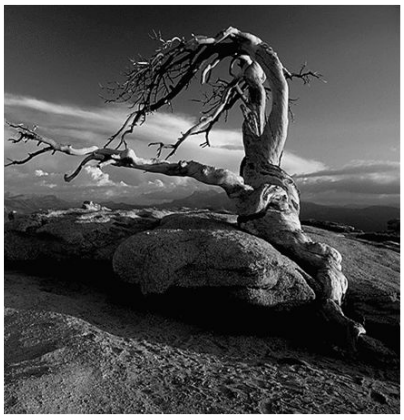
a b

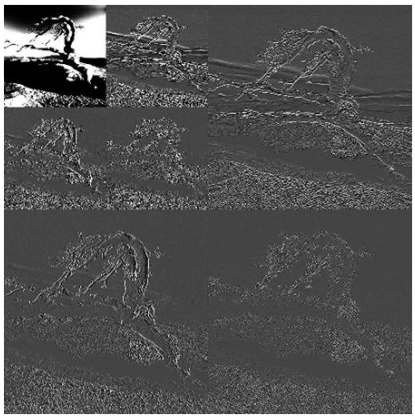
FIGURE 7.1
(a) The familiar Fourier expansion functions are sinusoids of varying frequency and infinite duration.
(b) DWT expansion functions are "small waves" of finite duration and varying frequency.

7-1

In the remainder of the chapter, we introduce a number of wavelet kernels. Each possesses the following general properties:

Property1: Separability, Scalability, and Translatability. The kernels can be represented as three separable 2-D wavelets and one separable 2-D scaling function (P.244). Each of these 2-D functions is the product of two 1-D real, square-integrable scaling and wavelet functions。





Property 2: Multiresolution Compatibility. The 1-D scaling function just introduced satisfies the following requirements of multiresolution analysis: 1. $\phi_{j,k}$ is

orthogonal to its integer translates. 2.The set of functions that can be represented as a series expansion of $\phi_{j,k}$ at low scales or resolutions is contained within those that can be represented at higher scales. 3.The only function that can be represented at every scale is *f(x)=0*. 4. Any function can be represented with arbitrary precision as j→∞.

Wavelet $\psi_{j,k}$, together with its integer translates and binary scalings, can represent the difference between any two sets of $\phi_{j,k}$ -representable functions at adjacent scales.

Property 3: Orthogonality. The expansion functions form an orthonormal or biorthogonal basis for the set of 1-D measurable, square-integrable functions.

7.2 The Fast Wavelet Transform

An important consequence of the above properties is that both $\phi(x)$ and $\psi(x)$ can be expressed as linear combinations of double-resolution copies of themselves. That is, via the series expansions[Page 245],where h_ϕ and h_ψ 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.7.2.

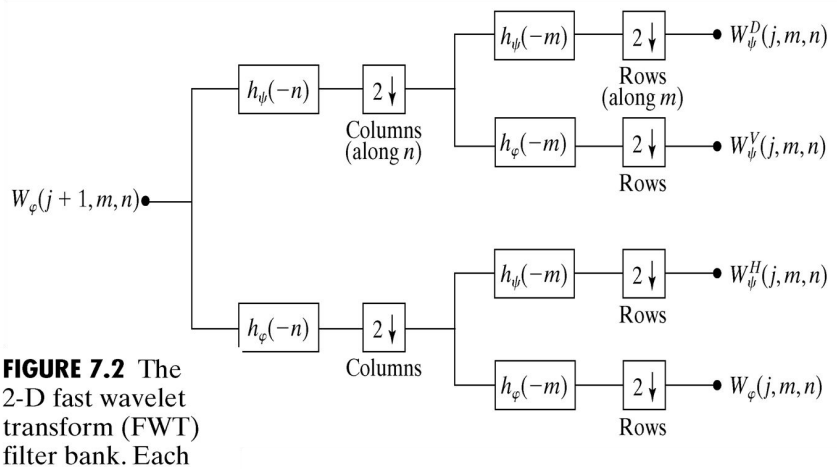


FIGURE 7.2 The 2-D fast wavelet transform (FWT) filter bank. Each pass generates one DWT scale. In the first iteration, $W_\varphi(j+1,m,n) = f(x,y)$.

7.2.1 FWTs using the Wavelet Toolbox

In this section, we use MATLAB's Wavelet Toolbox to compute the FWT of a simple 4*4 test image. 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:

[L0_D, Hi_D, Lo_R, Hi_R] = wfilters(wname)

Frequently coupled filter pairs can alternately be retrieved using [F1, F2] = wfilters(wname, type).

Table 7.1 lists the FWT filters included in the Wavelet Toolbox. Some important properties are provided by the Wavelet Toolbox's waveinfo and wavefun functions.

1.waveinfo(wfamily) % To print a written description of wavelet famliy wfamily.

Table 7.1

TABLE 7.1 Wavelet Toolbox FWT filters and filter family names.

Wavelet	wfamily	wname
Haar	'haar'	'haar'
Daubechies	'db'	'db2', 'db3', ..., 'db45'
Coiflets	'coif'	'coif1', 'coif2', ..., 'coif5'
Symlets	'sym'	'sym2', 'sym3', ..., 'sym45'
Discrete Meyer	'dmey'	'dmey'
Biorthogonal	'bior'	'bior1.1', 'bior1.3', 'bior1.5', 'bior2.2', 'bior2.4', 'bior2.6', 'bior2.8', 'bior3.1', 'bior3.3', 'bior3.5', 'bior3.7', 'bior3.9', 'bior4.4', 'bior5.5', 'bior6.8'
Reverse Biorthogonal	'rbio'	'rbio1.1', 'rbio1.3', 'rbio1.5', 'rbio2.2', 'rbio2.4', 'rbio2.6', 'rbio2.8', 'rbio3.1', 'rbio3.3', 'rbio3.5', 'rbio3.7', 'rbio3.9', 'rbio4.4', 'rbio5.5', 'rbio6.8'

2. [phi, psi, xval] = wavefun (wname, iter) % To obtain a digital approximation of an orthonormal transform's scaling and/or wavelet functions. For biorthogonal transforms, the appropriate syntax is

[phi1, psi1, phi2, psi2, xval] = wavefun(wname, iter).

EXAMPLE 7.1: Haar filters, scaling, and wavelet functions

The oldest and simplest wavelet transform is based on the Haar scaling and wavelet functions. The decomposition and reconstruction filters for a Haar_based transform are of length 2 and can be obtained as follows:

```
>> [Lo_D, Hi_D, Lo_R, Hi-R] = wfilters( ‘haar’)
Lo_D =
       0.7071        0.7071
Hi_D =
  -0.7071        0.7071
Hi_R =
0.7071      -0.7071
Lo_R =
   0.7071        0.7071
```

Their key properties and plots of the associated scaling and wavelet functions can be obtained using.

```
>> waveinfo( ‘haar’);
>> [phi, psi, xval] = wavefun('haar', 10);
>> xaxis = zeros(size(xval));
>> subplot(121); plot(xval, phi, 'k', xval, xaxis, '--k');
>> axis([0 1 -1.5 1.5]); axis square;
>> title('Haar Scaling funtion');
>> subplot(122); plot(xval, psi, 'k', xval, xaxis, '--k');
>> axis([0 1 -1.5 1.5]); axis square;
>> title('Haar Wavelet function');
```

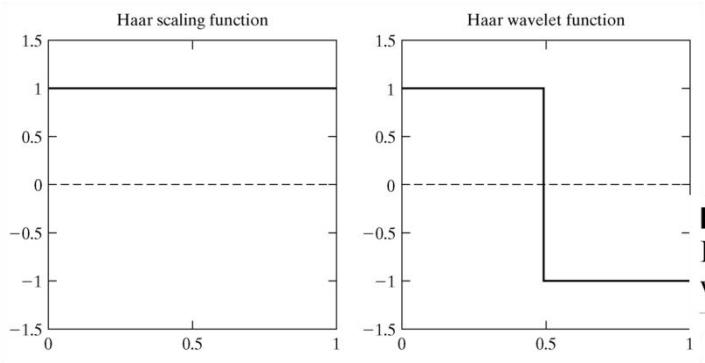


FIGURE 7.3 The Haar scaling and wavelet functions.

7-3

The Haar scalling and wavelet functions shown in Figure 7.3 are discontinuous and compactly supported. In addition, Because the Haar expansion functions are orthogonal,the forward and inverse transformation kernels are identical. Given a set of decomposition filters, the simplest way of computing the associated wavelet transform is through the Wavelet Toolbox’s wavedec2 function. It is invoked using [C, S] = wavedec2(X, N, Lo_D, Hi_D).

The slightly more efficient syntax [C, S] = wavedec2(X, N, Wname).

EXAMPLE 7.2: A simple FWT using Haar filters.

Consider the following single-scale wavelet transform with respect to Haar wavelets:

>> f = magic(4)

>> [c1, s1] = wavedec2(f, 1, 'haar')

When the single-scale transform described above is extended to two scales, we get

>> [c2, s2] = wavedec2(f, 2, 'haar')

To minimize border distortions, the border must be treated differently. Many Wavelet Toolbox functions, including the wavedec2 function, extend or pad the image being processed based on global parameter dwtmode.

>> dwtmode % To examine the active

>> st = dwtmode('status') % extension mode.

>> dwtmode(STATUS)

% To set the extension mode to STATUS.

>> dwtmode('save', STATUS)

% To make STATUS the default extension mode.

Table 7.2

STATUS	Description
'sym'	The image is extended by mirror reflecting it across its borders. This is the normal default mode.
'zpd'	The image is extended by padding with a value of 0.
'spd', 'sp1'	The image is extended by first-order derivative extrapolation—or padding with a linear extension of the outmost two border values.
'sp0'	The image is extended by extrapolating the border values—that is, by boundary value replication.
'ppd'	The image is extended by periodic padding.
'per'	The image is extended by periodic padding after it has been padded (if necessary) to an even size using 'sp0' extension.

TABLE 7.2
Wavelet Toolbox image extension or padding modes.

7.2.2 FWT without the wavelet Toolbox

In this section, we develop a pair of custom functions, wavefilter and wavefast, to replace the Wavelet Toolbox functions, wfilters and wavedec2. The first step is to devise a function for generating wavelet decomposition and reconstruction filter.

function [varargout] = wavefilter(wname, type)

To maintain compatibility with the existing Wavelet Toolbox, we employ the same decomposition structure(i.e.,[c,s]). The following routine, which we call wavefast, uses symmetric image extension to reduce the border distortion associated with the computed FWT:

function [c, s] = wavefast(x, n, varargin)

EXAMPLE 7.3: Comparing the execution times of wavefast and wavedec2.

The following test routine uses functions tic and toc to compare the execution times of the Wavelet Toolbox function wavedec2 and custom function wavefast:

function [ratio, maxdiff] = fwtcompare(f, n, wname)

For the 515*512 image of Fig.7.4 and a five-scale wavelet transform with respect to 4th order Daubechies' wavelets, fwtcompare yields

```
>> f = imread( 'Vase', 'tif' );
>> [ ratio, maxdifference] = fwtcompare( f, 5, 'db4')
ratio = 0.5508
% Execution time ratio (custom/toolbox)
maxdifference = 3.2969e-012
% Maximum coefficient difference.
```

Note that custom function wavefast was almost twice as fast as its Wavelet Toolbox counterpart while producing virtually identical results.

FIGURE 7.4
A 512 × 512 image of a vase.

7.3 Working with Wavelet Decomposition Structures

In this section, we formally define{c, S}, examine some of the Wavelet Toolbox functions for manipulating it, and develop a set of custom functions that can

be used without the Wavelet Toolbox. These functions are then used to build a general purpose routine for displaying c.

The representation scheme introduced in Example 7.2 integrates the coefficients of a multiscale two-dimensional wavelet transform into a single, one-dimentsional vector

c = [AN(:)' HN(:)' … Hi(:)' Vi(:)' Di(:)' … V1(:)' D1(:)']

Because c assumes N decompositions, c contains 3N+1 sections. Note that the highest scale coefficients are computed when i = 1; the lowest scale coefficients are associated with i = N. Thus, the coefficients of c are ordered from low to high scale.

Matrix S of the decomposition structure is an (N+2)*2 bookkeeping array of the form

S = [saN; sdN; sdN-1; … sdi; … sd1; sf]

The information in S can be used to locate the individual approximation and detail coefficients in c.

EXAMPLE 7.4: Wavelet Toolbox functions for manipulating transform decomposition vector **c**.

The Wavelet Toolbox provides a variety of functions for locating, extracting, reformatting, and/or manipulating the approximation and horizontal, vertical, and diagonal coefficients of **c** as a function of decomposition level. Consider, for example, the following sequence of commands:

```
>> f = magic(8);
>> [c1, s1] = wavedec2(f, 3, 'haar');
>> size(c1)
  ans =    1      64
>> s1
s1 =        1            1
            1            1
            2            2
            4            4
            8            8
>> approx = appcoef2(c1, s1, 'haar')
approx =    260.0000
>> horizdet2 = detocef2('h', c1, s1, 2)
horizdet2 =    1.0e-013      *
```

```
                0          -0.2842
                0          0
>> newc1 = wthcoef2('h', c1, s1, 2)
>> newhorizdet2 = detcoef2('h', newc1, s1, 2)
newhorizdet2 =    0          0
                  0          0
```

Matrix approx = 260 is extracted from c1 using toolbox function appceof2, which has the following syntax:

a = appcoef2(c, s, wname)

The horizontal detail coefficients at level 2 are retrieved using detcoef2, a function of similar syntax

d = detcoef2(o, c, s, n)

The coefficients corresponding to horizde2 in c1 are then zeroed using wthcoef2, a wavelet thresholding function of the form

nc = wthcoef2(type, c, s, n, t, sorth)

7.3.1 Editing Wavelet Decomposition Coefficients without the Wavelet Toolbox

Use **S** to build a set of general-purpose routines for the manipulation of c. Function wavework is the foundation of the routines developed, which are based on the familiar cut-copy-paste metaphor of modern word processing applications.

function [varargout] = wavework(opcode, type, c, s, n, x)

The following three functions—wavecut, wavecopy, and wavepaste—using wavework to manipulate c using a more intuitive syntax:

function [nc, y] = wavecut(type, c, s, n)

function y = wavecopy(type, c, s, n)

function nc = wavepaste(type, c, s, n, x)

EXAMPLE 7.5: Manipulating c with wavecut and wavecopy.

Functions wavecopy and wavecut can be used to reproduce the Wavelet Toolbox based results of Example 7.4:

```
>> f = magic(8);
>> [c1, s1] = wavedec2(f, 3, 'haar');
>> approx = wavecopy('a', c1, s1)
```

```
approx =
        260.0000
>> horizdet2 = wavecopy('h', c1, s1, 2)
horizdet2 =
      1.0e-013*
                  0            -0.2842
                  0                  0
>> [newc1, horizdet2] = wavecut('h', c1, s1, 2);
>> newhorizdet2 = wavecopy('h', newsc1, s1, 2)
newhorizdet2 =
        0         0
        0         0
```

7.3.2 Displaying Wavelet Decomposition Coefficients

The coefficients that are packed into one-dimensional wavelet decomposition vector c are, in reality, the coefficients of the two-dimensional output arrays from the filter bank in Fig.7.2.

For each iteration of the filterbank, four quarter-size coefficient arrays(neglecting any expansion that result from the convolution process) are produced. They can be arranged as a 2*2 array of sub-matrices that replace the two dimensional input from which they are derived.

Function wave2gray performs this subimage composting– and both scales the coefficients to better reveal their differences and inserts borders delineate the approximation and various horizontal, vertical, and diagonal detail matrices.

function w = wave2gray(c, s, scale, border)

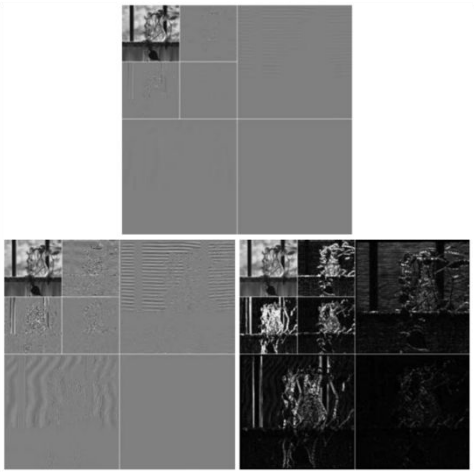
EXAMPLE 7.6: Transform coefficient display using wave2gray.

The following sequence of commands computes the two-scale DWT of the image in Fig.7.4 with respect to fourth-order Daubechies' wavelets and displays the resulting coefficients:

```
>> f = imread( 'vase.tif' );
>> [c, s] = wavefast(f, 2, 'db4');
>> wave2gray(c, s);
>> figure; wave2gray(c, s, 8);
```

>> figure; wave2gray(c, s, -8);

The images generated by the final three command lines are shown in Fig.7.5(a) Through (c), respectively.



a
b c

FIGURE 7.5
Displaying a two-scale wavelet transform of the image in Fig. 7.4: (a) Automatic scaling; (b) additional scaling by 8; and (c) absolute values scaled by 8.

7.4 The Inverse Fast Wavelet Transform

Like its forward counterpart, the inverse fast wavelet transform can be computed iteratively using digital filters. Figure 7.6 shows the required synthesis or reconstruction filter bank. At each iteration, four scale j approximation and detail subimages are upsampled and convolved with two one-dimension filters– one operating on the subimages' columns and the other on its rows.

Addition of the results yields the scale j+1 approximation, and the process is repeated until the original image is reconstructed.

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

g = waverec2(C, S, wname)

The required reconstruction filters can be alternately supplied via syntax

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

The following custom routine, which we call waveback, is the final function needed to complete our wavelet-based package for processing images in conjunction with IPT.

function [varargout] = waveback(c, s, varargin)

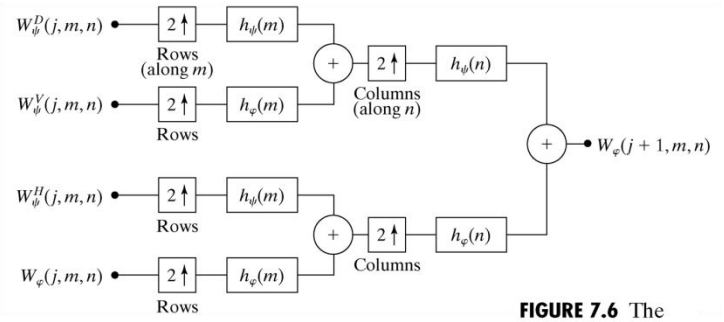


FIGURE 7.6 The 2-D FWT^{-1} filter bank. The boxes with the up arrows represent upsampling by inserting zeroes between every element.

EXAMPLE 7.7: Comparing the execution times of waveback and waverec2.

The following test routine compares the execution times of Wavelet Toolbox function waverec2 and custom function waveback using a simple modification of the test function in Example 7.3:

function [ratio, maxdiff] = ifwtcompare(f, n, wname)

For a five scale transform of the 512*512 image in Fig.7.4 with respect to –4th-order Daubechies' wavelets, we get

>> [ratio, maxdifference] = ifwtcompare(f, 5, 'db4')

>> f = imread('Vase', 'tif');

ratio = 1.0000

Maxdifference = 3.6948e-013

7.5 Wavelets in Image Processing

The basic approach to wavelet-based image processing is to:

1. Compute the two-dimensional wavelet transform of an image.

2. Alter the transform coefficients.

3. Compute the inverse transform.

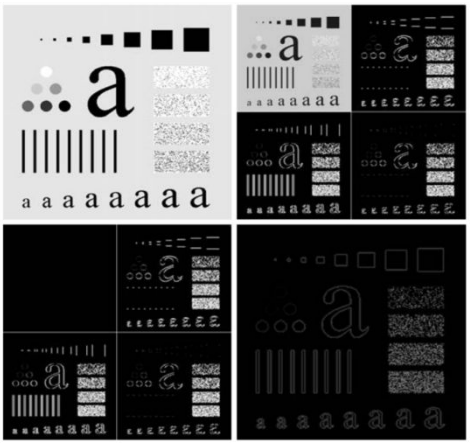
In this section, we use the preceding three-step procedure to give several examples of the use of wavelets in image processing.

EXAMPLE 7.8: Wavelet directionality and edge detection.

Consider the 500*500 test image in Fig.7.7(a). We use it to demonstrate the directional sensitivity of the 2-D wavelet transform and its usefulness in edge

detection:

```
>> f = imread('A.tif');
>> imshow(f);
>> [c, s] = wavefast(f, 1, 'sym4');
>> figure; wave2gray(c, s, -6);
>> [nc, y] = wavecut('a', c, s);
>> figure; wave2gray(nc, s, -6);
>> edges = abs(waveback(nc, s, 'sym4'));
>> figure; imshow(mat2gray(edges));
```



a b
c d

FIGURE 7.7 Wavelets in edge detection: (a) A simple test image; (b) its wavelet transform; (c) the transform modified by zeroing all approximation coefficients; and (d) the edge image resulting from computing the absolute value of the inverse transform.

EXAMPLE 7.9: Wavelet-based image smoothing or blurring.

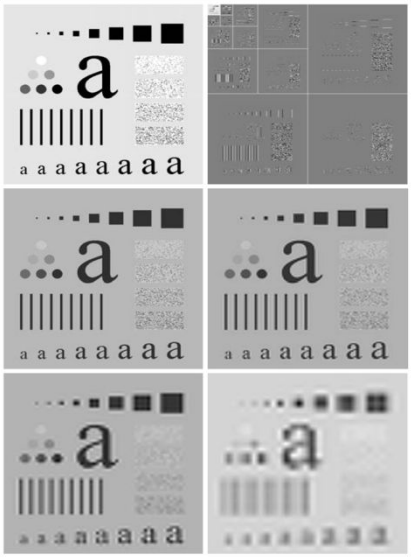
Wavelets are effective instruments for smoothing or blurring images. Consider again the test image of Fig.7.7(a). To streamline the smoothing process, we employ the following utility function:

function [nc, g8] = wavezero(c, s, l, wname)

Using wavezero, a series of increasingly smoothed versions of Fig.7.8(a) can be generated with the following sequence of commands:

```
>> f = imread('A.tif');
>> [c, s] = wavefast(f, 4, 'sym4');
>> wave2gray(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’);



a b
c d
e f

FIGURE 7.8
Wavelet-based image smoothing: (a) A test image; (b) its wavelet transform; (c) the inverse transform after zeroing the first-level detail coefficients; and (d) through (f) similar results after zeroing the second-, third-, and fourth-level details.

EXAMPLE 7.10: Progressive reconstruction.

Here, we deviate from the three-step procedure described at the beginning of this section and consider an application without a Fourier domain counterpart.

Each image in the database is stored as a multiscale wavelet decomposition.

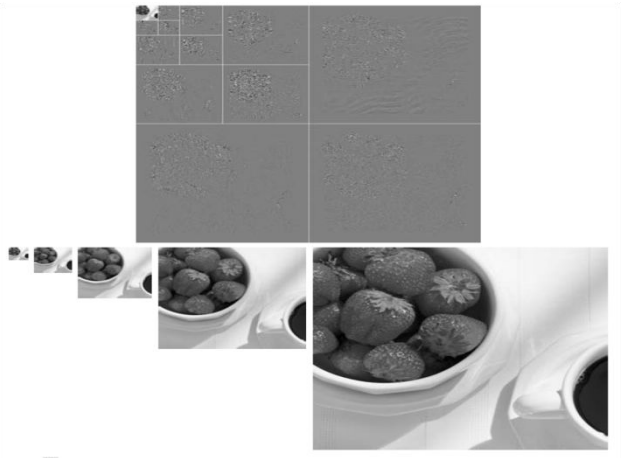
This structure is well suited to progressive reconstruction applications. For the four-scale transform of this example, the decomposition vector is

[**A4**(:)’ **H4**(:)’ … **Hi**(:)’ **Vi**(:)’ **Di**(:)’ … **V1**(:)’ **D1**(:)’]

This progressive reconstruction process is easily simulated using the following MATLAB command sequence:

```
>> f = imread(‘Strawberries.tif’);        % Generate transform
>> [c, s] = wavefast(f, 4, ‘jpeg9.7’);
>>    wave2gray(c, s, 8);
 >> f = wavecopy(‘a’, c, s);              % Approximation 1
>> figure; imshow(mat2gray(f));
>> [c, s] = waveback(c, s, ‘jpeg9.7’, 2);      % Approximation 2
>> f = wavecopy(‘a’, c, s);
>> figure; imshow(mat2gray(f));
>> [c, s] = waveback(c, s, ‘jpeg9.7’, 3);       % Approximation 3
>> f = wavecopy(‘a’, c, s);
>> figure; imshow(mat2gray(f));
```

```
>> [c, s] = waveback(c, s, 'jpeg9.7', 4);      % Approximation 4
>> f = wavecopy('a', c, s);
>> figure; imshow(mat2gray(f));
>> [c, s] = waveback(c, s, 'jpeg9.7', 1);      %   Final image
>> f = wavecopy('a', c, s);
>> figure; imshow(mat2gray(f));
```



a
b c d e f

FIGURE 7.9 Progressive reconstruction: (a) A four-scale wavelet transform; (b) the fourth-level approximation image from the upper-left corner; (c) a refined approximation incorporating the fourth-level details; (d) through (f) further resolution improvements incorporating higher-level details.

Summary

1. Introduces the wavelet transform and its use in image processing. Like the Fourier transform, wavelet transforms can be used in tasks ranging from edge detection to image smoothing.

2. Because they provide significant insight into both an image's spatial and frequency characteristics, wavelets can also be used in applications in which Fourier methods are not well suited, like progressive image reconstruction.

3. Because the Image Processing Toolbox does not include routines for computing or using wavelet transforms, a significant portion of this chapter is devoted to the development of a set of functions that extend the Image Processing Toolbox to wavelet-based imaging.

4. The functions developed were designed to be fully compatible with MATLAB's Wavelet Toolbox, which is introduced in this chapter but is not a part of the Image Processing Toolbox.

Chapter 8 Image Compression

Preview

Image compression addresses the problem of reducing the amount of data required to represent a digital image. Compression is achieved by the removal of one or more of three basic data redundancies:

(1) coding redundancy;

(2) interpixel redundancy;

(3) psychovisual redundancy.

In this chapter

1). Examine each of these redundancies,

2). Describe a few of the many techniques that can be used to exploit them

3). Examine two important compression standards—JPEG and JPEG 2000.

8.1 Background

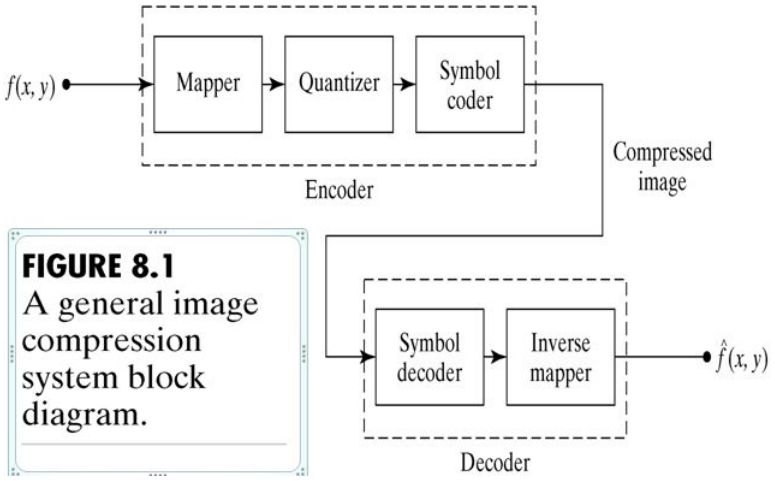


FIGURE 8.1 A general image compression system block diagram.

8-1

Image compression system are composed of two distinct structural blocks:an encoder and a decoder.

compression ratio: CR=$n1/n2$,

where n1 and n2 denote the number of information carrying units in the original and encoded images respectively.

In the first stage of the encoding process, the mapper transforms the input image into a format designed to reduce interpixel redundancies. The second stage, or quantizer block, reduces the accuracy of the mapper's output in accordance with a predefined fidelity criterion-attempting to eliminate only psychovisually redundant data. It is irreversible. The third stage of the process,a symbol coder creates a code for the quantizer output and maps the output in accordance with the code.

function cr = imratio (f1, f2), imratio compute the ratio of the bytes in two images/variables.

For example, the compression of the JPEG encoded image in fig.2.4(c) of chapter 2 can be computed via

r = imratio(imread('bubbles25.jpg'),'bubbles25.jpg')

r = 35.1612

Information preserving or lossless; lossy compression

Define e(x,y) between f(x,y) and f^(x,y):

e(x,y) = f^(x,y)- f(x,y)

$$e_{rms} = [\frac{1}{MN}\sum_{x=0}^{M-1}\sum_{y=0}^{N-1}[\hat{f}(x,y)-f(x,y)]^2]^{1/2}$$

The following M-function computes erms and displays both e(x,y) and its histogram.

function rmse = compare(f1, f2, scale)

%compare computes and displays the error between two matrices.

8.2 coding redundancy

$$p_r(r_k) = n_k/n \quad k = 1,2,\ldots\ldots, L$$

$p_r(r_k)$ represent the gray level of an L-gray-level image. n_k is the number of times that the *k*th gray level appears in the image . n is the total number of pixels in the image.

the average number of bits required to represent each pixel is

$$L_{avg} = \sum_{k=1}^{l} l_2(k)p_r(r_k)$$

Table 8.1

TABLE 8.1
Illustration of coding redundancy: $L_{avg} = 2$ for Code 1; $L_{avg} \simeq 1.81$ for Code 2.

r_k	$p_r(r_k)$	Code 1	$l_1(r_k)$	Code 2	$l_2(r_k)$
r_1	0.1875	00	2	011	3
r_2	0.5000	01	2	1	1
r_3	0.1250	10	2	010	3
r_4	0.1875	11	2	00	2

$$L_{avg} = \sum_{k=1}^{l} l_2(k) p_r(r_k)$$

=3(0.1875)+1(0.5)+3(0.125)+2(0.1875) = 1.8125

function h = entropy(x,n); %entropy computes a first-order estimate of the entropy of a matrix.

Example 8.1

Computing first-order entropy estimates.

Consider a simple 4*4 image whose histogram models the symbol probabilities in Table 8.1.

```
>>f = [119 123 168 119; 123 119 168 168];
>>f = [f ; 119 119 107 119; 107 107 119 119]
p = hist(f(:),8);
p = p / sum(p)
p =   0.1875    0.5    0.125    0    0    0    0    0.1875
h = entropy(f)
h = 1.7806
```

8.2.1 Huffman codes

Huffman codes contain the smallest possible number of code symbols per source symbol subject to the constraint that the source symbols are coded one at a time(the first step,….the second step……).

Original Source		Source Reduction	
Symbol	Probability	1	2
a_2	0.5	0.5	0.5
a_4	0.1875	0.3125	0.5
a_1	0.1875	0.1875	
a_3	0.125		

a
b

FIGURE 8.2 Huffman (a) source reduction and (b) code assignment procedures.

Original Source			Source Reduction			
Symbol	Probability	Code	1		2	
a_2	0.5	1	0.5	1	0.5	1
a_4	0.1875	00	0.3125	01	0.5	0
a_1	0.1875	011	0.1875	00		
a_3	0.125	010				

8-2

The source reduction and code assignment procedures just described are implemented by the following M-function, which we call huffman:

function CODE = huffman(p)

huffman builds a variable-length Huffman code for a symbol source.

The following command line sequence uses huffman to generate the code in Fig.8.2:

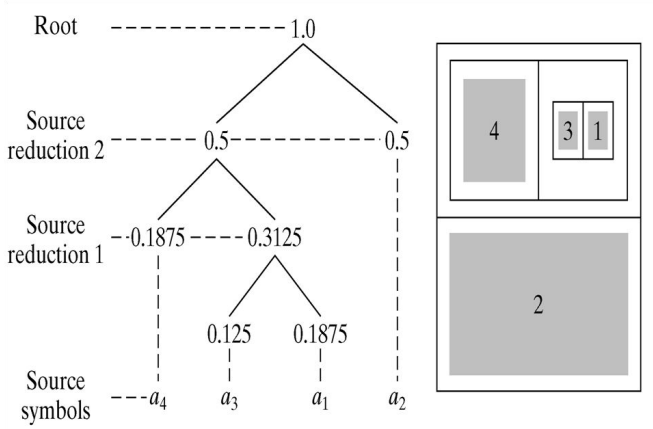
```
>>p = [0.1875 0.5 0.125 0.1875]
>>c = huffman(p)
c =     '011'
        '1'
        '010'
        '00'
```

global X Y Z: This statement makes variables X, Y, and Z available to the function in which they are declared. In huffman, CODE is initialized using the cell function, whose syntax is

x = cell(m, n)

It creates an m*n array of empty matrices that can be referenced by cell or by content. In each iteration of the loop, vector p is sorted in ascending order of probability. This is done by the sort function, whose syntax is

[y, i] = sort (x).



a b c

FIGURE 8.3 Source reductions of Fig. 8.2(a) using function `huffman`: (a) binary tree equivalent; (b) display generated by `cellplot(s)`; (c) `celldisp(s)` output.

8-3

Figure 8.3 shows the final output of the process for the symbol probabilities in table 8.1 and fig.8.2(a). Figure 8.3(b) and(c) were generated by inserting between the last two statements of the huffman main routine.

(1)Each two-way branch in the tree corresponds to a two-element cell array in s;

(2)Each two-element cell array contains the indices of the symbols that were merged in the corresponding source reduction.

Table 8.2 details the sequence of makecode calls that results for the source reduction cell array in fig.8.3.

Table 8.2

Call	Origin	sc	codeword
1	main routine	{1x2 cell} [2]	[]
2	makecode	[4] {1x2 cell}	0
3	makecode	4	0 0
4	makecode	[3] [1]	0 1
5	makecode	3	0 1 0
6	makecode	1	0 1 1
7	makecode	2	1

TABLE 8.2 Code assignment process for the source reduction cell array in Fig. 8.3.

8.2.2 Huffman Encoding

Example 8.2: variable-length code mappings in MATLAB.

Consider the simple 16-bytes 4*4 image:

```
>>f2 = unit8([2 3 4 2 ; 3 2 4 4 ; 2 2 1 2 ; 1 1 2 2])
f2 =    2   3   4   2
        3   2   4   4
        2   2   1   2
        1   1   2   2
>>whos('f2')
      name        size        bytes        class
       f2          4*4          16        unit8 array
Grand total is 16 elements using 16 bytes
>>c = huffman(hist(double(f2(:)), 4)
c =     '011'
        '1'
        '010'
        '00'
```

function y = mat2huff(x); % mat2huff huffman encodes a matrix.

Example 8.3: encoding with *mat2huff* to illustrate further the compression performance of Huffman encoding, consider the 512*512 8-bit monochrome image of fig 8.4(a). The compression of the image using *mat2huff* is carried out by the following command sequence:

```
>>f = imread('Tracy.tif');
>>c = mat2huff(f);
>>cr1 = imratio(f, c);
cr1 = 1.2191
```

We write it to disk using the save function:

```
>>save SqueezeTracy c;
>>cr2 = imratio('Tracy.tif','SqueezeTracy.mat')
cr2 =   1.2365
```

a b

FIGURE 8.4 A 512 × 512 8-bit monochrome image of a woman and a close-up of her right eye.

8.2.3 Huffman decoding

function x = *huff2mat*(y), can be broken into five basic steps:

1.Extract dimension m and n, and minimum value xmin (of eventual output x) from input structure y.

2.Re-create the Huffman code that was used to encode x by passing its histogram to function huffman. The generated code is called map in the listing.

3.Build a data structure (transition and output table link) to streamline the decoding of the encoded data in y. code through a series of computationally efficient binary searches.

4.Pass the data structure and the encoded data to C function unravel. This function minimizes the time required to perform the binary searches, creating decoded output vector x of class double.

5.Add xmin to each element of x and reshape it to match the dimensions of the original x.

function x = huff2mat(y)

huffman decodes a Huffman encoded matrix.

Table 8.3

Index i	Value in link(i)
1	2
2	4
3	−2
4	−4
5	6
6	−3
7	−1

TABLE 8.3 Decoding table for the source reduction cell array in Fig. 8.3.

Table 8.3 shows the link table that is generated for the Huffman code in Example 8.2. If each link index is viewed as a decoding state,*i*, each binary coding decision and/or Huffman decoded output is determined by link(*i*):

1.If link(*i*) < 0, a Huffman code word has been decoded. The decoded output is |link|, where | | denotes the absolute value.

2.If link(*i*) > 0 and the next encoded bit to be processed is a 0, the next decoding state is index link(*i*). That is, we let *i* = link(*i*).

If link(*i*) > 0 and the next encoded bit to be processed is a 1, the next decoding state is index link(*i*)+1. That is, *i* = link(*i*)+1.

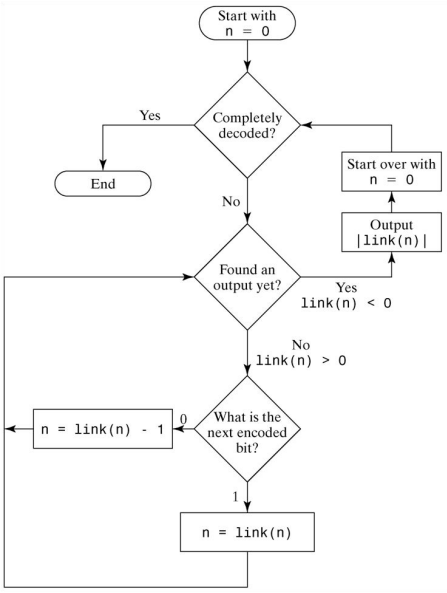


FIGURE 8.5
Flow diagram for C function `unravel`.

A MATLAB external function produced from C or Fortran source code. It has a platform dependent extension. For example, we type

>>mex unravel.c

A MEX-file named unravel.dll with extension .dll will be created.

Like all C MEX-files, C MEX-file unraval. C consists of two distinct parts: a computational routine and a gateway routine. The computational routine, also named unravel, contains the C code that implements the link-based decoding process of fig.8.5. The gateway routine, which must always be named mexFunction, interfaces C computational routine unravel to M-file calling function, huff2mat.

Example 8.4 decoding with huff2mat

The Huffman encoded image of example 8.3 can be decoded with the following sequence of commands:

>>load SqueezeTracy;

>>g = huff2mat(c);

>>f = imread('Tracy.tif');

>>rmse = compare(f, g)

rmse = 0

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

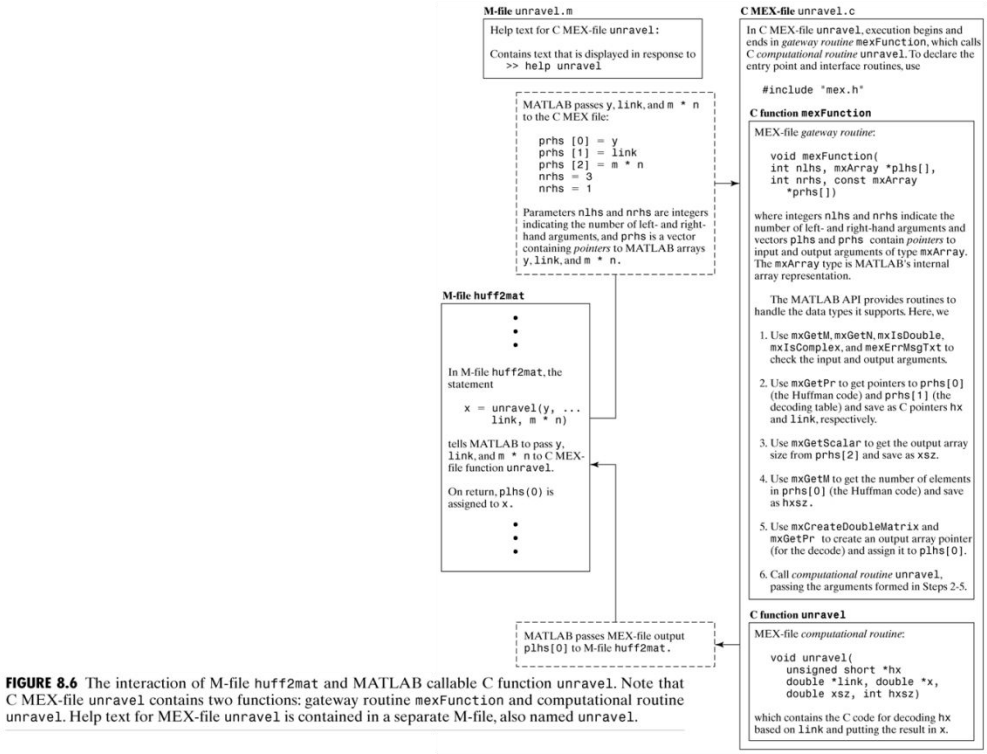


FIGURE 8.6 The interaction of M-file `huff2mat` and MATLAB callable C function `unravel`. Note that C MEX-file `unravel` contains two functions: gateway routine `mexFunction` and computational routine `unravel`. Help text for MEX-file `unravel` is contained in a separate M-file, also named `unravel`.

8.3 Interpixel Redundancy

As fig 8.7 (b) and (d) show, they have virtually identical histograms.

>>f1 = imread('Random Matches.tif');

>>c1 = mat2huff(f1);

>>entropy(f1)

ans = 7.4253

>>imratio(f1, c1)

ans = 1.0704

>>f2 = imread('Aligned Matches.tif');

>>c2 = mat2huff(f2);

>>entropy(f2)

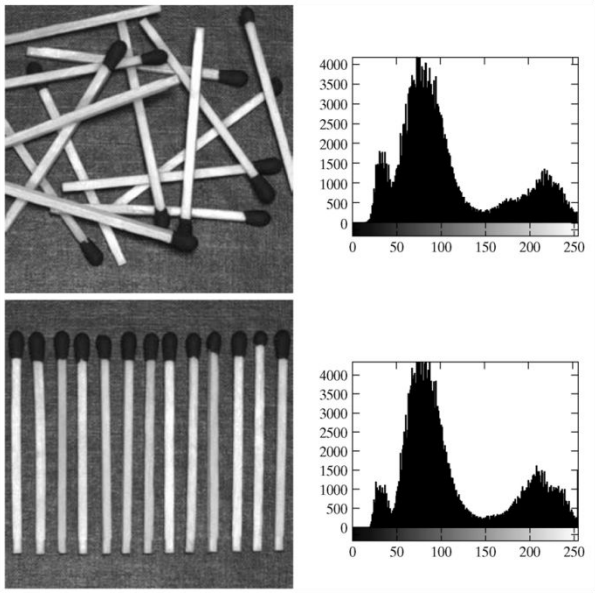
ans = 7.3505

>>imratio(f2, c2)

ans = 1.0821

A simple mapping procedure is illustrated in fig.8.8.

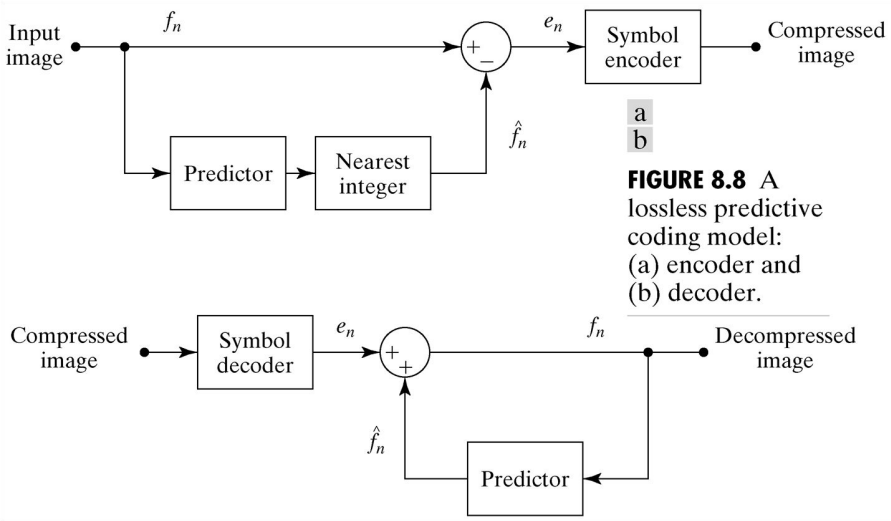
prediction error: en=fn-f^n



a b
c d

FIGURE 8.7 Two images and their gray-level histograms.

8-7



a
b

FIGURE 8.8 A lossless predictive coding model: (a) encoder and (b) decoder.

8-8

function y = mat2lpc(x, f)

mat2lpc compresses a matrix using 1-D lossless predictive coding.

function x = lpc2mat(y, f)

lpc2mat decompresses a 1-D lossless predictive encoded matrix.

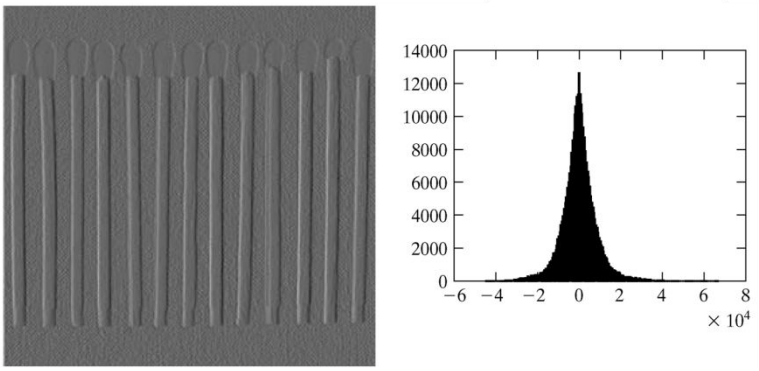
Example 8.5 lossless predictive coding

Consider encoding the image of fig.8,7(c) using the simple first-order linear predictor.

f^(x,y) = round[af(x, y - 1)]

Fig.8.9(a) shows the prediction error image that result with a = 1.

```
>>f = imread('Aligned Matched.tif');
>>e = mat2lpc(f);
>>imshow(mat2gray(e));
>>entropy(e)
ans =    5.9727
>>c = mat2huff(e);
>>cr = imratio(f, c)
cr =     1.3311
>>[h, x] = hist(e(:) * 512, 512);
>>figure; bar(x, h, 'k');
>>g = lpc2mat(huff2mat(c));
>>compare(f, g)
ans = 0
```



a b

FIGURE 8.9
(a) The prediction error image for Fig. 8.7(c) with `f = [1]`.
(b) Histogram of the prediction

8.4 Psychovisual Redundancy

Psychovisual Redundancy is associated with real or quantifiable visual information.

Example 8.6: compression by quantization.

Consider the image in fig 8.10. Figure 8.10(a) shows a monochrome image with 256 gray level. Figure 8.10(b) is the same image after uniform quantization to four bits or 16 possible levels. The resulting compression ratio is 2:1. Figure 8.10(c) illustrates the significant improvements possible with quantization that takes advantage of the peculiarities of the human visual system.

The method used to produce fig.8.10(c) is called improved gray-scale quantization.

a b c
FIGURE 8.10
(a) Original image.
(b) Uniform quantization to 16 levels. (c) IGS quantization to 16 levels.

quantize

function y = quantize(x, b, type)

%quantizes the elements of a UINT8 matrix.

Example 8.7: Combining IGS quantization with lossless predictive and Huffman coding. The following sequence of command combines IGS quantization, lossless predictive coding, and Huffman coding to compress the image of fig.8.10(a) to less than a quarter of its original size:

>>f = imread('Brushes.tif');

>>q = quantize(f, 4, 'igs');

>>qs = double(q) / 16;

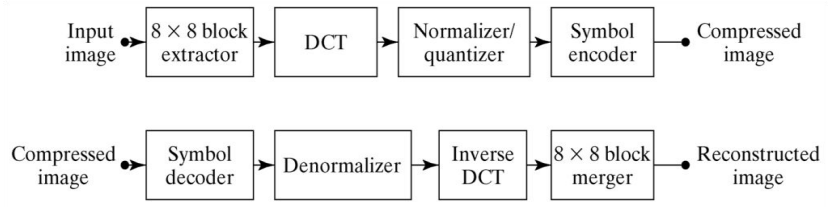
>>e = mat2lpc(qs);

```
>>c = mat2huff(e);
>>imratio(f, c)
ans = 4.1420
>>ne = huff2mat(c);
>>nqs = lpc2mat(ne);
>>nq = 16*nqs;
>>compare(q, nq);
ans = 0
>>rmse = compare(f, nq)
rmse = 6.8382
```

8.5 JPEG Compression

8.5.1 JPEG JPEG(for Joint Photographic Expert Group)

In the JPEG baseline coding system, which is based on the discrete cosine transform and limited to 8 bits, while the quantized DCT coefficient values are restricted to 11 bits.



a
b

FIGURE 8.11
JPEG block diagram:
(a) encoder and
(b) decoder.

T^(u, v) = [T(u, v)/Z(u, v)]

T^(u, v) for u, v = 0, 1, …., 7 are the resulting normalized and quantized coefficients,T(u, v) is the DCT of an 8*8 block of image f(x, y), and Z(u, v) is a transform normalization array like that of fig.8.12(a).

16	11	10	16	24	40	51	61
12	12	14	19	26	58	60	55
14	13	16	24	40	57	69	56
14	17	22	29	51	87	80	62
18	22	37	56	68	109	103	77
24	35	55	64	81	104	113	92
49	64	78	87	103	121	120	101
72	92	95	98	112	100	103	99

0	1	5	6	14	15	27	28
2	4	7	13	16	26	29	42
3	8	12	17	25	30	41	43
9	11	18	24	31	40	44	53
10	19	23	32	39	45	52	54
20	22	33	38	46	51	55	60
21	34	37	47	50	56	59	61
35	36	48	49	57	58	62	63

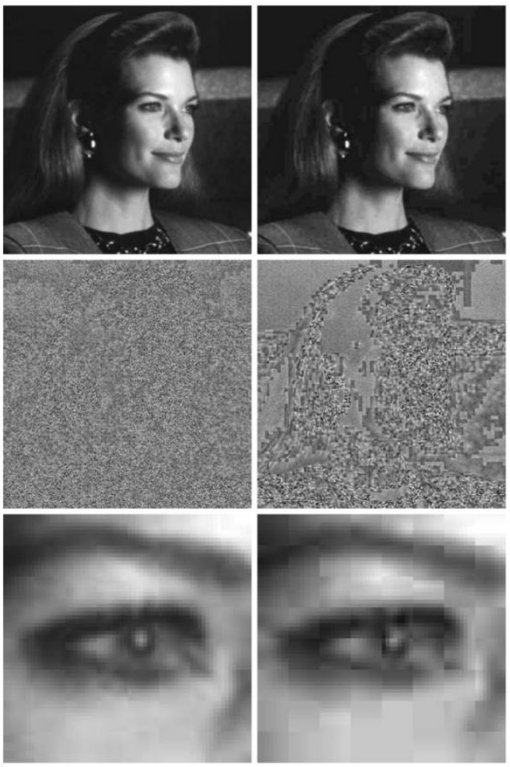
a b

FIGURE 8.12 (a) The default JPEG normalization array. (b) The JPEG zigzag coefficient ordering sequence.

8-12

function y = im2jpeg(x, quality)

im2jpeg compressed an image using a JPEG approximation.



a b
c d
e f

FIGURE 8.13 Left column: Approximations of Fig. 8.4 using the DCT and normalization array of Fig. 8.12(a). Right column: Similar results with the normalization array scaled by a factor of 4.

8-13

Example 8.8: JPEG compression

Fig.8.13 (a) and (b) show two JPEG coded and subsequently decoded approximations of the monochrome image in Fig.8.4(a).

```
>>f = imread('Tracy.tif');
>>c1 = im2jpeg(f);
>>f1 = jpeg2im(c1);
>>imratio(f, c1)
```

ans = 18.2450

>>compare(f, f1, 3)

ans = 2.4675

>>c4 = im2jpeg(f, 4);

>>f4 = jpeg2im(c4);

>>imratio(f, c4)

Ans =

41.7862

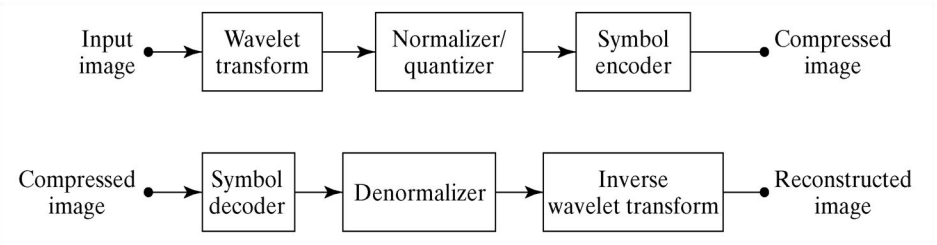
>>compare(f, f4, 3)

Ans =

4.4184

8.5.2 JPEG 2000

Figure 8.14 show a simplified JPEG 2000 coding system (absent several optional operation).



a
b

FIGURE 8.14
JPEG 2000 block diagram:
(a) encoder and
(b) decoder.

function y = im2jpeg2k(x, n, q)

im2jpeg2k compresses an image using a JPEG 2000 approximation.

Example 8.9:JPEG 2000 compression

>>f = imread('tracy.tif');

>>c1 = im2jpeg2k(f, 5, [8 8.5]);

>>f1 = jpeg2k2im(c1);

```
>>rms1 = compare(f, f1)
rms1 = 3.6931
>>cr1 = imratio(f, c1)
cr1 =    42.1589
>>c2 = im2jpeg2k(f, 5, [8    7]);
>>f2 = jpeg2k2im(c2);
>>rms2 = compare(f, f2)
rms2 = 5.9172
>>cr2 = imratio(f, c1)
cr2 = 87.7323
>>c3 = im2jpeg2k(f, 1, [1 1 1 1]);
>>f3 = jpeg2k2im(c3);
>>rms3 = compare(f, f3)
rms3 =
        1.1234
>>cr3 = imratio(f, c3)
cr3 =
         1.6350
```

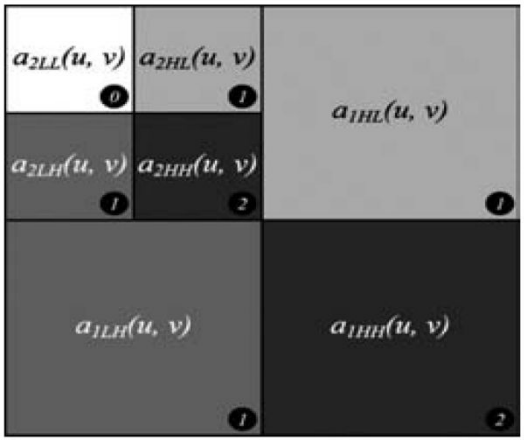
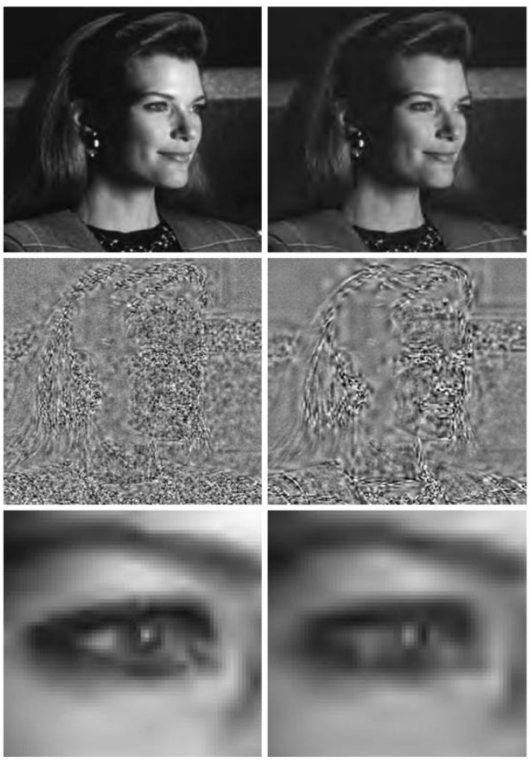


FIGURE 8.15 JPEG 2000 two-scale wavelet transform coefficient notation and analysis gain (in the circles).



a b
c d
e f

FIGURE 8.16 Left column: JPEG 2000 approximations of Fig. 8.4 using five scales and implicit quantization with $\mu_0 = 8$ and $\varepsilon_0 = 8.5$. Right column: Similar results with $\varepsilon_0 = 7$.

Summary

The material in this chapter introduces the fundamentals of digital image compression through the removal of coding, interpixel, and psychovisual redundancy. Matlab procedures are developed.

An overview of the popular JPEG and JPEG2000 image compression standards is given.

Chapter9 Morphological Image Processing

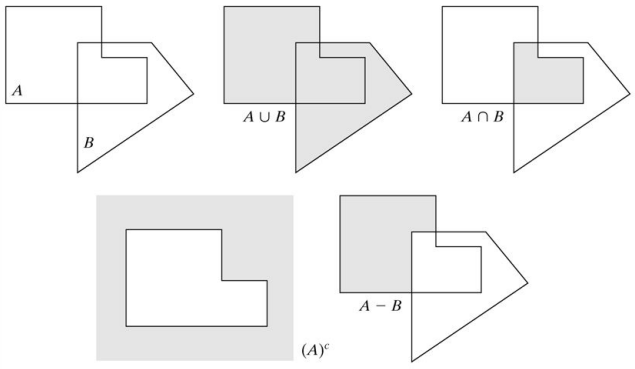
The word morphology commonly denotes a branch of biology that deals with the form and structure of animals and plants.

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.

Morphological techniques for pre- or post processing, such as morphological filtering, thinning, and pruning. Morphology is a cornerstone of the mathematical set of tools underlying the development of techniques that extract "meaning" from an image.

9.1 Preliminaries（set theory)

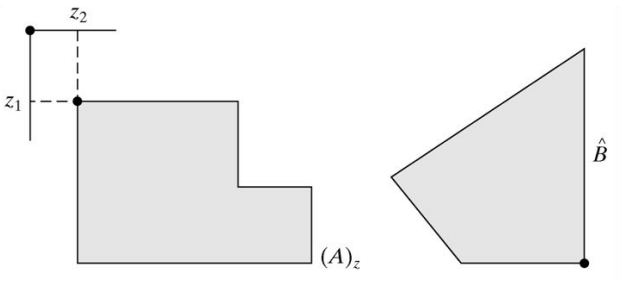
1. $w \in A$
2. $w \notin A$
3. $B = \{w \mid \text{condition}\}$
4. $A^c = \{w \mid w \notin A\}$ Complement of A
5. $C = A \cup B$ union of two sets
6. $C = A \cap B$ intersection of two sets
7. $A - B = \{w \mid w \in A, w \notin B\}$ difference of two sets
8. $\hat{B} = \{w \mid w = -b, \text{for } b \in B\}$ reflection of set B
9. $(A)_z = \{c \mid c = a + z, \text{for } a \cup A\}$ translation of set A



a b c
d e

FIGURE 9.1
(a) Two sets A and B. (b) The union of A and B. (c) The intersection of A and B. (d) The complement of A. (e) The difference between A and B.

9- 1



a b

FIGURE 9.2
(a) Translation of A by z. (b) Reflection of B. The sets A and B are from Fig. 9.1.

9- 2

Morphological theory views a binary image as the set of its foreground (1-valued) pixels, the elements of which are in Z2(Z is the set of integers). Set operations such as union and intersection can be applied directly to binary image sets.

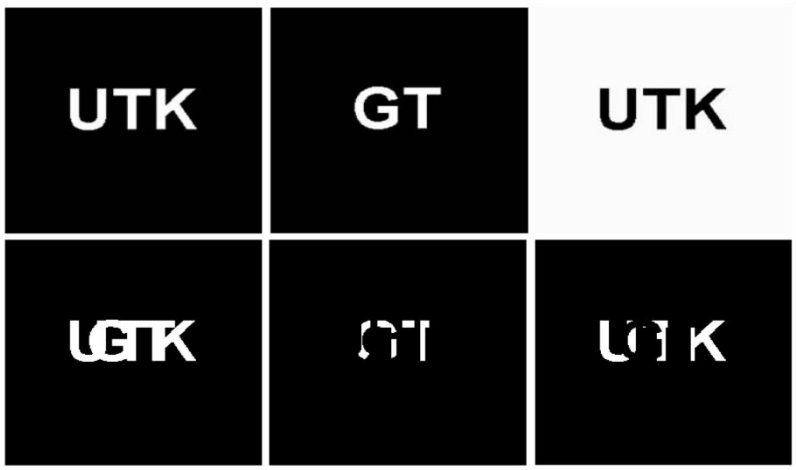
The set operations defined in Fig. 9.1 can be performed on binary images using MATLAB's logical operators OR (|), AND (&), and NOT (~), as Table 9.1 shows.

Table 9.1

Set Operation	MATLAB Expression for Binary Images	Name
$A \cap B$	A & B	AND
$A \cup B$	A | B	OR
A^c	~A	NOT
$A - B$	A & ~B	DIFFERENCE

TABLE 9.1
Using logical expressions in MATLAB to perform set operations on binary images.

As a simple illustration, Fig. 9.3 shows the results of applying several logical operators to two binary images containing text.



a b c
d e f

FIGURE 9.3 (a) Binary image A. (b) Binary image B. (c) Complement ~A. (d) Union A | B. (e) Intersection A & B. (f) Set difference A & ~B.

9.2 Dilation and Erosion

The operations of dilation and erosion are fundamental to morphological image processing. Dilation is an operation that "grows" or "thickens" objects in a binary image. The specific manner and extent of this thickening is controlled by a shape referred to as a structuring element.

Dilation

定义A用B结构单元扩张记作$A \oplus B$，定义为

$A \oplus B = \left\{ c \subset E^N, c = a + b, \forall a \subset A, \forall b \subset B \right\}$或

$A \oplus B = \left\{ B_a, \forall a \subset A \right\}$

例：$A = \left\{ (0,1), (1,1), (2,1), (2,2), (3,0) \right\}$

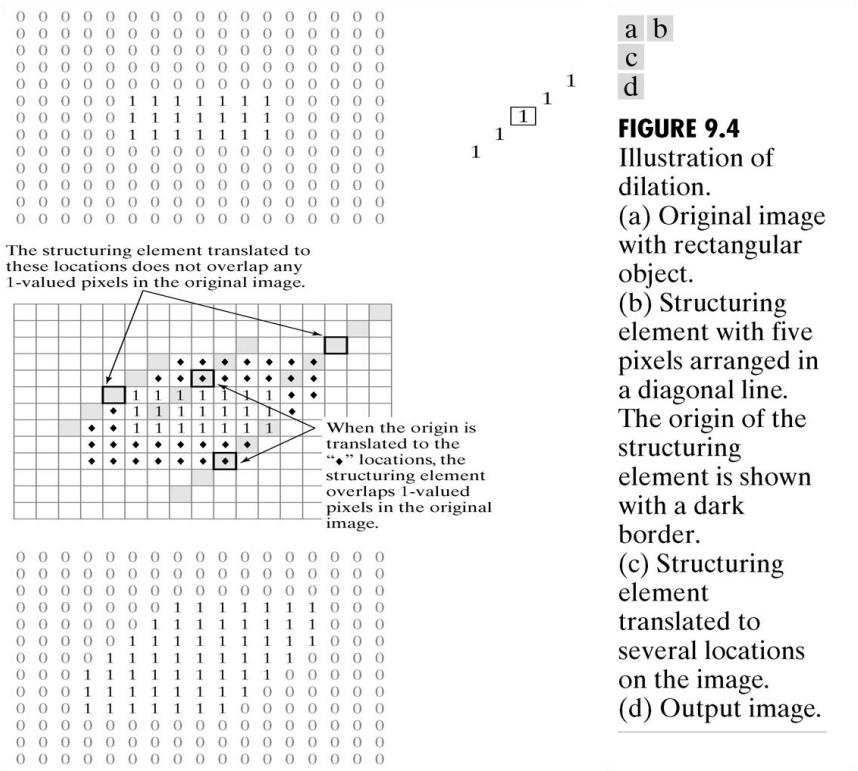
$B = \left\{ (0,0), (0,1) \right\}$

则$A \oplus B =$

$\left\{ (0,1), (1,1), (2,1), (2,2), (3,0), (0,2), (1,2), (2,2), (2,3), (3,1) \right\}$

$A \oplus B$的意义A用B扩张，

即所有A的点集使B_a击中A且交集非零。



a b
c
d

FIGURE 9.4 Illustration of dilation. (a) Original image with rectangular object. (b) Structuring element with five pixels arranged in a diagonal line. The origin of the structuring element is shown with a dark border. (c) Structuring element translated to several locations on the image. (d) Output image.

Figure 9.4 illustrates how dilation works.

Figure 9.4(a) shows a simple binary image containing a rectangular object.

Figure 9.4(b) is a structuring element, a five-pixel-long diagonal line in this case. Figure 9.4(b) shows the origin of the structuring element using a black outline.

Figure 9.4(c) graphically depicts dilation as a process that translates the origin of the structuring element throughout the domain of the image and checks to see where it overlaps with 1-valued pixels.

Fig. 9.4(d) is 1 at each location of the origin such that the structuring element overlaps at least one 1-valued pixel in the input image.

Dilation is commutative; that is,

$A \oplus B = B \oplus A$.

It is a convention in image processing to let the first operand of $A \oplus B$ be the image and the second operand be the structuring element, which usually is much smaller than the image. We follow this convention from this point on.

IPT function imdilate performs dilation. Its basic calling syntax is

A2 = imdilate(A, B)

A = imread('broken_text.tif');

B = [0 1 0; 1 1 1; 0 1 0];

A2 = imdilate(A, B);

imshow(A2);

a b

FIGURE 9.6
A simple example of dilation. (a) Input image containing broken text. (b) Dilated image.

Dilation is associative. That is,

$$A \oplus (B \oplus C) = (A \oplus B) \oplus C$$

Suppose:

$$A \oplus B = A \oplus (B_1 \oplus B_2) = (A \oplus B_1) \oplus B_2 .$$

The associative property is important because the time required to compute dilation is proportional to the number of nonzero pixels in the structuring element. The gain in speed with the decomposed implementation is still significant.

IPT function strel constructs structuring elements with a variety of shapes and sizes. Its basic syntax is: se = strel(shape, parameters), where shape is a string specifying the desired shape, and parameters is a list of parameters that specify information about the shape.

In addition to simplifying the generation of common structuring element shapes, function strel also has the important property of producing structuring elements in decomposed form.

Function imdilate automatically uses the decomposition information to speed up the dilation process.

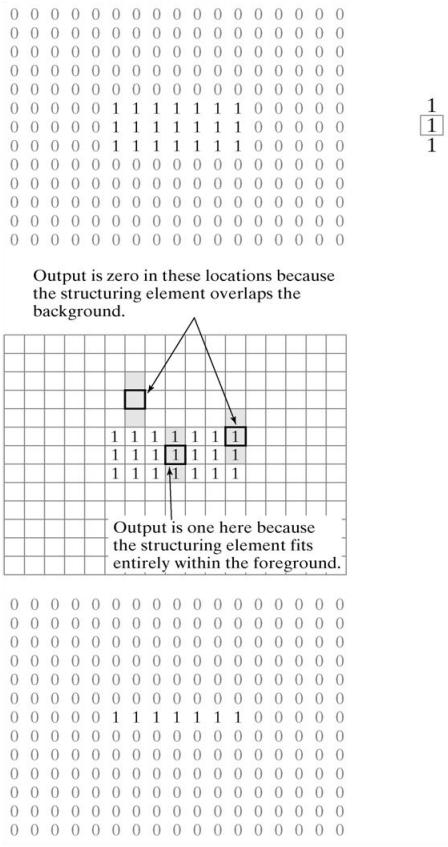
Examples:

Table 9.2

Syntax Forms	Description
`se = strel('diamond', R)`	Creates a flat, diamond-shaped structuring element, where `R` specifies the distance from the structuring element origin to the extreme points of the diamond.
`se = strel('disk', R)`	Creates a flat, disk-shaped structuring element with radius `R`. (Additional parameters may be specified for the disk; see the `strel` help page for details.)
`se = strel('line', LEN, DEG)`	Creates a flat, linear structuring element, where `LEN` specifies the length, and `DEG` specifies the angle (in degrees) of the line, as measured in a counterclockwise direction from the horizontal axis.
`se = strel('octagon', R)`	Creates a flat, octagonal structuring element, where `R` specifies the distance from the structuring element origin to the sides of the octagon, as measured along the horizontal and vertical axes. `R` must be a nonnegative multiple of 3.
`se = strel('pair', OFFSET)`	Creates a flat structuring element containing two members. One member is located at the origin. The second member's location is specified by the vector `OFFSET`, which must be a two-element vector of integers.
`se = strel('periodicline', P, V)`	Creates a flat structuring element containing `2*P + 1` members. `V` is a two-element vector containing integer-valued row and column offsets. One structuring element member is located at the origin. The other members are located at `1*V`, `-1*V`, `2*V`, `-2*V`, `...`, `P*V`, and `-P*V`.
`se = strel('rectangle', MN)`	Creates a flat, rectangle-shaped structuring element, where `MN` specifies the size. `MN` must be a two-element vector of nonnegative integers. The first element of `MN` is the number rows in the structuring element; the second element is the number of columns.
`se = strel('square', W)`	Creates a square structuring element whose width is `W` pixels. `W` must be a nonnegative integer scalar.
`se = strel('arbitrary', NHOOD)` `se = strel(NHOOD)`	Creates a structuring element of arbitrary shape. `NHOOD` is a matrix of 0s and 1s that specifies the shape. The second, simpler syntax form shown performs the same operation.

TABLE 9.2
The various syntax forms of function `strel`. (The word *flat* means that the structuring element has zero height. This is meaningful only for gray-scale dilation and erosion. See Section 9.6.1.)

Erosion "shrinks" or "thins" objects in a binary image. As in dilation, the manner and extent of shrinking is controlled by a structuring element. Figure 9.7 illustrates the erosion process.



a b
c
d

FIGURE 9.7
Illustration of erosion.
(a) Original image with rectangular object.
(b) Structuring element with three pixels arranged in a vertical line. The origin of the structuring element is shown with a dark border.
(c) Structuring element translated to several locations on the image.
(d) Output image.

Erosion

定义A用B结构单元腐蚀为$A\Theta B$,其意义为

$A\Theta B=\{c\subset E^N, c+b\subset A, \forall b\subset B\}$或

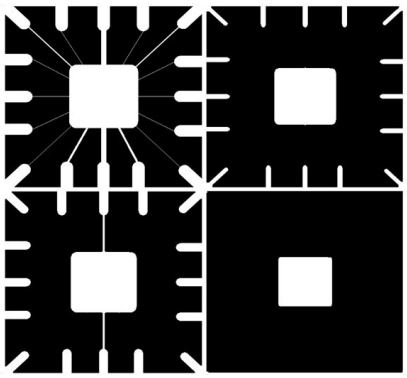
$A\Theta B=\{c, B_c\subset A\}$

例:$A=$

$\{(1,0),(1,1),(1,2),(1,3),(1,4),(1,5),(2,1),(3,1),(4,1),(5,1)\}$

$B=\{(0,0),(0,1)\}$

则$A\Theta B=\{(1,0),(1,1),(1,2),(1,3),(1,4)\}$



a b
c d

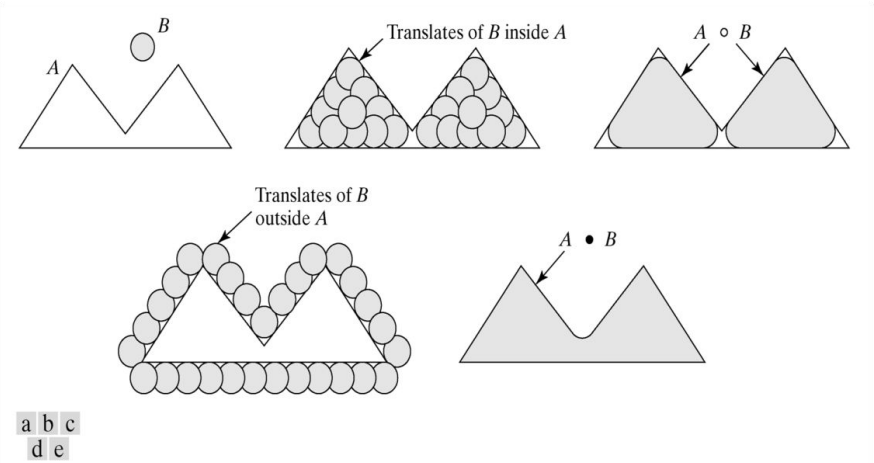
FIGURE 9.8 An illustration of erosion. (a) Original image. (b) Erosion with a disk of radius 10. (c) Erosion with a disk of radius 5. (d) Erosion with a disk of radius 20.


```
A = imread('wirebond_mask.tif');
se = strel('disk', 10);
A2 = imerode(A, se);    imshow(A2)
se = strel('disk', 5);
A3 = imerode(A, se);
imshow(A3)
A4 = imerode(A, strel('disk', 20));
imshow(A4)
```

9.3 Combining Dilation and Erosion

In practical image-processing applications, dilation and erosion are used most often in various combinations. In this section we consider three of the most common combinations of dilation and erosion: opening, closing, and the hit-or-miss transformation. We also introduce lookup table operations and discuss bwmorph, an IPT function that can perform a variety of practical morphological tasks.

9.3.1 opening and closing



a b c
d e

FIGURE 9.9 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).

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

$$A \circ B = (A\text{-}B) \oplus B$$

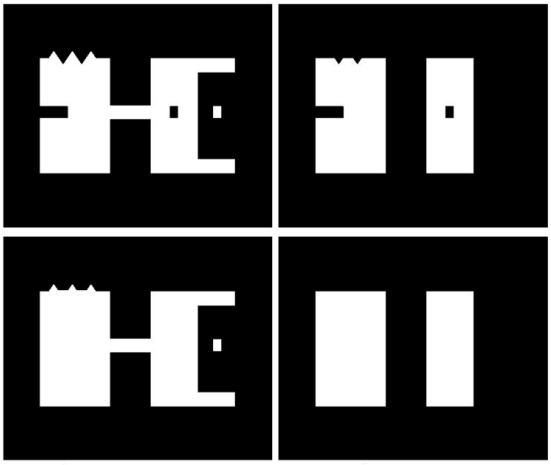
$$A \circ B = \cup \{(B)_z | (B)_Z \subseteq A\}$$

the union of all translations of B that fit entirely within A. Figure 9.9 (a,b,c) illustrates the morphological opening of A by B.

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

$$A \bullet B = (A \oplus B)\text{-}B$$

Geometrically, $A \bullet B$ is the complement of the union of all translations of B that do not overlap A. Figure 9.9(d,e) illustrates the morphological closing. Tend to smooth the contours of objects.



a b
c d

FIGURE 9.10
Illustration of opening and closing.
(a) Original image.
(b) Opening.
(c) Closing.
(d) Closing of (b).


```
f = imread('shapes.tif');
se = strel('square', 20);
fo = imopen(f, se);
imshow(fo)
fc = imclose(f, se);
imshow(fc)
foe = imclose(fo, se);
imshow(foc)
```

Figure 9.11 further illustrates the usefulness of closing and opening by applying these operations to a noisy fingerprint.

a b c

FIGURE 9.11 (a) Noisy fingerprint image. (b) Opening of image. (c) Opening followed by closing. (Original image courtesy of the National Institute of Standards and Technology.)

```
f = imread('fingerprint.tif');
se = strel('square',  3);
fo = imopen(f,  se);
imshow(fo);
foe = imclose(fo,se);
imshow(foc);
```

Morphological opening removes completely regions of an object that cannot contain the structuring element, smoothes object contours, breaks thin connections, and removes thin protrusions.

C=imopen(A, B)

Morphological closing tends to smooth the contours of objects, joins narrow breaks, fills long thin gulfs, and fills holes smaller than the structuring element.

C=imclose(A, B)

9.3.2 The Hit-or-Miss transformation

The *hit-or-miss transformation* is useful to be able to identify specified configurations of pixels, such as isolated foreground pixels, or pixels that are end points of line segments. It is useful for applications such as these.

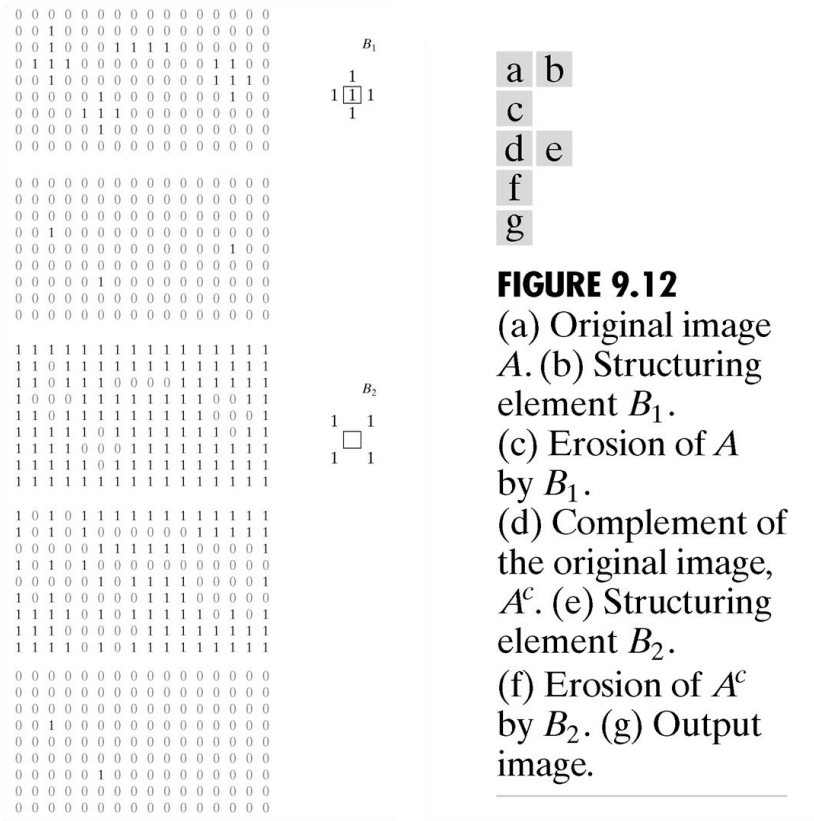
The hit-or-miss transformation of A by B is denoted $A \otimes B = (A \Theta B_1) \cap (A^c \Theta B_2)$. The hit-or-miss transformation is implemented in IPT by function bwhitmiss, which has the syntax

C = bwhitmiss(A, B1, B2)

where C is the result, A is the input image, and B1 and B2 are the structuring elements just discussed.

```
B1  = strel( [0 0 0;  0 1  1; 0 1  0] );
    B2 = strel( [1  1  1; 1  0 0; 1  0  0] );
    g = bwhitmiss(f,  B1  ,B2);
    imshow(g);
```

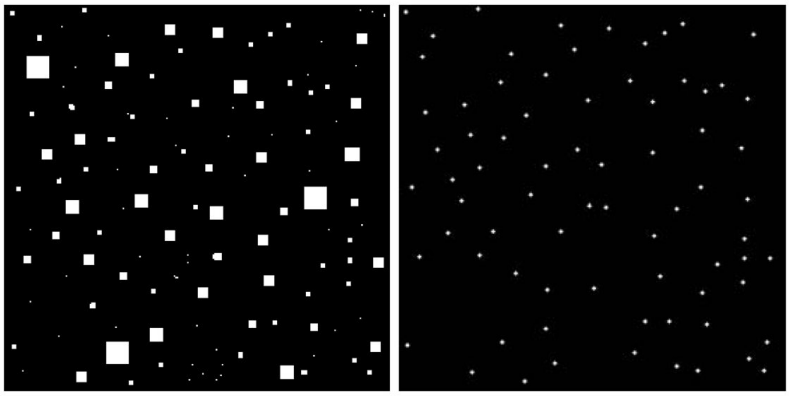
Each single-pixel dot in Fig. 9.13(b) is an upper-left-corner pixel of the objects in Fig. 9.13(a). The pixels in Fig. 9.13(b) were enlarged for clarity.



a b
c
d e
f
g

FIGURE 9.12
(a) Original image A. (b) Structuring element B_1. (c) Erosion of A by B_1. (d) Complement of the original image, A^c. (e) Structuring element B_2. (f) Erosion of A^c by B_2. (g) Output image.

9-12



a b

FIGURE 9.13
(a) Original image. (b) Result of applying the hit-or-miss transformation (the dots shown were enlarged to facilitate viewing).

9-13

9.3.3 Using lookup tables

When the hit-or-miss structuring elements are small, a faster way to compute the hit-or-miss transformation is to use a lookup table (LUT). The technique is to precompute the output pixel value for every possible neighborhood configuration and then store the answers in a table for later use.

The toolbox provides two functions, makelut and applylut (illustrated later in this

section), that can be used to implement this technique.

As an illustration, write a function, endpoints.m, that uses makelut and applylut to detect end points in a binary image. Function makelut constructs a lookup table based on a user-supplied function, and applylut processes binary images using this lookup table.

Figure 9.14 illustrates a typical use of function endpoints. Figure 9.14(a) is a binary image containing a morphological skeleton (see Section 9.3.4), and Fig. 9.14(b) shows the output of function endpoints.

function g = endpoints(f)

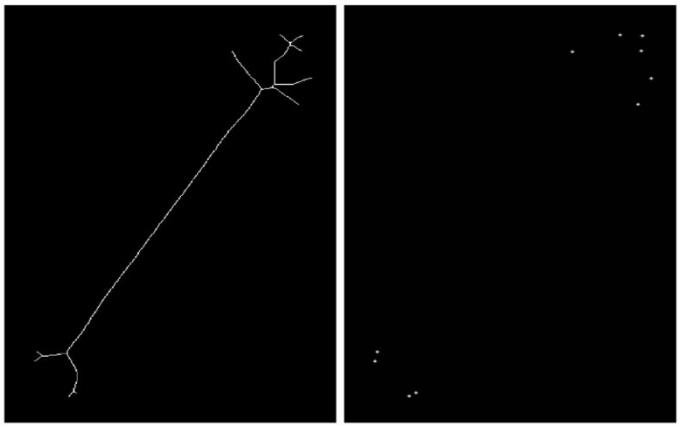
% computes the end points of the binary image f and returns them in the binary image g.

function is_end_point = endpoint_fcn(nhood)

% Determines if a pixel is an end point.

%IS_END_POINT = ENDPOINT_FCN(NHOOD) accepts a 3-by-3 binary neighborhood, NHOOD, and returns a 1 if the center element is an end point; otherwise it returns a 0.

is_end_point = nhood(2, 2) & (sum(nhood(:)) == 2);



a b

FIGURE 9.14
(a) Image of a morphological skeleton.
(b) Output of function `endpoints`. The pixels in (b) were enlarged for clarity.

9.3.4 function bwmorph

IPT function bwmorph implements a variety of useful operations based on combinations of dilations, erosions, and lookup table operations. Calling syntax is

g = bwmorph(*f*, *operation*, *n*)

where *f* is an input binary image, *operation* is a string specifying the desired operation, and *n* is a positive integer specifying the number of times the operation is

to be repeated.

Table 9.3

Operation	Description
`bothat`	"Bottom-hat" operation using a 3 × 3 structuring element; use `imbothat` (see Section 9.6.2) for other structuring elements.
`bridge`	Connect pixels separated by single-pixel gaps.
`clean`	Remove isolated foreground pixels.
`close`	Closing using a 3 × 3 structuring element; use `imclose` for other structuring elements.
`diag`	Fill in around diagonally connected foreground pixels.
`dilate`	Dilation using a 3 × 3 structuring element; use `imdilate` for other structuring elements.
`erode`	Erosion using a 3 × 3 structuring element; use `imerode` for other structuring elements.
`fill`	Fill in single-pixel "holes" (background pixels surrounded by foreground pixels); use `imfill` (see Section 11.1.2) to fill in larger holes.
`hbreak`	Remove H-connected foreground pixels.
`majority`	Make pixel p a foreground pixel if at least five pixels in $N_8(p)$ (see Section 9.4) are foreground pixels; otherwise make p a background pixel.
`open`	Opening using a 3 × 3 structuring element; use function `imopen` for other structuring elements.
`remove`	Remove "interior" pixels (foreground pixels that have no background neighbors).
`shrink`	Shrink objects with no holes to points; shrink objects with holes to rings.
`skel`	Skeletonize an image.
`spur`	Remove spur pixels.
`thicken`	Thicken objects without joining disconnected 1s.
`thin`	Thin objects without holes to minimally connected strokes; thin objects with holes to rings.
`tophat`	"Top-hat" operation using a 3 × 3 structuring element; use `imtophat` (see Section 9.6.2) for other structuring elements.

TABLE 9.3
Operations supported by function `bwmorph`.

Thinning means reducing binary objects or shapes in an image to strokes that are a single pixel wide.

a b c

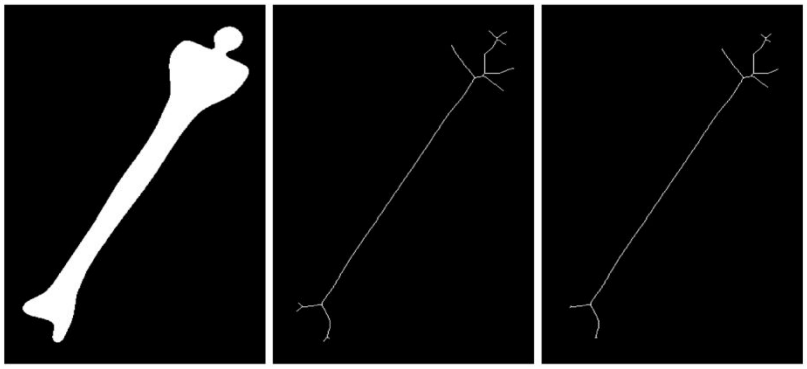
FIGURE 9.15 (a) Fingerprint image from Fig. 9.11(c) thinned once. (b) Image thinned twice. (c) Image thinned until stability.

9-15

```
f = imread('fingerprint_cleaned.tif');
g1 = bwmorph(f, 'thin', 1);
g2 = bwmorph(f, 'thin', 2);    imshow(g1), figure, imshow(g2)
ginf = bwmorph(f,'thin',Inf);
imshow(ginf)
```

Skeletonization is another way to reduce binary image objects to a set of thin strokes that retain important information about the shapes of the original objects. Figure 9.16(b) shows the resulting skeleton, which is a reasonable likeness of the

basic shape of the object.



a b c

FIGURE 9.16 (a) Bone image. (b) Skeleton obtained using function `bwmorph`. (c) Resulting skeleton after pruning with function `endpoints`.

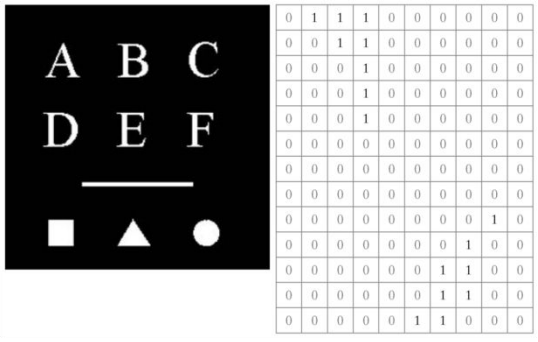
9-16

```
fs = bwmorph(f, 'skel', Inf);
imshow(f), figure, imshow(fs);
for k =1:5
fs = fs & -endpoints(fs);
end
```

9.4 Labeling Connected Components

The concepts discussed thus far are applicable mostly to all foreground (or all background) individual pixels and their immediate neighbors.

From Fig. 9.17, we can see that To develop computer programs that locate and operate on objects, we need a more precise set of definitions for key terms.

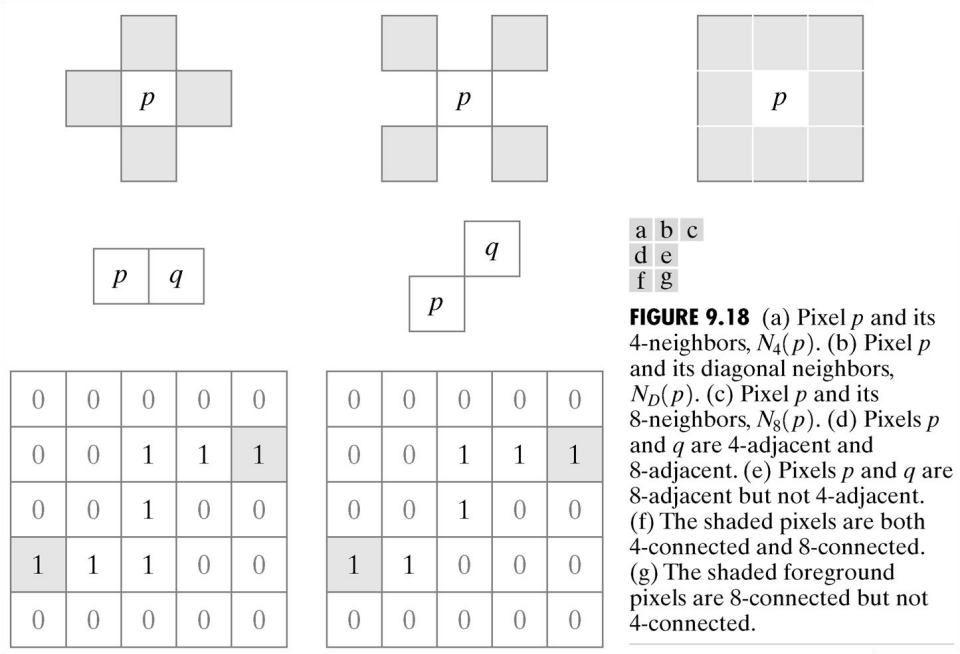


0	1	1	1	0	0	0	0	0	0
0	0	1	1	0	0	0	0	0	0
0	0	0	1	0	0	0	0	0	0
0	0	0	1	0	0	0	0	0	0
0	0	0	1	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	1	0
0	0	0	0	0	0	0	1	0	0
0	0	0	0	0	0	1	1	0	0
0	0	0	0	0	0	1	1	0	0
0	0	0	0	0	1	1	0	0	0

a b

FIGURE 9.17
(a) Image containing ten objects. (b) A subset of pixels from the image.

9-17



a b c
d e
f g

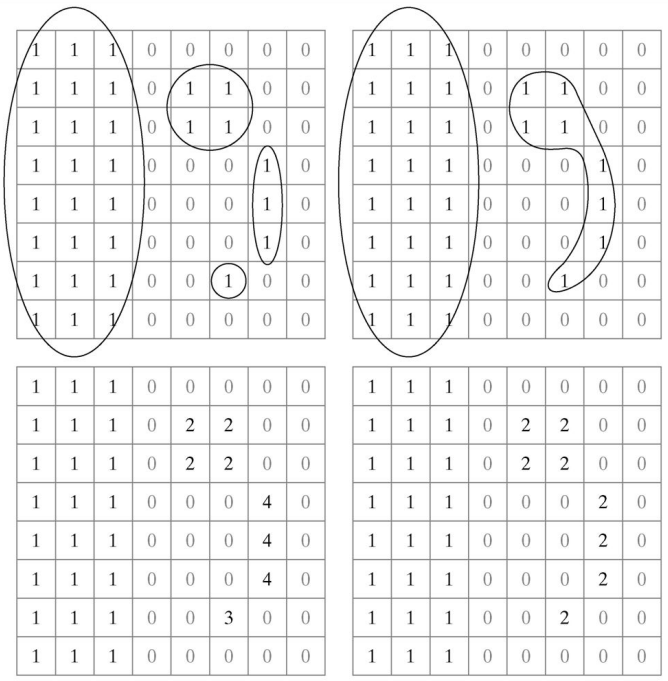
FIGURE 9.18 (a) Pixel p and its 4-neighbors, $N_4(p)$. (b) Pixel p and its diagonal neighbors, $N_D(p)$. (c) Pixel p and its 8-neighbors, $N_8(p)$. (d) Pixels p and q are 4-adjacent and 8-adjacent. (e) Pixels p and q are 8-adjacent but not 4-adjacent. (f) The shaded pixels are both 4-connected and 8-connected. (g) The shaded foreground pixels are 8-connected but not 4-connected.

A pixel p at coordinates (x, y) has two horizontal and two vertical neighbors whose coordinates are (x + 1, y), (x-1, y), (x, y + 1) and (x, y-1). This set of 4-neighbors of p, denoted N4(p), is shaded in Fig. 9.18(a).

The four diagonal neighbors of p have coordinates (x+ 1, y + 1), (x + 1, y - 1), (x- 1, y + 1) and (x -1, y+1). Figure 9.18(b) shows these neighbors, which are denoted ND(p).

The union of N4(p) and ND(p) in Fig. 9.18(c) are the 8-neighbors of p, denoted N8(p). Two pixels p and q are said to be 4-adjacent if $q \in N_4(p)$. Similarly,p and q are said to be 8-adjacent if $q \in N_8(p)$. Figures 9.18(d) and (e) illustrate these concepts. A path can be 4-connected or 8-connected, depending on the definition of adjacency used.

The term connected component was just defined in terms of a path, and the definition of a path in turn depends on adjacency. This implies that the nature of a connected component depends on which form of adjacency we choose, with 4- and 8-adjacency being the most common. Figure 9.19(a) shows a small binary image with four 4-connected components. Figure 9.19(b) shows that choosing 8-adjacency reduces the number of connected components to two.



a b
c d

FIGURE 9.19 Connected components (a) Four 4-connected components. (b) Two 8-connected components. (c) Label matrix obtained using 4-connectivity (d) Label matrix obtained using 8-connectivity.

IPT function bwlabel computes all the connected components in a binary image. The calling syntax is

[L, num] = bwlabel(*f, conn*)

where *f* is an input binary image and conn specifies the desired connectivity (either 4 or 8). Figure 9.19(c) shows the label matrix corresponding to Fig. 9.19(a), computed using bwlabel (f, 4).

Example: shows how to compute and display the center of mass of each connected component in Fig. 9.17(a).

FIGURE 9.20 Centers of mass (white asterisks) shown superimposed on their corresponding connected components.

```
f = imread('objects.tif');
[L, n] = bwlabel(f);
imshow(f)
hold on    % So later plotting commands plot on top of the image.
for k = 1:n
[r, c] = find(L == k);
rbar = mean(r);
cbar = mean(c);
plot(cbar, rbar, 'Marker', 'o',    'MarkerEdgeColor' , 'k',...    'MarkerFaceColor' , 'k',
'Marker-Size1 , 10);
plot(cbar, rbar, 'Marker', '*', 'MarkerEdgeColor', 'w');
end
```

9.5 Morphological Reconstruction

Reconstruction is a morphological transformation involving two images and a structuring element. One image, the marker, is the starting point for the transformation. The other image, the mask, constrains the transformation.

If g is the mask and f is the marker, the reconstruction of g from f, denoted *Rg(f)*, is defined by the following iterative procedure:

1. Initialize *h1* to be the marker image *f*.
2. Create the structuring element: B = ones (3).
3. Repeat:

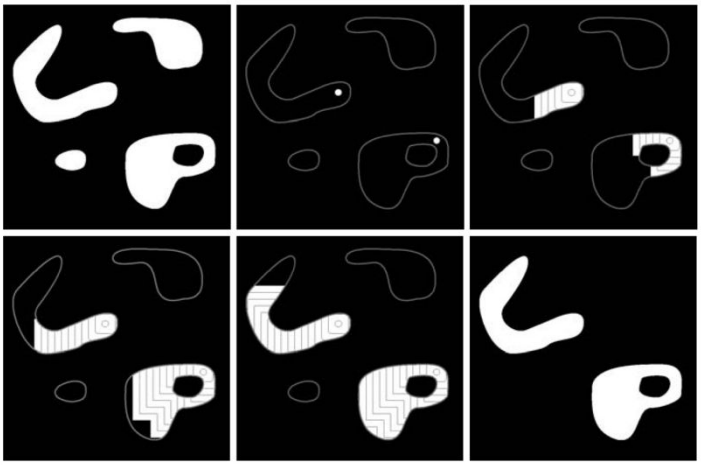
$$h_{k+1} = (h_k \oplus B) \cap g$$

until hk+1 = hk.

Marker *f* must be a subset of *g*; that is, $f \in g$.

Figure 9.21 illustrates the preceding iterative procedure. IPT function imreconstruct uses the "fast hybrid reconstruction" algorithm described in Vincent [1993]. The calling syntax for imreconstruct is

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



a b c
d e f

FIGURE 9.21 Morphological reconstruction. (a) Original image (the mask). (b) Marker image. (c)–(e) Intermediate result after 100, 200, and 300 iterations, respectively. (f) Final result. [The outlines of the objects in the mask image are superimposed on (b)–(e) as visual references.]

9-21

A comparison between opening and opening by reconstruction for an image containing text.

a b
c d
e f
g

FIGURE 9.22 Morphological reconstruction: (a) Original image. (b) Eroded with vertical line. (c) Opened with a vertical line. (d) Opened by reconstruction with a vertical line. (e) Holes filled. (f) Characters touching the border (see right border). (g) Border characters removed.

9-22

9.6 Gray-Scale Morphology

In this section, as in the binary case, we start with dilation and erosion, which for gray-scale images are defined in terms of minima and maxima of pixel neighborhoods. All the binary morphological operations , with the exception of the hit-or-miss transform, have natural extensions to gray-scale images.

9.6.1 Dilation and Erosion

The gray-scale dilation of f by structuring element b, denoted $f \oplus b$, is defined as

$$(f \oplus b)(x,y) = \max\{f(x - x',y - y') + b(x',y') \mid (x',y') \in D_b\}$$

The gray-scale erosion of f by structuring element b, denoted $f - b$, is defined as

$$(f-b)(x,y) = \min\{f(x + x',y + y') - b(x',y') \mid (x',y') \in D_b\}$$

a b
c d

FIGURE 9.23 Dilation and erosion. (a) Original image. (b) Dilated image. (c) Eroded image. (d) Morphological gradient. (Original image courtesy of NASA.)

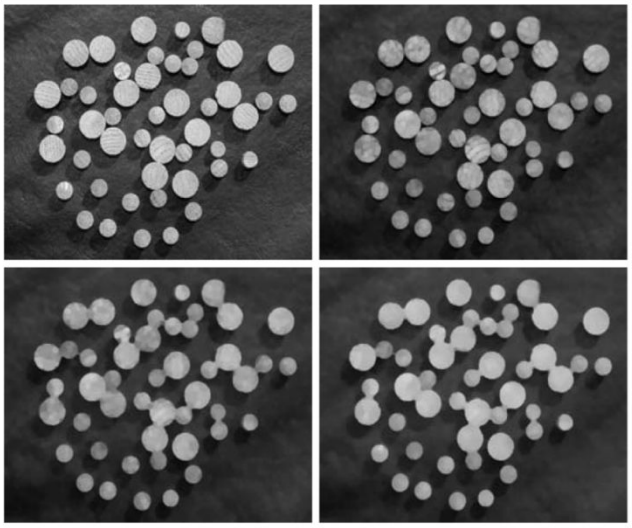
Flat structuring elements for gray-scale images are created using *strel* in the same way as for binary images. For example, the following commands show how to dilate and erosion the image *f* in Fig. 9.23(a) using a flat 3x3 structuring element:

9.6.2 Opening and Closing

The opening of image f by structuring element b, denoted $f \circ b$, is defined as $f \circ b = (f-b) \oplus b$.

Similarly, the closing of *f* by *b*, denoted $f \bullet b$, is dilation followed by erosion: $f \bullet b = (f \oplus b) - b$.

Opening suppresses bright details smaller than the structuring element. Closing suppresses dark details smaller than the structuring element. They are used often in combination for image smoothing and noise removal.



a b
c d

FIGURE 9.25 Smoothing using openings and closings. (a) Original image of wood dowel plugs. (b) Image opened using a disk of radius 5. (c) Closing of the opening. (d) Alternating sequential filter result.

Figure 9.25(a) :

```
f = imread('plugs.jpg');
se = strel('disk',      5);
fo = imopen(f,      se);
foe = imclose(fo,      se);
```

we use imopen and imclose to smooth the image of wood dowel plugs.

Figure 9.25(b,c) :

```
fasf = f;
for k = 2:5
se = strel('disk', k);
fasf = imclose(imopen(fasf, se), se);
end
```

we use openings and closings in combination is in alternating sequential filtering.

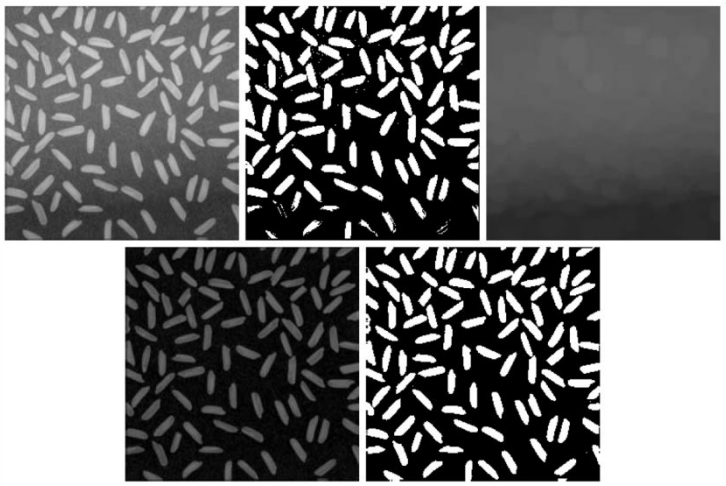
Openings can be used to compensate for nonuniform background illumination. Figure 9.26(a) shows an image, f, of rice grains in which the background is darker towards the bottom than in the upper portion of the image. Figure 9.26(b) , for example, is a thresholded version in which grains at the top of the image are well separated from the background, but grains at the bottom are improperly extracted from the background.

the commands se = strel('disk', 10); fo = imopen(f, se);resulted in the opened image in Fig. 9.26(c).

By subtracting this image from the original image, we can produce an image of the grains with a reasonably even background:

f2 = imsubtract(f, fo); % top-hat transformation

Figure 9.26(d) shows the result, Fig. 9.26(e) shows the new thresholded image. The improvement is apparent.



a b c
d e

FIGURE 9.26 Top-hat transformation. (a) Original image. (b) Thresholded image. (c) Opened image. (d) Top-hat transformation. (e) Thresholded top-hat image. (Original image courtesy of The MathWorks, Inc.)

9.6.3 Reconstruction

h-minima transform

marker image is produced by (mask image-h)

opening-by-reconstruction

f=imread('plugs.jpg'); se=strel('disk',5);

fe=imerode(f, se); fobr=imreconstruct(fe, f)

closing-by-reconstruction

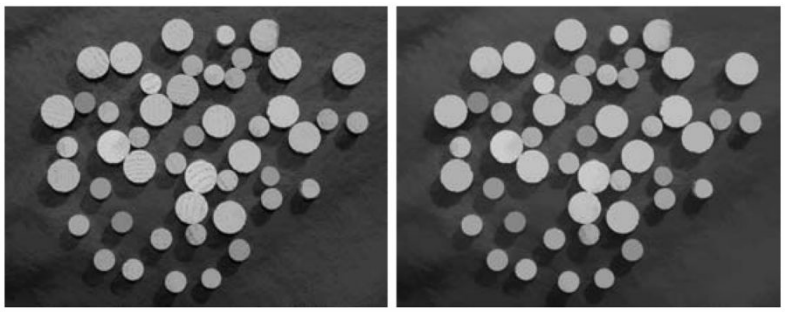
fobrc=imcomplement(fobr);

fobrce=imerode(fobrc, se);

forbcbr=imcomplement(imreconstruct(fobrce, fobrc))

a b

FIGURE 9.29
(a) Opening-by-reconstruction. (b) Opening-by-reconstruction followed by closing-by-reconstruction.



9-29

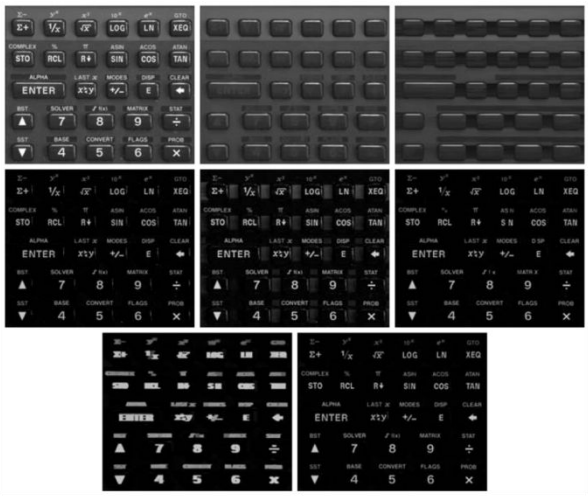


FIGURE 9.30 An application of gray-scale reconstruction. (a) Original image. (b) Opening-by-reconstruction. (c) Opening. (d) Tophat-by-reconstruction. (e) Tophat. (f) Opening-by-reconstruction of (d) using a horizontal line. (g) Dilation of (f) using a horizontal line. (h) Final reconstruction result.

9-30

Summary

The morphological concepts and techniques introduced in this chapter constitute a powerful set of tools for extracting features from an image.

The basic operators of erosion, dilation, and reconstruction defined for both binary and gray-scale image processing can be used in combination to perform a wide variety of tasks.

As shown in the following chapter, morphological techniques can be used for image segmentation. Moreover, they play a major role in algorithms for image description, as discussed in Chapter 11.

Chapter 10 Image Segmentation

Image segmentation subdivides an image into its constituent regions or objects. Segmentation algorithms for gray images are based on two basic properties: discontinuity and similarity. One is the approach that is to partition an image based on abrupt changes in intensity, such as edges in an image. The other is approach that is based on partitioning an image into regions that are similar according to a set of predefined criteria.

This chapter we discuss: point、line 、and edge detection and region-oriented segmentation approaches (Hough transform, thresholding, watershed segmentation).

10.1 Point 、Line 、and Edge Detection

The most common way to look for discontinuities is to run a mask through the image.

The response R of mask at any point in the image is given by:

$$R = w_1 z_1 + w_2 z_2 + \cdots + w_9 z_9 = \sum_{i=1}^{9} w_i z_i$$

Where *Zi* is the intensity of the pixel associated with mask coefficient *Wi*.

10.1.3 point detection

The detection of isolate points embed-ed in constant or nearly constant areas is detected as :

$$|R| \geq T$$

In matlab using function: imfilter.

Example 10.1

```
>>w=[-1,-1,-1;-1,8,-1;-1,-1,-1];
>>g=abs(imfilter(double(f),w));
>>g=g>=T;          %(T given threshold)
>>imshow(T)
```

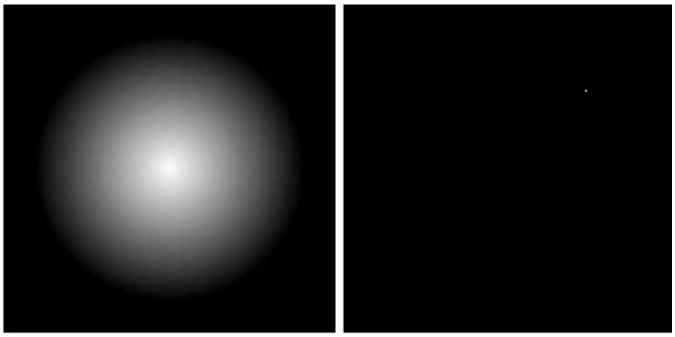
−1	−1	−1
−1	8	−1
−1	−1	−1

FIGURE 10.1 A mask for point detection.

10- 1 A mask for point detection

10- 2

Another approach is to find the points in all neighborhoods of size m*n for which the difference of the maximum and



a b

FIGURE 10.2 (a) Gray-scale image with a nearly invisible isolated black point in the dark gray area of the northeast quadrant. (b) Image showing the detected point. (The point was enlarged to make it easier to see.)

minimum pixels values exceeds a specified value of T.

Function imsubtract

g=imsubtract(ordfilt2(f,m*n,ones(m,n)),…ordfilt2(f,1,ones(m,n)));

g=g>=T;

10.1.2 Line Detection

−1	−1	−1	−1	−1	2	−1	2	−1	2	−1	−1
2	2	2	−1	2	−1	−1	2	−1	−1	2	−1
−1	−1	−1	2	−1	−1	−1	2	−1	−1	−1	2
Horizontal			+45°			Vertical			−45°		

FIGURE 10.3 Line detector masks.

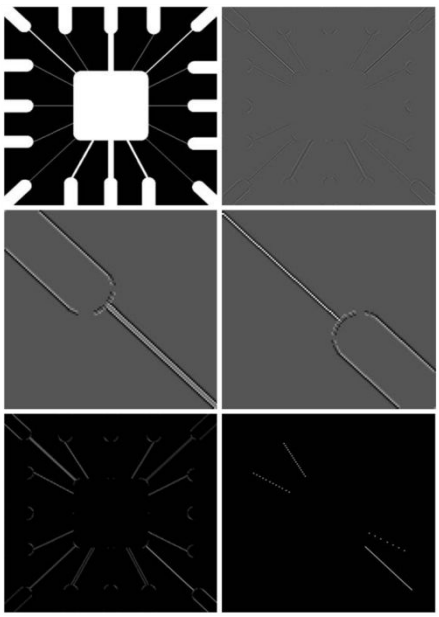
10- 3

If the first mask were moved around an image, it would response to the horizontal lines. The preferred direction of each mask is weighted with a large coefficient than other possible directions. The coefficients of each mask sums to zero.

Let R_1, R_2, R_3, R_4 denotes the responses of the mask i in Figure 10.3.

If at a certain point in a image $|R_i| > |R_j|, j \neq i$ that point is said to be associated with a line in the direction of mask i.

Example 10.2



a b
c d
e f

FIGURE 10.4
(a) Image of a wire-bond mask. (b) Result of processing with the −45° detector in Fig. 10.3. (c) Zoomed view of the top, left region of (b). (d) Zoomed view of the bottom, right section of (b). (e) Absolute value of (b). (f) All points (in white) whose values satisfied the condition g >= T, where g is the image in (e). (The points in (f) were enlarged slightly to make them easier to see.)

10.1.3 Edge Detection using Function Edge

Edge is detected by the first and second-order derivations

The gradient of a 2-D function f(*x,y*) is defined as:

$$\nabla f = \begin{bmatrix} G_x \\ G_y \end{bmatrix} = \begin{bmatrix} \frac{\partial f}{\partial x} \\ \frac{\partial f}{\partial y} \end{bmatrix}$$

The magnitude of this vector is $\nabla f \approx G^2{}_x + G_y{}^2$.

Second-order derivatives are generally computed using the Laplacian, which as follows:

$$\nabla f = \frac{\partial^2 f(x,y)}{\partial x^2} + \frac{\partial^2 f(x,y)}{\partial y^2}$$

Notice:the laplacian is seldom used by itself for edge detection: Sensitive to noise; Produces double edges; Unable to detect edge direction;

However: it can be powerful when used in combination with other edge detection techniques.

Two general criteria for edge detection:

1. Find places where the first derivative of the intensity is greater in magnitude

than a specified threshold.

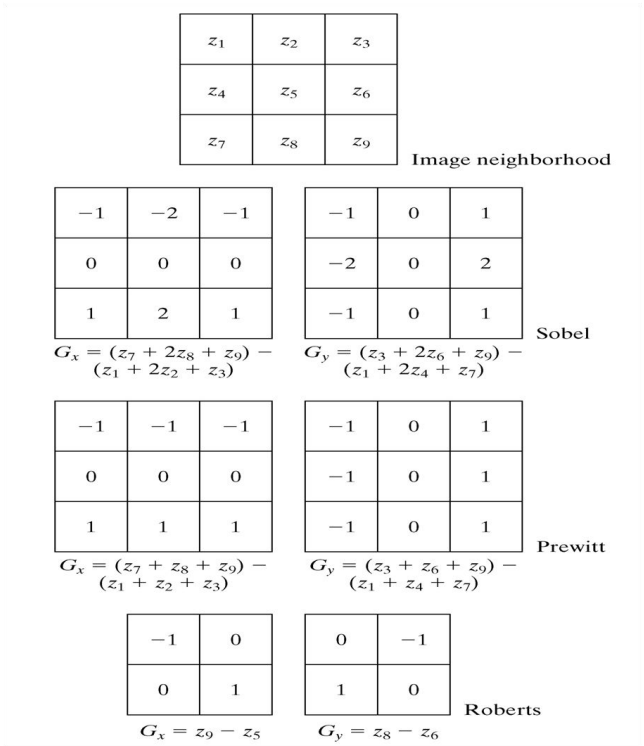
2. Find places where the second derivative of the intensity has a zero crossing.

[g , t]=edge(f,'method',parameters)

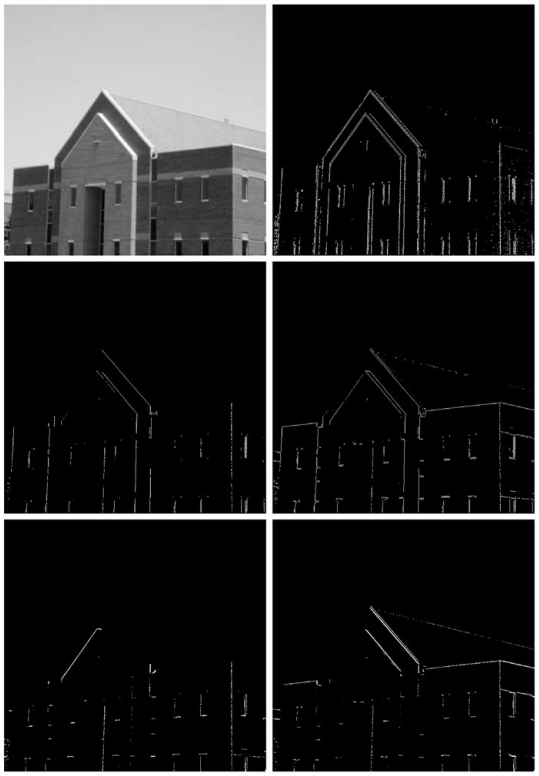
Tab 10.1 Edge detectors available in function edge

Edge Detector	Basic Properties
Sobel	Finds edges using the Sobel approximation to the derivatives shown in Fig. 10.5(b).
Prewitt	Finds edges using the Prewitt approximation to the derivatives shown in Fig. 10.5(c).
Roberts	Finds edges using the Roberts approximation to the derivatives shown in Fig. 10.5(d).
Laplacian of a Gaussian (LoG)	Finds edges by looking for zero crossings after filtering $f(x, y)$ with a Gaussian filter.
Zero crossings	Finds edges by looking for zero crossings after filtering $f(x, y)$ with a user-specified filter.
Canny	Finds edges by looking for local maxima of the gradient of $f(x, y)$. The gradient is calculated using the derivative of a Gaussian filter. The method uses two thresholds to detect strong and weak edges, and includes the weak edges in the output only if they are connected to strong edges. Therefore, this method is more likely to detect true weak edges.

Some edge detector masks and the first-order derivatives they implement.



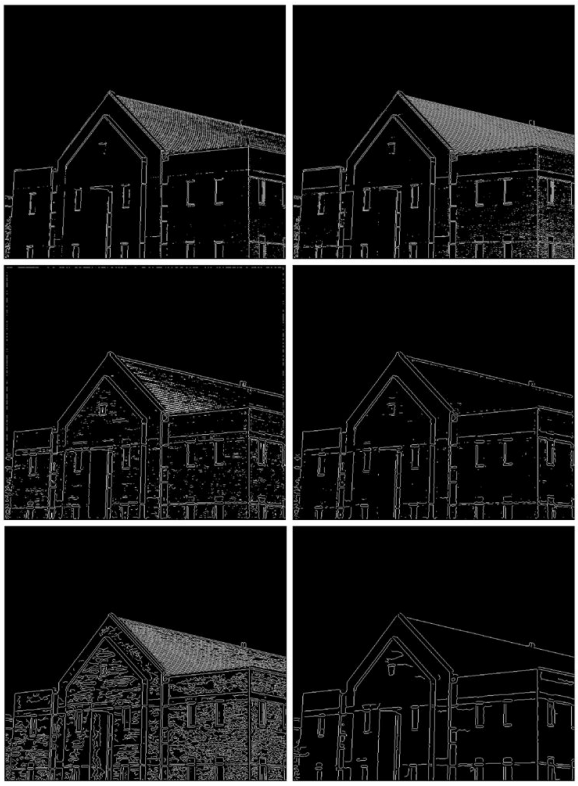
Example 10.3



a b
c d
e f

FIGURE 10.6
(a) Original image. (b) Result of function edge using a vertical Sobel mask with the threshold determined automatically. (c) Result using a specified threshold. (d) Result of determining both vertical and horizontal edges with a specified threshold. (e) Result of computing edges at 45° with `imfilter` using a specified mask and a specified threshold. (f) Result of computing edges at −45° with `imfilter` using a specified mask and a specified threshold.

Example 10.4



a b
c d
e f

FIGURE 10.7 Left column: Default results for the Sobel, LoG, and Canny edge detectors. Right column: Results obtained interactively to bring out the principal features in the original image of Fig. 10.6(a) while reducing irrelevant, fine detail. The Canny edge detector produced the best results by far.

10.2 Line Detection Using the Hough Transform

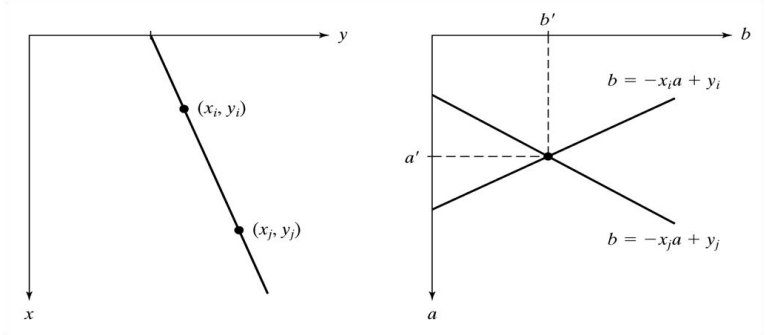
The previous edge detection methods yield pixels that seldom characterize an edge completely because of noise and other effects. Hough transform can be used to find and link line segments.

Hough transform:

Suppose that many lines pass through a point (x_i, y_i), all of which satisfy the slope-intercept equation $y_i = ax_i + b$, for some values of a and b. The equation can be write as $b = -x_i a + y_i$.

A second point (x_j, y_j) also has a line in parameters space associated with it, and this line intersects the line associated with (x_j, y_j) at (a', b'). In fact, all points contained on this line have lines in parameter space that intersect at (a', b').

Figure 10-8 illustrates these concepts.

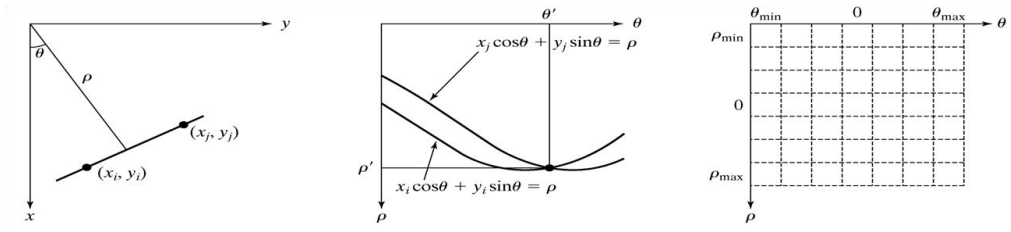


a b

FIGURE 10.8
(a) *xy*-plane.
(b) Parameter space.

A practical difficulty is that a approaches infinity as the line approaches the vertical direction.One way around this difficulty is to use the normal representation of a line:

$$x\cos\theta + y\sin\theta = \rho$$



a b c

FIGURE 10.9 (a) (ρ, θ) parameterization of lines in the *xy*-plane. (b) Sinusoidal curves in the $\rho\theta$-plane; the point of intersection, (ρ', θ'), corresponds to the parameters of the line joining (x_i, y_i) and (x_j, y_j). (c) Division of the $\rho\theta$-plane into accumulator cells.

Figure 10.9(a) illustrates the geometric interpretation of the parameters.

Figure 10.9(b) represents the family of lines that pass through a particular point.

Figure 10.9(c) illustrates the accumulator cells.

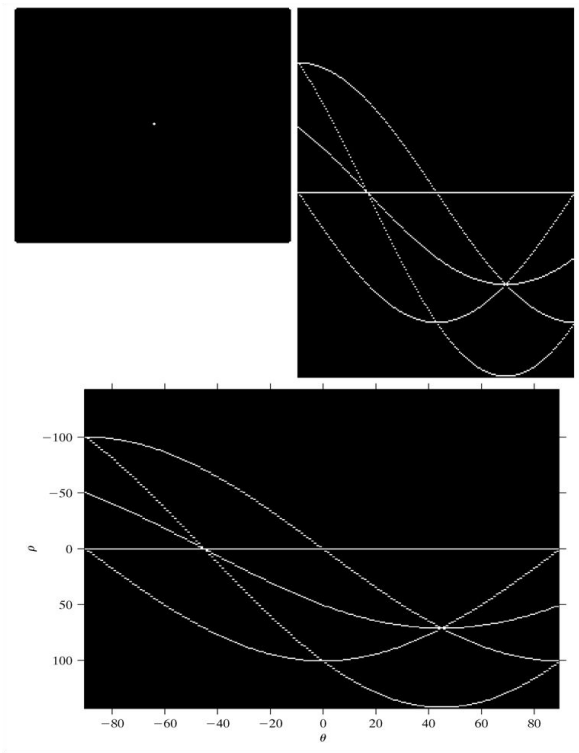
Initially, the cell at coordinates (*i,j*) are set to zeros. Then, for every nonbackground point (x_k, y_k), we let θ equal each of the allowed subdivision values on the θ axis and solve for the corresponding ρ using the equation: $\rho = x_k \cos\theta + y_k \sin\theta$. The resulting ρ value are then rounded off to the nearest allowed cell value along the ρ-axis, the corresponding cell is then incremented. A value of Q in A(*i,j*) means that Q points in the *xy*-plane lie on the line $x\cos\theta + y\sin\theta = \rho$. The number of subdivisions in the $\rho\theta$-plane determines the accuracy of the co-linearity of these points.

A function for computing the Hough transfor-mation is given next.This function makes use of sparse matrices, which provides advantages in both matrix storage space and computation time.

Function hough.m at P396

The test:

Figure 10.10(a) shows the test image. Figure 10.10(b) shows the test results. Figure 10.10(c) shows the labeled results.



a b
c

FIGURE 10.10
(a) Binary image with five dots (four of the dots are in the corners).
(b) Hough transform displayed using `imshow`.
(c) Alternative Hough transform display with axis labeling. (The dots in (a) were enlarged to make them easier to see.)

10.2.1 Hough Transform Peak Detection

Peak Detection is the first step in using Hough Transform.

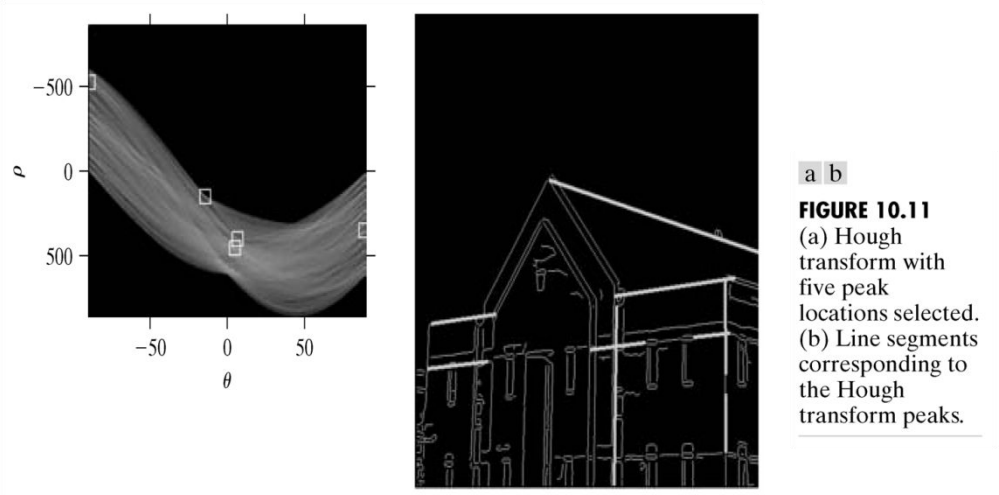
To find a meaningful set of distinct peaks in a Hough transform:(1)Find the Hough transform cell containing the highest value and record its locations. (2)Suppress Hough transform cells in the immediate neighborhood of the maximum found in step 1. (3)Repeat until the desired number of peaks has been found, or until a specified threshold has been reached.

Function ***houghpeaks***

After peak detection, it remains to determine if there are line segments associated with those peaks, as well as where they start and end. The first step is to find the location of all nonzero pixels in the image that contributed to that peak.(function houghpixels.m). The pixels associated with the locations found using houghpixels must be grouped into line segments (function houghlines.m).

Example 10.6 use function hough, houghpeaks, and houghlines to find a set of line segments in the binary image f.

The result.



a b

FIGURE 10.11
(a) Hough transform with five peak locations selected. (b) Line segments corresponding to the Hough transform peaks.

10.3 Thresholding

Thresholding is intuitive and simplicity to implementation. Suppose that the intensity histogram shown in Fig.10.12 corresponds to an image f(*x,y*). The object and background pixels have intensity levels grouped into two dominant modes.

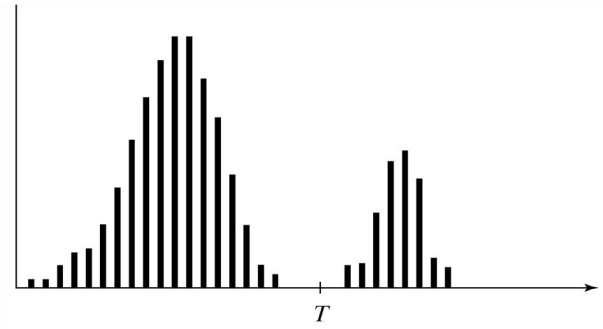


FIGURE 10.12 Selecting a threshold by visually analyzing a bimodal histogram.

One obvious way to extract the objects from the backgrounds is to select a threshold T that separates these modes.

The thresholded image g(*x*,*y*) is defined as:

$$g(x,y)=\begin{cases}1 & if\ f(x,y)\geq T\\ 0 & if\ f(x,y)<T\end{cases}$$

Pixels labeled 1 corresponds to objects. Pixels labeled 0 corresponds to backgrounds. T is a constant, this approach is called global thresholding.

10.3.1 Global Thresholding

One way to choose a threshold is by visual inspection of the image histogram. Another way of choosing T is by trial and error, picking different thresholds until one is found that produces a good result as judged by the observer, which is particularly effective in an interactive environment.

A method for choosing a threshold automatically:

1. Select an initial estimate for T;

2. Segment the image using T, G1(intensities>=T),G2(intensities<T);

3. Compute the average intensity values μ_1 and μ_2 for the pixels in regions G_1 and G_2;

4. Compute a new threshold value: $T=\frac{1}{2}(\mu_1+\mu_2)$;

5. Repeat steps 2 through 4 until the difference in T in successive iterations is smaller than a predefined parameter T_0.

Example 10.7 show how to implement this procedure in Matlab. The toolbox provides a function called graythresh that computes a threshold using Ostu's method.

Treating the normalized histogram as a discrete probability density function as

in :

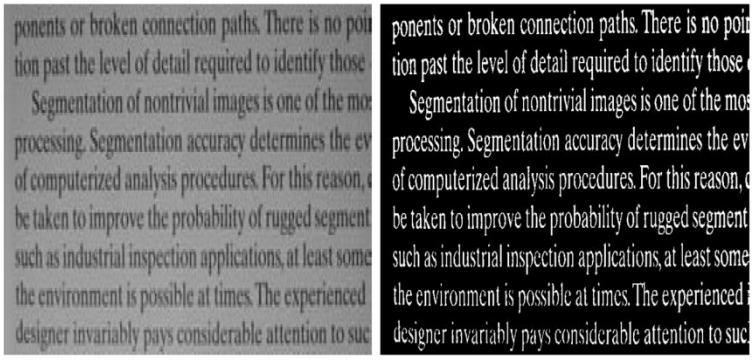
$$p_r(r_q) = \frac{n_q}{n}, q=0,1,2\ldots,L-1$$

ostu's method is to choose the threshold value k that maximum the between-class variance: $\sigma^2{}_B = \omega_0(\mu_0 - \mu_T)^2 + \omega_1(\mu_1 - \mu_T)^2$

The results of Example 10.7.

a b

FIGURE 10.13
(a) Scanned text. (b) Thresholded text obtained using function `graythresh`.



10.3.2 Local Thresholding

- When the background illumination is uneven, global thresholding methods can fail. A common practice in such situations is to preprocess the image to compensate for the illumination problems and then apply a global threshold to the preprocessed image. This process is equivalent to : (fo is the opening operation to f)

$$g(x,y) = \begin{cases} 1 & if\ f(x,y) \geq T(x,y) \\ 0 & if\ f(x,y) < T(x,y) \end{cases}$$

where $T(x,y) = f_0(x,y) + T_0$

10.4 Region-based Segmentation

The objection of segmentation is to partition an image into regions: 10.1 and 10.2 approached this problem by finding boundaries between regions based on discontinuities in intensity levels; 10.3 segmentation was accomplished via thresholds

based on the distribution of pixel properties;10.4 discuss segmentation techniques that based on finding the regions directly.

10.4.1 Basic Formulation

Let R represent the entire image region. Segmen-tation is a process that partitions R into n sub-regions, $R_1, R_2 \cdots R_n$, such that:

$$\sum_{i=1}^{n} R_i = R$$

R_i is a connected region , i=1,2,…,n;

$R_i \bigcap R_j = \varphi$ for all i and j, $i \neq j$;

$P(R_i) = TRUE$ for i=1,2,…,n;

$P(R_i \bigcup R_j) = FALSE$ for any adjacent regions R_i and R_j;

Here, $P(R_i)$ is a logical predicate defined over the points in set R_i and φ is the null set.

10.4.2 Region Growing

Region growing is a procedure that groups pixels or sub-regions into larger regions based on prede-fined criteria for growth. The basic approach is to start with a "seed" points and from these grow regions by appending to each seed those neighboring pixels that have predefined properties similar to the seed.

Selecting a set of one or more starting points often can be based on the nature of the problem. The selection of similarity criteria depends not only on the problem under consideration, but also on the type of image data available.

Another problem in region growing is the formulation of a stopping rule. Basically, growing a region should stop when no more pixels satisfy the criteria for inclusion in that region. To illustrate the principles of how region segmen-tation can be handled in matlab, we develop next an m-function, *regiongrow.m*.

Example 10.8

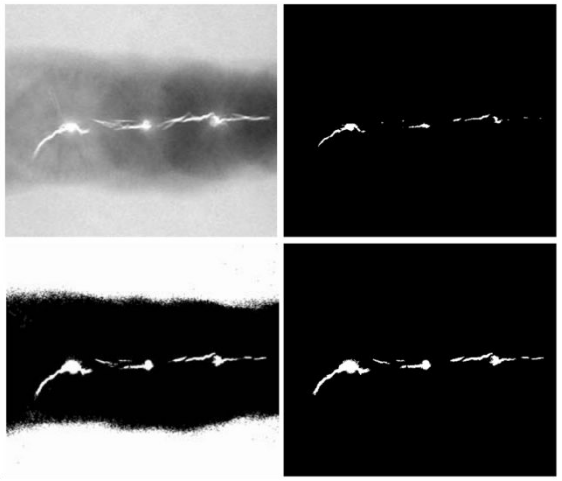
1. Determine the initial seed points.

2. To choose a threshold or threshold array

3. A pixel has to be 8-connected to at least one pixel in a region to be included in

that region. If a pixel is found to be connected to more than one region, the regions are automatically merged by *regiongrow* .

The results:



a b
c d

FIGURE 10.14
(a) Image showing defective welds. (b) Seed points. (c) Binary image showing all the pixels (in white) that passed the threshold test. (d) Result after all the pixels in (c) were analyzed for 8-connectivity to the seed points. (Original image courtesy of X-TEK Systems, Ltd.)

10- 14

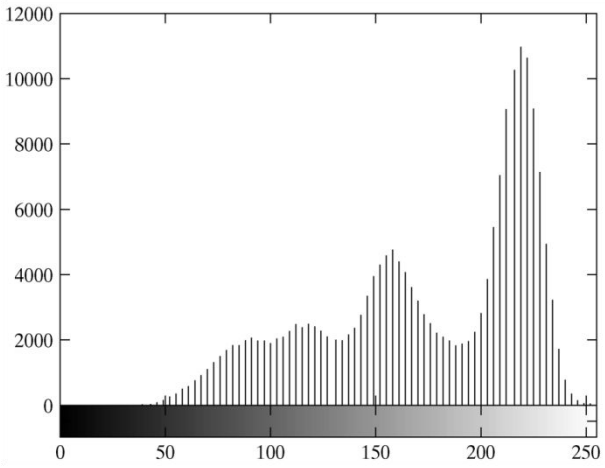


FIGURE 10.15
Histogram of Fig. 10.14(a).

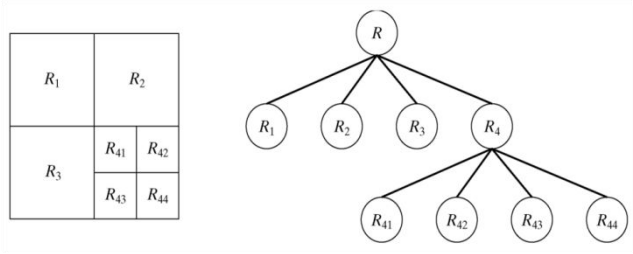
10- 15

10.4.3 Region Splitting and Merging

Region growing's alternative is to subdivide an image initially into a set of arbitrary, disjointed regions and then merge and/or split the regions in an attempt to satisfy the conditions stated in Section 10.4.1.

Let R represent the entire image region and select a predicate P. One approach for segmenting R is to subdivide it successively into smaller and smaller quadrant regions so that, for any region $P(R_i) = TRUE$ .This particular splitting technique has a

convenient representation in the form of a so-called *quadtree*, which illustrated in Fig.10-6.

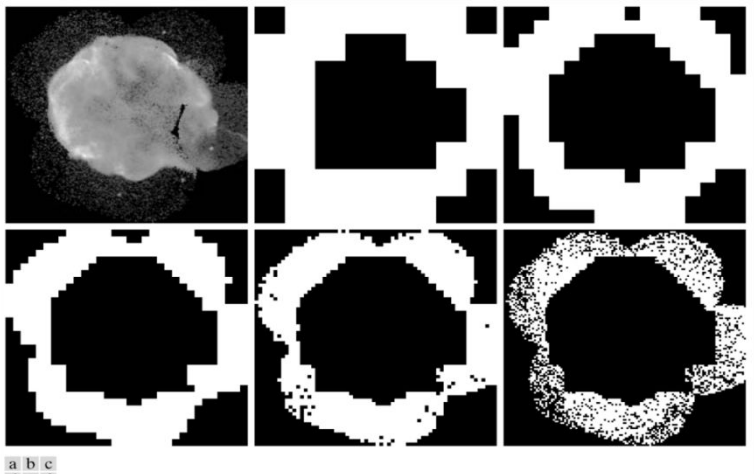


a b

FIGURE 10.16
(a) Partitioned image.
(b) Corresponding quadtree.

The problem can be summarized by the following procedure in which, at any step. 1.Split into four disjoint quadrants any region for R_i which $P(R_i) = TRUE$. 2. When no further splitting is possible, merges any adjacent regions R_i and R_j for $P(R_i \bigcup R_j) = TRUE$ .3.Stop when no further merging is possible.

Example 10.9 Image segmentation using region splitting and merging.



a b c
d e f

FIGURE 10.17 Image segmentation by a split-and-merge procedure. (a) Original image. (b) through (f) results of segmentation using function `splitmerge` with values of `mindim` equal to 32, 16, 8, 4, and 2, respectively. (Original image courtesy of NASA.)

10.5 Segmentation Using the Watershed Transform

In geography, a *watershed* is the ridge that divides areas drained by different river systems. A *catchment* basin is the geographical area draining into a river or reservoir. The watershed transform applies these ideas to gray-scale image processing

in a way that can be used to solve a variety of image segmentation problems.

Understanding the watershed transform requires that we think of a gray-scale image as a topological surface, where the values of f(*x,y*) as heights.Fig.10.18(b)shows 10.18(a)'s three-dimensional surface.

The watershed transform finds the catchment basins and ridge lines in a gray-scale image. The key concepts are to change the starting image into another image whose catchment basins are the objects or regions we want to identity.

10.5.1 Watershed Segmentation Using the Distance Transform

The distance transform is a commonly used tool. The distance transform of a binary image is a relatively simple concept: It is the distance from every pixel to the nearest nonzero-valued pixel.

Figure 10.19 The distance transform. The distance transform can be computed using IPT function *bwdist*. Whose calling syntax is :D=bwdist(f).

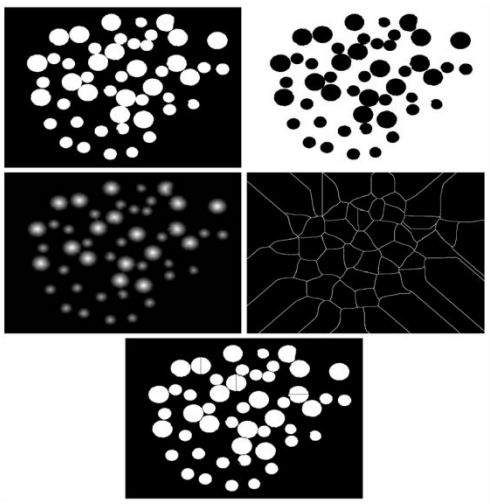
a b

FIGURE 10.19
(a) Small binary image.
(b) Distance transform.

1	1	0	0	0	0.00	0.00	1.00	2.00	3.00
1	1	0	0	0	0.00	0.00	1.00	2.00	3.00
0	0	0	0	0	1.00	1.00	1.41	2.00	2.24
0	0	0	0	0	1.41	1.00	1.00	1.00	1.41
0	1	1	1	0	1.00	0.00	0.00	0.00	1.00

Example 10.10 Segmenting a binary image using the distance and watershed transform.

In this example, some object in Fig.10.20(e) were split improperly. This is called oversegmentation and is a common problem with watershed-based segmentation methods. The next two sections discuss different techniques for overcoming this difficulty.



a b
c d
e

FIGURE 10.20
(a) Binary image. (b) Complement of image in (a). (c) Distance transform. (d) Watershed ridge lines of the negative of the distance transform. (e) Watershed ridge lines superimposed in black over original binary image. Some oversegmentation is evident.

10.5.2 Watershed Segmentation Using Gradient

The gradient magnitude is used often to preprocess a gray-scale image prior to using the watershed transform for segmentation. The gradient magnitude image has high pixel values along object edges, and low pixel values everywhere else. Ideally, then, the watershed transform would result in watershed ridge lines along object edges.

Example 10.11 illustrates this concept.

Fig.10.21(a) shows an image ; Fig.10.21(b) shows the gradient magnitude; Fig.10.21(c) shows the oversegmented image; Fig.10.21(d) the superimposed result, using the approach of smoothing the gradient image before computing watershed transform. The (d) result still have some extraneous ridge lines, and it can be difficult to determine which catchment basins are actually associated with the objects of interest.

10.5.3 Marker-Controlled Watershed Segmentation

Direct application of the watershed transform to a gradient image usually leads to oversegmentation due to noise and other local irregularities of the gradient .The resulting problems can be serious enough to render the result useless.

A practical solution is to limit the number of allowable regions by incorporating a preprocessing stage designed to bring additional knowledge into the segmentation

procedure.

A marker is a connected component belonging to an image. Internal markers are inside each of the objects of interest; External markers are contained within the background. These markers are then used to modify the gradient image using a procedure described in Example 10.12.

Methods used for computing markers: Linear filtering; Nonlinear filtering; Morphological processing.

Example 10.12 results.

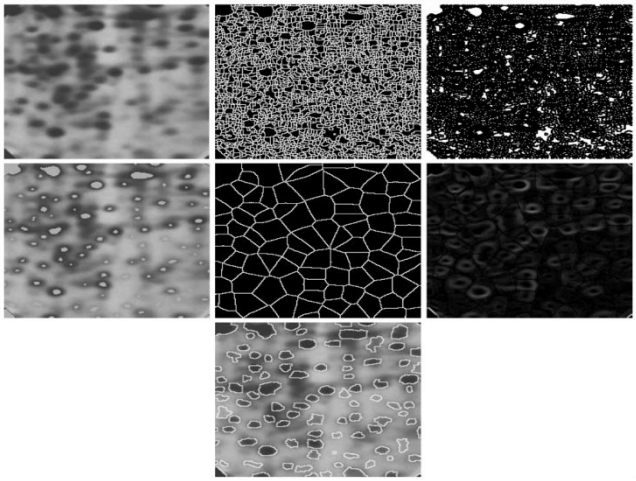


FIGURE 10.22 (a) Gel image. (b) Oversegmentation resulting from applying the watershed transform to the gradient magnitude image. (c) Regional minima of gradient magnitude. (d) Internal markers. (e) External markers. (f) Modified gradient magnitude. (g) Segmentation result. (Original image courtesy of Dr. S. Beucher, CMM/Ecole des Mines de Paris.)

Marker selection can range from the simple procedures just described to considerably more complex methods involving size, shape, location, relative distances, texture content, and so on. The point is that using markers brings a priori knowledge to bear on the segmentation problem.

Summary

Image segmentation is an essential preliminary step in most automatic pictorial pattern recognition and scene analysis problems. As indicated by the range of examples presented in this chapter, the choice of one segmentation technique over another is dictated mostly by the particular characteristics of the problem being considered.

The methods discussed in this chapter, although far from exhaustive, are representative of techniques used commonly in practice.

Chapter 11 Representation and Description

Preview

Representing a region involves two basic choices: (1) external characteristics (selected when interest is on shape characteristics); (2)internal characteristics (selected when interest is on regional properties). In either case, the features selected as descriptors should be as insensitive as possible to variations in region size, translation, and rotation.

11.1 Background

A region is a connected component, and the boundary (border or contour) of a region is the set of pixels in the region that have one or more neighbors that are not in the region. We allow pixels to have gray-scale or multispectral values.

Boundary is a connected set of points. If the points form a clockwise or counterclockwise sequence we say to be ordered.

11.1.1 Cell Arrays and Structures

Cell Arrays

Cell Arrays provide a way to combine a mixed set of objects under one variable name.

A single variable C

C={f, b, char_array}

Here f , an uint8 image of size 512*512; b, a sequence of 2-D coordinates in the form of rows of a 188*2 array; char_array, a cell array containing two character name.

Structures

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

function s=image_stats(f)

s.dim=size(f);

s.AI=mean(f);

s.AIrows=mean(f);

s.AIcols=mean(f);

s is a structure. AI (a scalar), dim(a 1*2 vector), AIrows(an M*1 vector), Aicols(a 1*N).

Notes the use of a dot to separate the structure from its various fields. The field names arbitrary, but they must begin with a nonnumeric character.

11.1.2 Some Additional MATLAB and IPT Function Used in This Chapter

fB、fI represent binary and intensity images. gB: binary output images .

gB=imfill(fB, locations, conn)

gB=imfill(fB, conn, 'holes')

G=imfill(fI, conn, 'holes')

[r,c]=find(g==2)

z=sortrows(S)

[z, m, n]=unique(s, 'rows')

Z=circshift(s, [ud lr])

Z=circshift(s, ud)

Suppose that we want to find the boundary of the object with the longest boundary in image f:

```
>>B=boundaries(f);
>>d=cellfun('length', B);
>>[max_d, k]=max(d);
>>v=B{k(1)};
```

b8=boundary2eight(b)

b4=boundary2four(b)

G=boundary2im(b, M, N, x0, y0)

b=cat(1,B{:})

[s, su]=bsubsamp(b,gridsep)

Z=connectpoly(s(:,1), s(:, 2))

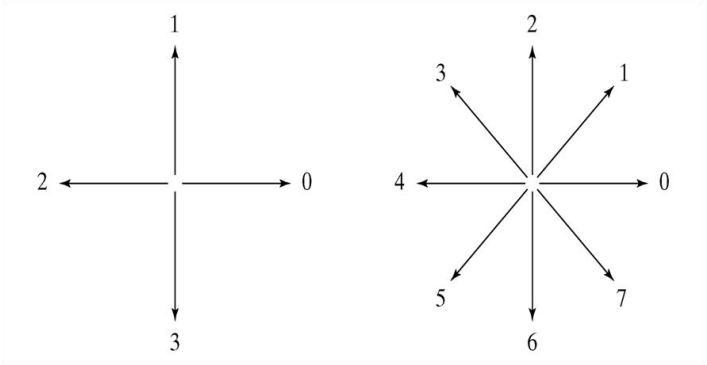
[x, y]=intline(x1, x2, y1, y2)

11.2 Representation

Standard practice is to use schemes that compact the data into representations that are considerably more useful in the computation of descriptors.

11.2.1 Chain Codes

Chain codes are used to represent a boundary by a connected sequence of straight-line segments of specified length and direction. The direction of each segment is coded by using a numbering scheme such as the ones shown in Figs. 11.1(a) and (b).



a b

FIGURE 11.1
(a) Direction numbers for (a) a 4-directional chain code, and (b) an 8-directional chain code.

The chain code of a boundary depends on the starting point.

Function fchcode

c=fchcode(b, conn, dir)

Computes the Freeman chain code of an np*2 set of ordered boundary points stored in array b.

The output c is a structure with the following fields.

c.fcc=Freeman chain code (1*np)； c.diff=First difference of code c.fcc (1*np)

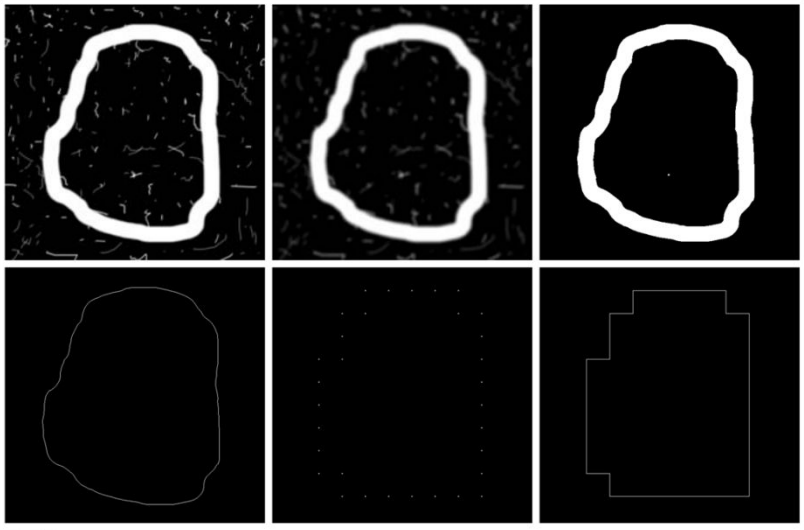
c.mm=Integer of minimum magnitude (1*np)；

c.diffmm=First difference of code c.mm(1*np)；

c.x0y0=Coordinates where the code starts (1*2).

Example 11.3 Freeman chain code and some of its variations

```
>>h=fspecial('average', 9);
>>g=imfilter(f, h, 'replicate');
>>g=im2bw(g, 0.5);
>>B=boundaries(g);
>>d=cellfun('length', B);
>>[max_d, k]=max(d);
>>b=B{1};
>>[M N]=size(g);
>>g=bound2im(b, M, N, min(b(:, 1)), min(b(:, 2)));
>>[s,su]=bsubsamp(b,50);
>>g2=bound2im(s, M, N, min(s(:, 1)), min(s(:, 2)));
>>cn=connectpoly(s(:, 1), s(:, 2));
>>g2=bound2im(cn, M ,N min(cn(:, 1)), min(cn(:, 2)));
>>c=fchcode(su);
```



a b c
d e f

FIGURE 11.2 (a) Noisy image. (b) Image smoothed with a 9 × 9 averaging mask. (c) Thresholded image. (d) Boundary of binary image. (e) Subsampled boundary. (f) Connected points from (e).

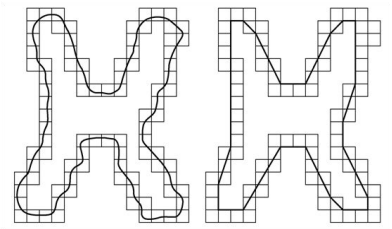
11.2.2 Polygonal Approximations Using Minimum-Perimeter Polygons

The goal of a polygonal approximation is to use the fewest possible to

capture the “essence” of the boundary shape.

A particularly attractive approach to polygonal approximation is to find the minimum-perimeter polygon (MPP) of a region or boundary.

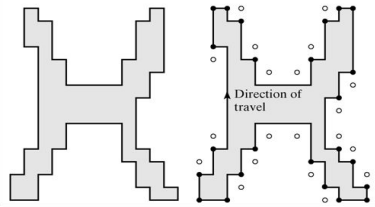
Foundation



a b

FIGURE 11.3 (a) Object boundary enclosed by cells. (b) Minimum-perimeter polygon.

11- 3



a b

FIGURE 11.4 (a) Region enclosed by the inner wall of the cellular complex in Fig. 11.3(a). (b) Convex (•) and concave (◦) corner markers for the boundary of the region in (a). Note that concave markers are placed diagonally opposite their corresponding corners.

11- 4

An Algorithm for Finding MPPs:(1)obtain the cellular complex; (2)obtain the region internal to the cellular complex; (3) use function boundaries to obtain the boundary of the region in step 2 as a 4-connected, clockwise sequence of coordinates; (4)obtain the Freeman chain code of this 4-connected sequence using function fchcode. (5)obtain the convex (black dots) and concave (white dots) vertices from the chain code; (6) From an initial ploygon using the black dotd as vertices, and delete from furture analysis any white dots that are outside this polygon. (7) From a polygon with the remaining black and white dots as vertices; (8) Delete all black dots that are concave vertices; (9) Repeat steps 7 and 8 until all changes cease, at which time all vertices with angles of 180 degrees are deleted. The remaining dots are the vertices of the MPP.

Some of the M-Function Used in Implementing the MPP Algorithm.

Function *qtdecomp*

Q=qtdecomp(B, threshold, [mindim maxdim])

Function *qtgetblk*

[vals, r, c]=qtgetblk(B, Q, mindim)

Example 11.4 Obtaining the cellular wall of the boundary of a region

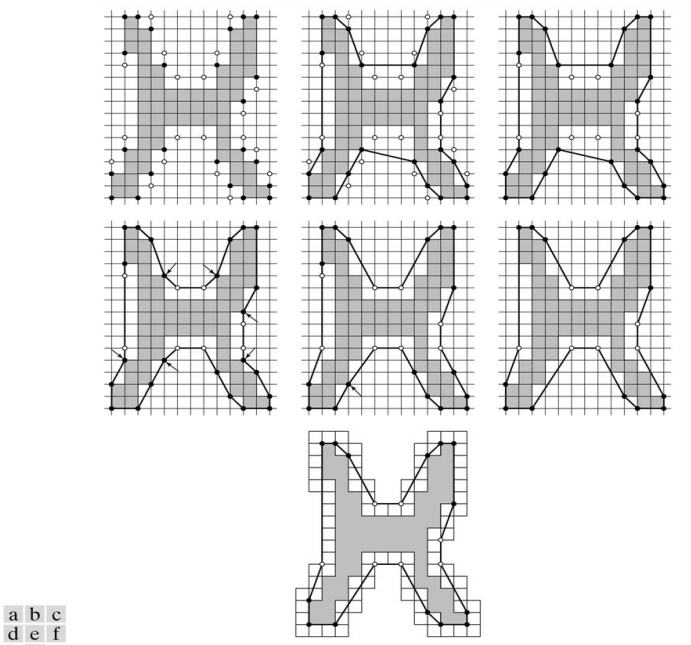
>>B=bwperim(B, 8);

>>Q=qtdecomp(B, 0, 2);

>>R=imfill(BF, ‘holes’) & ~BF;

>>b=boundaries(b, 4, ‘cw’);

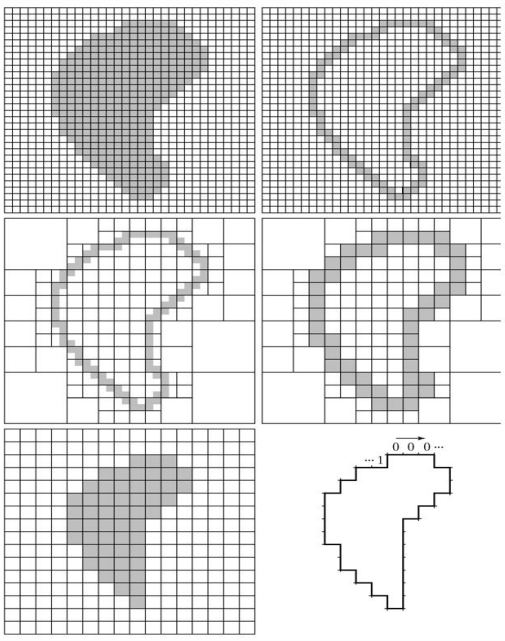
>>b=b{1};



a b c
d e f
g

FIGURE 11.5 (a) Convex (black) and concave (white) vertices of the boundary in Fig. 11.4(a). (b) Initial polygon joining all convex vertices. (c) Result after deleting concave vertices outside of the polygon. (d) Result of incorporating the remaining concave vertices into the polygon (the arrows indicate black vertices that have become concave and will be deleted). (e) Result of deleting concave black vertices (the arrow indicates a black vertex that now has become concave). (f) Final result showing the MPP. (g) MPP with boundary cells superimposed.

11- 5’



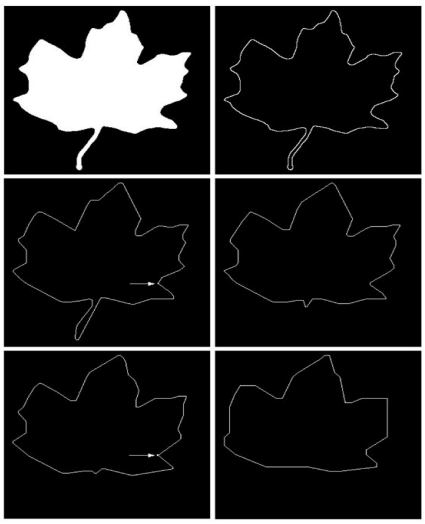
a b
c d
e f

FIGURE 11.6
(a) Original image, where the small squares denote individual pixels. (b) 4-connected boundary. (c) Quadtree decomposition using blocks of minimum size 2 pixels on the side. (d) Result of filling with 1s all blocks of size 2×2 that contained at least one element valued 1. This is the cellular complex. (e) Inner region of (d). (f) 4-connected boundary points obtained using function `boundaries`. The chain code was obtained using function `fchcode`.

11- 6

Example 11.5 Using Function minperpoly.

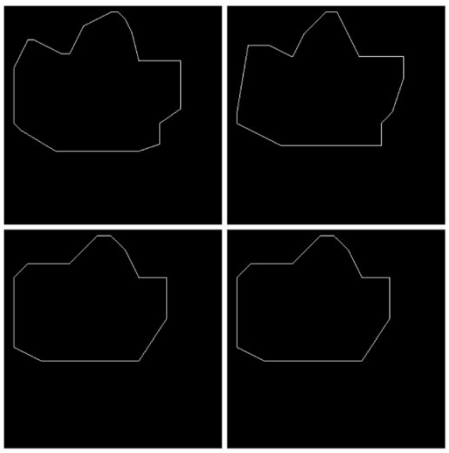
```
>>b=boundaries(B, 4, 'cw');
>>b=b{1};
>>[M, N]=size(B);
>>xmin=min(b(:, 1));
>>ymin=min(b(:, 2));
>>bim=bound2im(b, M, N, xmin, ymin);
>>imshow(bim)
>>[x, y]=minperpoly(B, 2);
>>b2=connectpoly(x, y);
>>B2=bound2im(b2, M, N, xmin, ymin);
>>imshow(B2)
```



a b
c d
e f

FIGURE 11.7
(a) Original image.
(b) 4-connected boundary.
(c) MPP obtained using square bounding cells of size 2. (d) through (f) MPPs obtained using square cells of sizes 3, 4, and 8, respectively.

11- 7



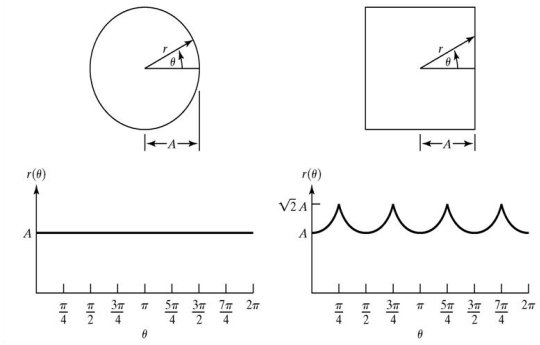
a b
c d

FIGURE 11.8
MPPs obtained with even larger bounding square cells of sizes (a) 10, (b) 16, (c) 20, and (d) 32.

11- 8

11.2.3 Signatures

One of the simplest ways that generate a signature is to plot the distance from an interior point (e.g., the centroid) to the boundary as a function of angle, as illustrated in Fig.11.9.



a b
c d

FIGURE 11.9
(a) and (b) Circular and square objects. (c) and (d) Corresponding distance versus angle signatures.

11- 9

Function *signature* finds the signature of a given boundary.

[st, angle, x0, y0]=signature(b, x0, y0)

Function *cart2pol* to convert Cartesian to polar coordinates

[THETA, RHO]=cart2pol(X, Y)

Function *pol2cart* is used for converting back to Cartesian coordinates

[X, Y]=plo2cart(THETA, RHO)

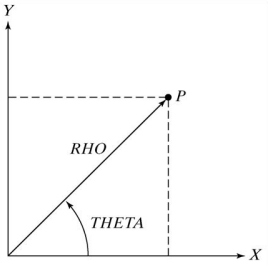
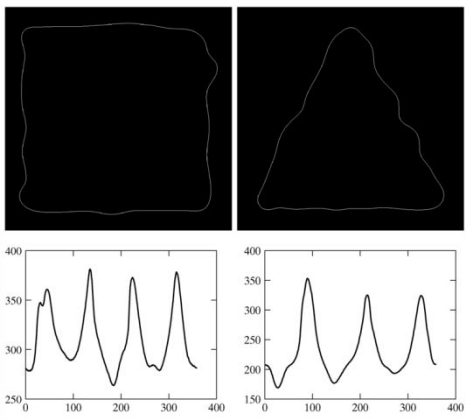


FIGURE 11.10
Axis convention used by MATLAB for performing conversions between polar and Cartesian coordinates, and vice versa.

11- 10



a b
c d

FIGURE 11.11
(a) and (b) Boundaries of an irregular square and triangle. (c) and (d) Corresponding signatures.

Example 11.6 Signatures

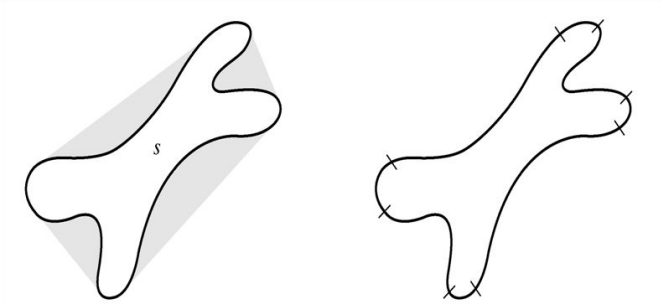
>>[st, angle, x0, y0]=signature(bs);

>>plot(angle, st)

11.2.4 Boundary Segments

The region boundary can be partitioned by following the contour of the arbitrary set S and marking the points at which a transition is made into or out of a component of the convex deficiency.

The region boundary can be partitioned by following the contour of the arbitrary set S and marking the points at which a transition is made into or out of a component of the convex deficiency.



a b

FIGURE 11.12
(a) A region *S* and its convex deficiency (shaded).
(b) Partitioned boundary.

11.2.5 Skeletons

An important approach for representing the structrual shape of a plane region is to reduce it to a gragh. This reduction may be accomplished by obtaining the skeleton of the region via a thinning (skeletonizing) algorithm.

Example 11.7 Computing the skeleton of a region.

>>f=im2double(f)

>>h=fspecial('gaussian', 25, 15);

>>g=imfilter(f, h, 'replicate');

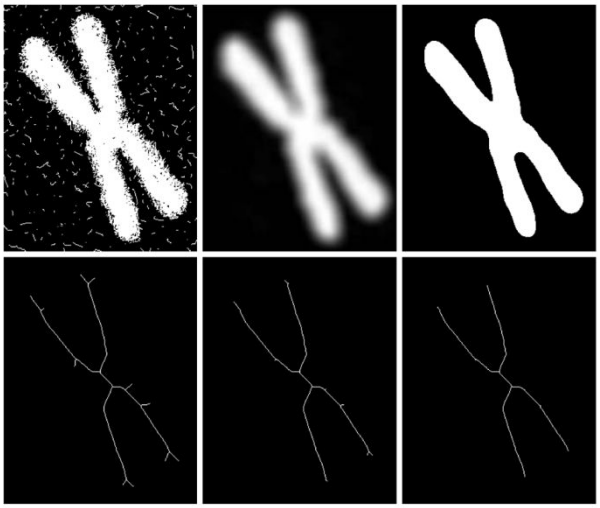
>>imshow(g)% Fig. 11.13(b)

>>g=im2bw(g, 1.5*graythresh(g));

>>figure, imshow(g)% Fig. 11.13(c)

>>s=bwmorph(g, 'skel', inf);% Fig. 11.13(d)

>>s=bwmorgh(s, 'spur', 8);% Fig. 11.13(e)



a b c
d e f

FIGURE 11.13 (a) Segmented human chromosome. (b) Image smoothed using a 25×25 Gaussian averaging mask with `sig = 15`. (c) Thresholded image. (d) Skeleton. (e) Skeleton after 8 applications of spur removal. (f) Result of 7 additional applications of spur removal.

11.3 Boundary Descriptors

11.3.1 Some Simple Descriptors

We extract the boundary of objects contained in image f using function bwperim.

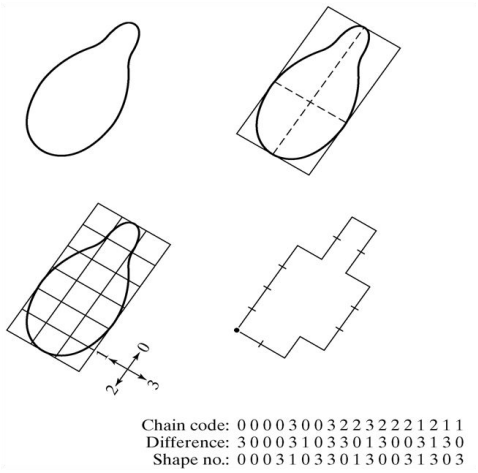
>>g=bwperim(f, conn)

Function diameter computes the diameter, major axis, minor axis, and basic rectangle of a boundary or region

>>s=diameter(L)

11.3.2 Shape Numbers

The order of a shape number is defined at the number of digits in its representation. The x-axis can be aligned with the major axis of a region or boundary by using function *x2majoraxis*：[B, theta]=x2majoraxis(A, B)。



a b
c d

FIGURE 11.14 Steps in the generation of a shape number.

11.3.3 Fourier Descriptors

Each coordinate pair can be treated as a complex ： *s(k) = x(k) + jy(k).*

The discrete Fourier transform (DFT) of s(k) is： $a(u)=\sum_{k=0}^{K-1} s(k)e^{-j\pi uk/K}$.

The inverse Fourier transform of these coefficients restors ： $s(k)=\frac{1}{K}\sum_{u=0}^{K-1} a(u)e^{j\pi uk/K}$.

Instead of all the Fourier coefficients, only the first P coefficients are used. The result is the following approximation to *s(k)*: $\hat{s}(k)=\frac{1}{P}\sum_{u=0}^{P-1} a(u)e^{-j2\pi uk/K}$.

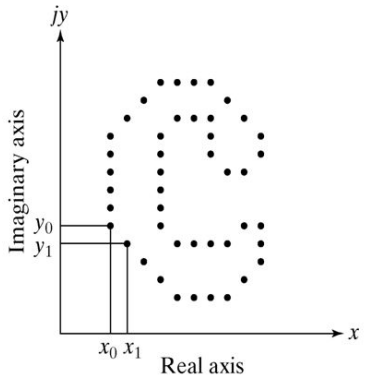


FIGURE 11.15 A digital boundary and its representation as a complex sequence. The points (x_0, y_0) and (x_1, y_1) are (arbitrarily) the first two points in the sequence.

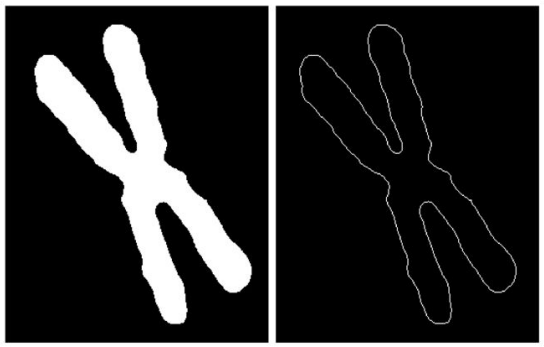
Example 11.8 Fourier descriptors:

```
>>b=boundaries(f);
>>b=b{1};
>>bim=bound2im(b, 344, 270);
```

>>z=frdescp(b);

>>z546=ifrdecp(z, 546);

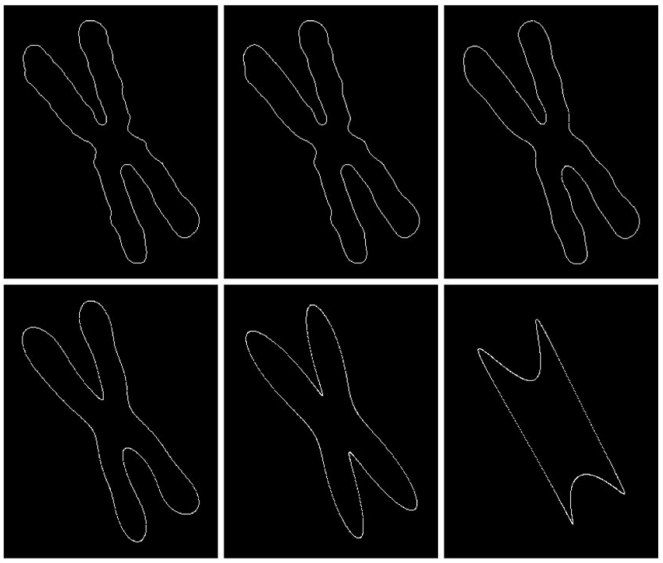
>>z546im=bound2im(z546, 344, 270);



a b

FIGURE 11.16
(a) Binary image. (b) Boundary extracted using function `boundaries`. The boundary has 1090 points.

11- 16



a b c
d e f

FIGURE 11.17 (a)–(f) Boundary reconstructed using 546, 110, 56, 28, 14, and 8 Fourier descriptors out of a possible 1090 descriptors.

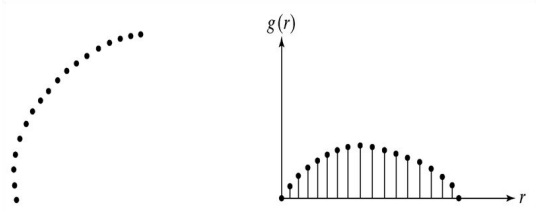
11- 17

11.3.4 Statistical Moments

The shape of 1-D boundary representations can be described quantitatively by using statistical moments. *r* is considered a random variable and the moments are

$$u_n = \sum_{i=0}^{K-1} (r_i - m)^n \, g(r_i)$$

$$m = \sum_{i=0}^{K-1} r_i g(r_i)$$



a b

FIGURE 11.18
(a) Boundary segment.
(b) Representation as a 1-D function.

11.4 Regional Descriptors

Function *regionprops*

D=regionprops(L, properties)

Table 11.1

TABLE 11.1 Regional descriptors computed by function `regionprops`.

Valid Strings for `properties`	Explanation
`'Area'`	The number of pixels in a region.
`'BoundingBox'`	1 × 4 vector defining the smallest rectangle containing a region. `BoundingBox` is defined by `[ul_corner width]`, where `ul_corner` is in the form `[x y]` and specifies the upper-left corner of the bounding box, and `width` is in the form `[x_width y_width]` and specifies the width of the bounding box along each dimension. Note that the `BoundingBox` is aligned with the coordinate axes and, in that sense, is a special case of the basic rectangle discussed in Section 11.3.1.
`'Centroid'`	1 × 2 vector; the center of mass of the region. The first element of `Centroid` is the horizontal coordinate (or *x*-coordinate) of the center of mass, and the second element is the vertical coordinate (or *y*-coordinate).
`'ConvexArea'`	Scalar; the number of pixels in `'ConvexImage'`.
`'ConvexHull'`	p × 2 matrix; the smallest convex polygon that can contain the region. Each row of the matrix contains the *x*- and *y*-coordinates of one of the *p* vertices of the polygon.
`'ConvexImage'`	Binary image; the convex hull, with all pixels within the hull filled in (i.e., set to `on`). (For pixels that the boundary of the hull passes through, `regionprops` uses the same logic as `roipoly` to determine whether the pixel is inside or outside the hull.) The image is the size of the bounding box of the region.
`'Eccentricity'`	Scalar; the eccentricity of the ellipse that has the same second moments as the region. The eccentricity is the ratio of the distance between the foci of the ellipse and its major axis length. The value is between 0 and 1, with 0 and 1 being degenerate cases (an ellipse whose eccentricity is 0 is a circle, while an ellipse with an eccentricity of 1 is a line segment).
`'EquivDiameter'`	Scalar; the diameter of a circle with the same area as the region. Computed as `sqrt(4*Area/pi)`.
`'EulerNumber'`	Scalar; equal to the number of objects in the region minus the number of holes in those objects.
`'Extent'`	Scalar; the proportion of the pixels in the bounding box that are also in the region. Computed as `Area` divided by the area of the bounding box.
`'Extrema'`	8 × 2 matrix; the extremal points in the region. Each row of the matrix contains the *x*- and *y*-coordinates of one of the points. The format of the vector is `[top-left, top-right, right-top, right-bottom, bottom-right, bottom-left, left-bottom, left-top]`.
`'FilledArea'`	The number of `on` pixels in `FilledImage`.
`'FilledImage'`	Binary image of the same size as the bounding box of the region. The `on` pixels correspond to the region, with all holes filled.
`'Image'`	Binary image of the same size as the bounding box of the region; the `on` pixels correspond to the region, and all other pixels are `off`.
`'MajorAxisLength'`	The length (in pixels) of the major axis† of the ellipse that has the same second moments as the region.
`'MinorAxisLength'`	The length (in pixels) of the minor axis† of the ellipse that has the same second moments as the region.
`'Orientation'`	The angle (in degrees) between the *x*-axis and the major axis† of the ellipse that has the same second moments as the region.
`'PixelList'`	A matrix whose rows are the `[x, y]` coordinates of the actual pixels in the region.
`'Solidity'`	Scalar; the proportion of the pixels in the convex hull that are also in the region. Computed as `Area/ConvexArea`.

† Note that the use of major and minor axis in this context is different from the major and minor axes of the basic rectangle discussed in Section 11.3.1. For a discussion of moments of an ellipse, see Haralick and Shapiro [1992].

Example 11.9 Using function regionprops

>>B=bwlabel(B);%convert B to a label matrix

>>D=regionprops(B, ‘area’, ‘boundingbox’)

>>w=[D, Area]

>>NR=length(w);

>>V=cat(1, D, Boundingbox)

11.4.2 Texture

An important approach for describing a region is to quantify its texture content.

1.Statistical Approaches

A frequently used approach for texture analysis is based on statistical properties of the intensity histogram. Ones class of such measures is based on statistical moments.

$$u_n = \sum_{i=0}^{L-1} (z_i - m)^n \, p(z_i), \quad m = \sum_{i=0}^{L-1} z_i p(z_i)$$

Table 11.2

TABLE 11.2 Some descriptors of texture based on the intensity histogram of a region.

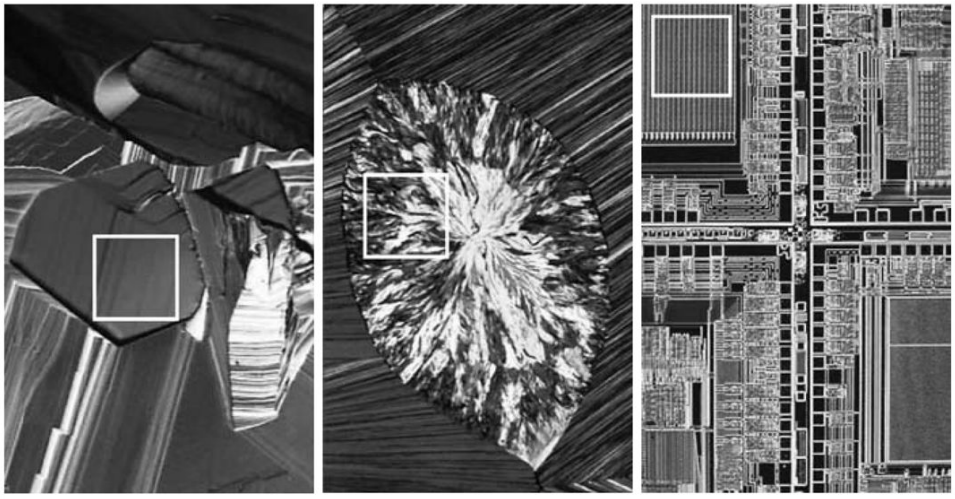
Moment	Expression	Measure of Texture
Mean	$m = \sum_{i=0}^{L-1} z_i p(z_i)$	A measure of average intensity.
Standard deviation	$\sigma = \sqrt{\mu_2(z)} = \sqrt{\sigma^2}$	A measure of average contrast.
Smoothness	$R = 1 - 1/(1 + \sigma^2)$	Measures the relative smoothness of the intensity in a region. R is 0 for a region of constant intensity and approaches 1 for regions with large excursions in the values of its intensity levels. In practice, the variance used in this measure is normalized to the range [0, 1] by dividing it by $(L - 1)^2$.
Third moment	$\mu_3 = \sum_{i=0}^{L-1} (z_i - m)^3 p(z_i)$	Measures the skewness of a histogram. This measure is 0 for symmetric histograms, positive by histograms skewed to the right (about the mean) and negative for histograms skewed to the left. Values of this measure are brought into a range of values comparable to the other five measures by dividing μ_3 by $(L - 1)^2$ also, which is the same divisor we used to normalize the variance.
Uniformity	$U = \sum_{i=0}^{L-1} p^2(z_i)$	Measures uniformity. This measure is maximum when all gray levels are equal (maximally uniform) and decreases from there.
Entropy	$e = -\sum_{i=0}^{L-1} p(z_i) \log_2 p(z_i)$	A measure of randomness.

Table 11.3

Texture	Average Intensity	Average Contrast	R	Third Moment	Uniformity	Entropy
Smooth	87.02	11.17	0.002	−0.011	0.028	5.367
Coarse	119.93	73.89	0.078	2.074	0.005	7.842
Periodic	98.48	33.50	0.017	0.557	0.014	6.517

TABLE 11.3 Texture measures for the regions shown in Fig. 11.19.

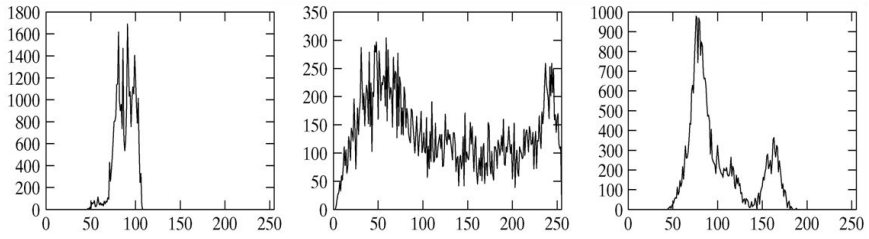
Example 11.10 Statistical texture measures



a b c

FIGURE 11.19 The subimages shown represent, from left to right, smooth, coarse, and periodic texture. These are optical microscope images of a superconductor, human cholesterol, and a microprocessor. (Original images courtesy of Dr. Michael W. Davidson, Florida State University.)

11- 19



a b c

FIGURE 11.20 Histograms corresponding to the subimages in Fig. 11.19.

11- 20

2.Spectral Measures of Texture

Spectral measures of texture are based on the Fourier spectrum, which is ideally suited for describing the directionality of periodic or almost periodic 2-D patterns in an image.

$$S(r)=\sum_{\theta=0}^{\pi} S_{\theta}(r)$$

$$S(\theta)=\sum_{r=1}^{R_0} S_{r}(\theta)$$

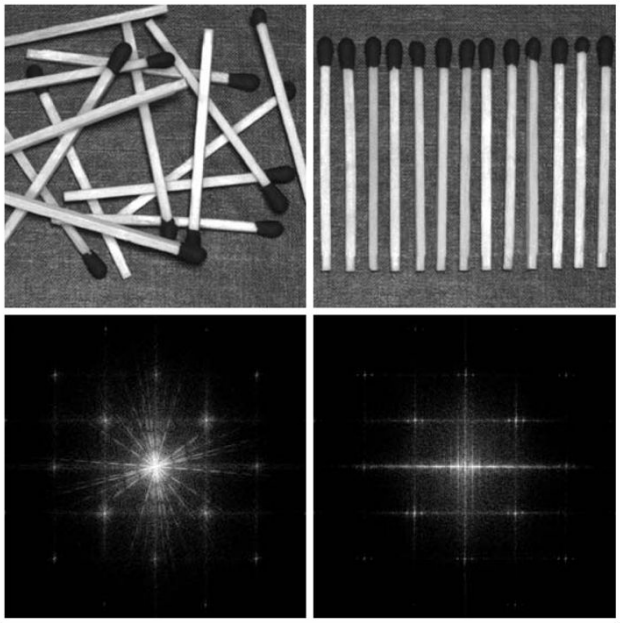
Function *specxture*

[srad, sang, S]=specxture(f), Where srad is *S(r)*, sang is $S(\theta)$, and S is the spectrum image.

Example Computing spectral texture.

>>[srad, sang, S]=specxture(f);

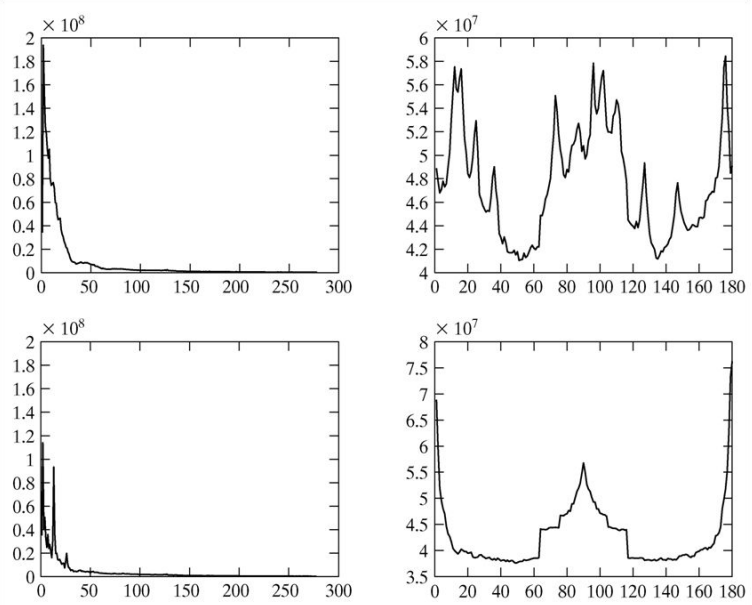
>>axis([horzmin horzmax vertmin vertmax]);



a b
c d

FIGURE 11.21
(a) and (b) Images of unordered and ordered objects. (c) and (d) Corresponding spectra.

11- 21



a b
c d

FIGURE 11.22
Plots of (a) $S(r)$ and (b) $S(\theta)$ for the random image. (c) and (d) are plots of $S(r)$ and $S(\theta)$ for the ordered image.

11- 22

11.4.3 Moment Invariants

The 2-D moment of order *(p+q)* of a digital image *f(x,y)*:

$$m_{pq} = \sum_x \sum_y x^p y^q f(x,y).$$

The corresponding central moment

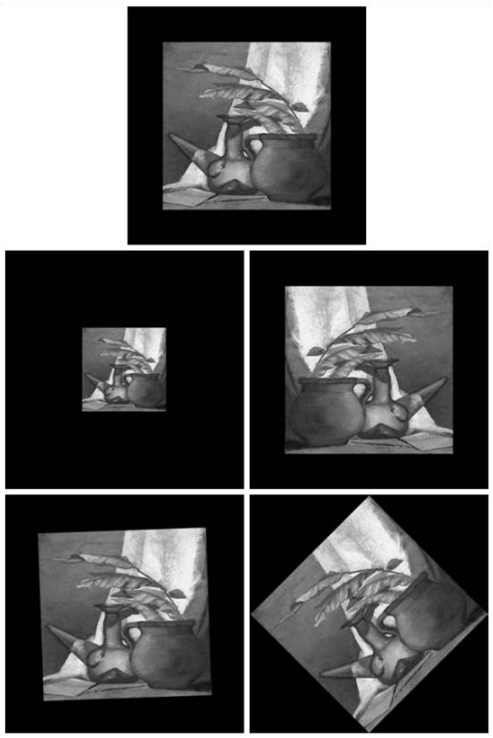
$$u_{pq} = \sum_x \sum_y (x-\bar{x})^p (y-\bar{y})^q f(x,y)$$

$$\bar{x} = \frac{m_{10}}{m_{00}} \qquad \bar{y} = \frac{m_{01}}{m_{00}}$$

The normalized central moment of order *(p + q)*:

$$\eta_{pq} = \frac{u_{pq}}{u_{00}^r}$$

$$r = \frac{p+q}{2} + 1$$



a
b c
d e

FIGURE 11.23
(a) Original, padded image. (b) Half size image. (c) Mirrored image. (d) Image rotated by 2°. (e) Image rotated 45°. The zero padding in (a) through (d) was done to make the images consistent in size with (e) for viewing purposes only.

Table 11.4

Invariant (|log|)	Original	Half Size	Mirrored	Rotated 2°	Rotated 45°
ϕ_1	6.600	6.600	6.600	6.600	6.600
ϕ_2	16.410	16.408	16.410	16.410	16.410
ϕ_3	23.972	23.958	23.972	23.978	23.973
ϕ_4	23.888	23.882	23.888	23.888	23.888
ϕ_5	49.200	49.258	49.200	49.200	49.198
ϕ_6	32.102	32.094	32.102	32.102	32.102
ϕ_7	47.953	47.933	47.850	47.953	47.954

TABLE 11.4
The seven moment invariants of the images in Figs. 11.23(a) through (e). Note the use of the magnitude of the log in the first column.

Example 11.12 Moment invariants

```
>>fp=padarray(f, [84 84], 'both');
>>fhs=f(1:2:end, 1:2:end);
>>fhsp=padarray(fhs, [184 184], 'both');
>>g=imrotate(f, angle, method, 'crop');
>>fr2=imrotate(f, 2, 'billinear');
>>fr2p=padarray(fr2, [76 76], 'both');
>>fr45=imrotate(f, 45, 'billinear');
>>phiorig=abs(log(invmoments(f)));
>>phihalf=abs(log(invmoments(fhs)));
>>phimirror=abs(log(invmoments(fm)));
>>phirot2=abs(log(invmoments(fr2)));
>>phirot45=abs(log(invmoments(fr45)));
```

11.5 Using Principal Components for Description

The mean vector of a vector population can be approximated by the sample average

$$\mathbf{m}_\mathbf{x} = \frac{1}{K}\sum_{k=1}^{K}\mathbf{x}_\mathbf{k}$$

The n*n covariance matrix of the population can be approximated

$$\mathbf{C}_\mathbf{x} = \frac{1}{K-1}\sum_{k=1}^{K}\left(\mathbf{x}_\mathbf{k}-\mathbf{m}_\mathbf{k}\right)\left(\mathbf{x}_\mathbf{k}-\mathbf{m}_\mathbf{k}\right)^T$$

The principal components transform

$$\mathbf{y} = \mathbf{A}\left(\mathbf{x}-\mathbf{m}_\mathbf{x}\right)$$
$$\mathbf{x} = \mathbf{A}^\mathbf{T}\mathbf{y}+\mathbf{m}_\mathbf{x}$$

The reconstruction is an approximation

$$\hat{\mathbf{x}} = \mathbf{A}_\mathbf{q}^\mathbf{T}\mathbf{y}+\mathbf{m}_\mathbf{x}$$

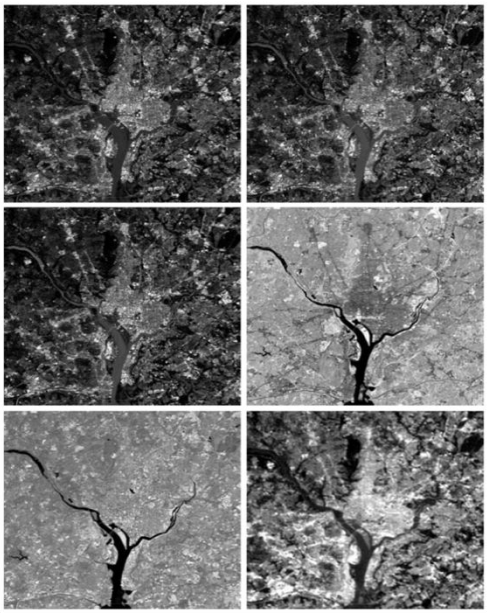
The mean square error

$$e_{ms} = \sum_{j=1}^{n}\lambda_j - \sum_{j=1}^{q}\lambda_j = \sum_{j=q+1}^{n}\lambda_j$$

Example 11.13 Principal components.

```
>>S=cat(3, f1, f2, f3, f4, f5, f6);
>>[X,R]=imstack2vectors(S);
>>P=princomp(X, 6);
```

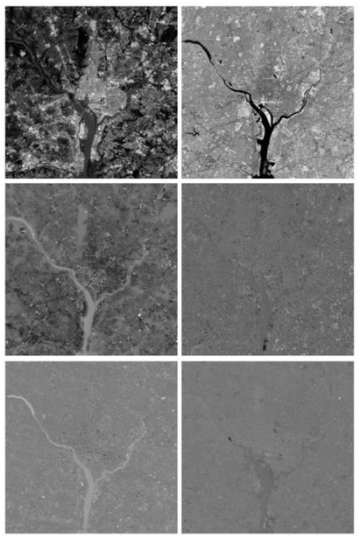
```
>>g1=P.Y(:, 1);
>>g1=reshape(g1, 512, 512);
>>imshow(g1, []);
>>d=diag(P.Cy);
>>P=princomp(X, 2);
>>h1=P.X(:, 1);
>>h1=reshape(h1, 512, 512);
>>D1=double(f1)-double(h1);
>>D1=gscale(D1);
>>imshow(D1);
```



a b
c d
e f

FIGURE 11.25
Six multispectral images in the (a) visible blue, (b) visible green, (c) visible red, (d) near infrared, (e) middle infrared, and (f) thermal infrared bands. (Images courtesy of NASA.)

11- 24



a b
c d
e f

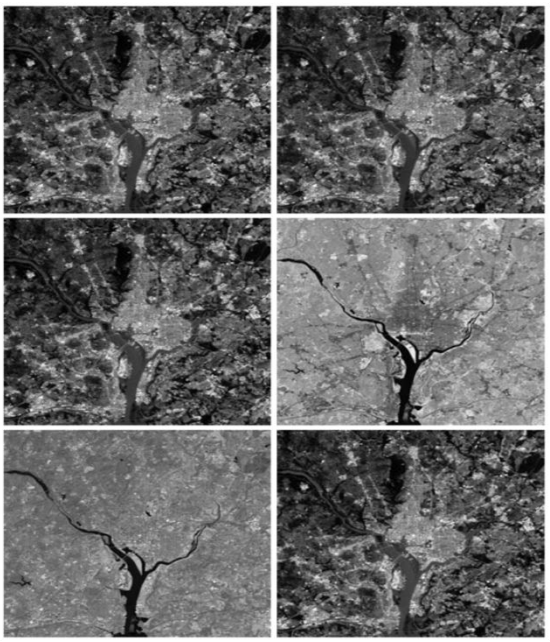
FIGURE 11.26
Principal-component images corresponding to the images in Fig. 11.25.

11- 25

Table 11.5

λ_1	λ_2	λ_3	λ_4	λ_5	λ_6
10352	2959	1403	203	94	31

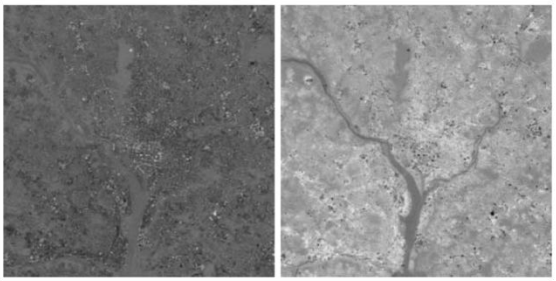
TABLE 11.5
Eigenvalues of `P.Cy` when `q = 6`.



a b
c d
e f

FIGURE 11.27
Multispectral images reconstructed using only the two principal-component images with the largest variance. Compare with the originals in Fig. 11.25.

11- 26



a b

FIGURE 11.28
(a) Difference between Figs. 11.27(a) and 11.25(a). (b) Difference between Figs. 11.27(f) and 11.25(f). Both images are scaled to the full [0, 255] 8-bit intensity scale.

11- 27

Summary

The representation of objects or regions that have been segmented out of an image is an early step in the preparation of image data for subsequent use in automation.

The choice of one type of descriptor over another is dictated to a large degree by the problem at hand. This is one of the principal reasons why the solution of image processing problem is aided significantly by having a flexible prototyping environment in which existing functions can be integrated with new code to gain flexibility and reduce development time.

Chapter 12 Object Recognition

Preview

We conclude the book with a discussion and development of several M-functions for region and/or boundary recognition, which in this chapter we call objects or patterns.

Two approaches to computerized pattern recognition:

Decision-theoretic: deals with patterns described using quantitative descriptors such as length, area, texture.

Structural: deals with patterns best represented by symbolic information, such as strings, and properties and relationships between those symbols.

12.1 Background

A pattern is an arrangement of descriptors.The name feature is used often to denote a descriptor.A pattern class is a family of patterns that share a set of common properties. Pattern classes are denoted ω_1, $\omega_2, \cdots \omega_W$,where W is the number of classes.

The two principal pattern arrangements are vectors (for quantitative descriptions) and strings (for structural descriptions).

Pattern vectors are represented by bold lowercase letters, such as x, y, and z, and have the $n \times 1$ vectors form:

$$x = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}$$

Where each component represents the ith descriptor and n is the total number of such descriptors associated with the pattern.

The nature of the components of a pattern vector x depends on the approach used to describe the physical pattern itself. In some applications, pattern characteristics are best described by structural relationships. The material in the following sections is

representative of techniques for implementing pattern recognition solutions in MATLAB.

12.2 Computing Distance Measures in MATLAB

This section deals with vectorizing distance computations that otherwise would involve for or while loops. The Euclidean distance between two n-dimensional vectors x and y is defined as scalar：

$$d(x,y) = \|x-y\| = \|y-x\| = [(x_1-y_1)^2 + \cdots + (x_n-y_n)^2]^{1/2}$$

we compute it using MATLAB's function norm: $d = norm(x-y)$.

n-dimensions vectors are arranged as the rows of a $p \times n$ matrix X, y has to be of dimension $1 \times np$. The distance between y and each element of X in the $p \times 1$ vector $d = sqrt(sum(abs(x - repmat(y, p, 1)).^2, 2))$,where d(i) is the Euclidean distance between y and the ith row of X[i.e.,x(I,:)].

Suppose we have two vectors populations X, of dimension $p \times n$, and Y, of dimension $q \times n$.

The matrix containing the distance between rows of these two populations can be obtained using the expression

$$D = sqrt(sum(abs(repmat(permute(X,[132]),[1q1])...$$
$$-repmat(permute(Y,[312]),[p11])).^2,3))$$

where D(i.j) is the Euclidean distance between the ith and ith rows of the populations; that is the distance between X(i,:) and Y(j,:).

The syntax for function permute is

B=permute (A, order)

This function reorders the dimensions of A according to the elements of the vector order (the elements of this vector must be unique).

In addition to the expressions just discussed, we use in this chapter a distance measure from a vector y to the mean m_x of a vector population, weighted inversely by the covariance matrix, C_x , of the population. This metric, called the Mahalanobis distance, is defined as $d(y, m_x) = (y - m_x)^T C_x^{-1} (y - m_x)$.

12.3 Recognition Based on Decision-Theoretic Methods

Decision-theoretic approaches to recognition are based on the use of decision (also called discriminant) functions.

Let $x = (x_1, x_2, ..., x_n)^T$ represent an n-dimensional pattern vector. For W pattern classes $\omega_1, \omega_2, ..., \omega_W$, the basic problem in decision-theoretic pattern recognition is to find W decision functions $d_1(x), d_2(x), ..., d_W(x)$ with the property that if a pattern x belongs to class w_i , then $d_i(x) = d_j(x)$, $j = 1, 2, ..., W; j \neq i$.

The decision boundary separating class w_i from w_j is given by values of x for which $d_i(x) = d_j(x)$,or, equivalently, by values of x for which $d_i(x) - d_j(x) = 0$ .Common practice is to express the decision boundary between two classes by the single function $d_{ij}(x) = d_i(x) - d_j(x) = 0$ .Thus $d_{ij}(x) > 0$ for patterns of class w_i and $d_{ij}(x) < 0$ for patterns of class w_j .

12.3.1 Forming Pattern Vectors

Pattern vectors can be formed from quantitative descriptors. Another approach used quite frequently when dealing with (registered) multispectral images is to stack the images and then form vectors from corresponding pixels in the images.

12.3.2 Pattern Matching Using Minimum-Distance Classifiers

Suppose that each pattern class w_j is characterized by a mean vector m_j .Use the mean vector of each population of training vectors as being representative of that class of vectors:

$$m_j = \frac{1}{N_j} \sum_{X \in \omega_j} x \text{ , j=1,2,…W}$$

where N_j is the number of training pattern vectors from class w_j and the summation is taken over these vectors.

One way to determine the class membership of an unknown pattern vector x is to assign it to the class of its closest prototype. The problem is reduced to computing the distance measures:

$$D_j(x) = \|x - m_j\|, \text{ j=1,2,...W}$$

We assign x to class w_i if $D_i(x)$ is the smallest distance (the best match).

Suppose that all the mean vectors are organized as rows of a matrix M. The distance from an arbitrary pattern x to all the mean vectors is:

$$d = sqrt(sum(abs(M - repmat(x, W, 1)).^2, 2))$$

The minimum of d determines the class membership of pattern vector x:

$$class = find(d == \min(d));$$

Selecting the smallest distance is equivalent to evaluating the functions

$$d_j(x) = x^T m_j - \frac{1}{2} m_j^T m_j \text{ , j=1,2,...,W}$$

and assigning x to class w_i if $D_i(x)$ yields the largest numerical value.

The decision boundary between classer w_i and w_j for a minimum distance classifier is

$$d_{ij}(x) = d_i(x) - d_j(x)$$
$$= x^T(m_i - m_j) - \frac{1}{2}(m_i - m_j)^T(m_i + m_j) = 0$$

The surface given by this equation is the perpendicular bisector of the line segment joining m_i and m_j.

12.3.3Matching by Correlation

Given an image f(x,y), the correlation problem is to find all places in the image that match a given subimage (also called a mask or template) w(x,y).

One approach for finding matches is to treat w(x,y) as a spatial filter and compute the sum of products for each location of w in f. The best matches of w(x,y) in f(x,y) are the locations of the maximum values in the resulting correlation image.

Practical implementations of spatial correlation typically rely on hardware-oriented solutions.

For prototyping, an alternative approach is to implement correlation in the frequency domain, making use of the correlation theorem relates spatial correlation to the product of the image transforms.

The correlation theorem:

$$f(x,y) \circ w(x,y) \Leftrightarrow F(u,v)H^*(u,v)$$

$$f(x,y)w^*(x,y) \Leftrightarrow F(u,v) \circ H(u,v)$$

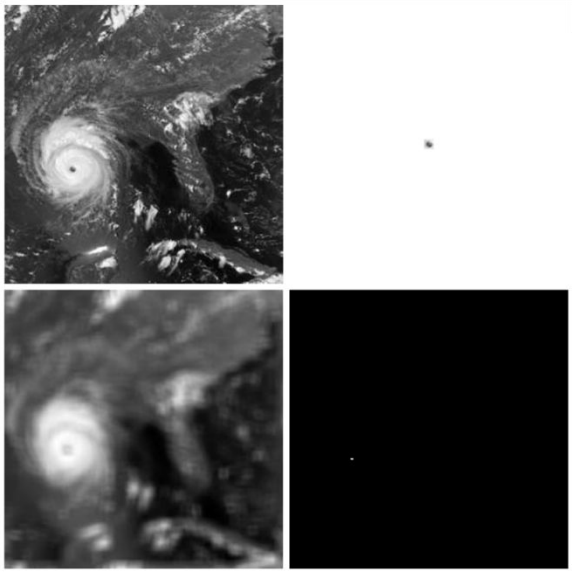
An M-function for the first correlation:

```
Function g=dftcorr(f,w)
%DFTCORR 2-D correlation in the frequency domain.
%G= DFTCORR(F,W) performs the correlation of a mask W,with image F.
  [M,N]=size(f);
f=fft2(f);
w=conj(fft2(w,M,N));
g=real(ifft2(w.*f))
```

Examples12.1

We wish to find the location of the best match in (a) of the hurricane eye image in Fig.12.1(b). Fig.12.1 (c) is the result of the following commands:

```
>>g=dftcorr(f,w);
>>gs=gscale(g);
>>imshow(gs)
```



a b
c d

FIGURE 12.1
(a) Multispectral image of Hurricane Andrew.
(b) Template.
(c) Correlation of image and template.
(d) Location of the best match.
(Original image courtesy of NOAA.)

To find the location of the best match implies finding the location(s) of the highest value in the correlation images:

```
>>[I,J]=find(g==max(g(:)))
I = 554
J = 203
```

Another approach is to threshold the correlation image near its maximum, or threshold its scaled version, gs, whose highest value is known to be 255. Fig.12.1 (d)

was obtained using the command：

>>imshow (gs>254)

12.3.4 Optimum Statistical Classifiers

The well-known Bayes classifier for 0-1 loss function has decision function (PDF) of the form

$$d_j(x) = p(x/\omega_j)P(\omega_j)\ ,\quad \text{j=1,2,…,W}$$

Where $p(x/\omega_j)$ is the probability density function of the pattern vectors of class w_j , and $P(\omega_j)$ is the probability (a scalar) that class ω_j occurs.

When the PDF are n-dimensional Gaussian PDF :

$$p(x/\omega_j) = \frac{1}{(2\pi)^{n/2}\left|C_j\right|^{1/2}} e^{-\frac{1}{2}[(x-m_j)^T C_j^{-1}(x-m_j)]}$$

where C_j and m_j are the covariance matrix and mean vector of the pattern population of class w_j , and $|C_j|$ is the determinant of C_j .

We use decision functions of the form:

$$d_j(x) = \ln[p(x/\omega_j)P(\omega_j)] = \ln p(x/\omega_j) + \ln P(\omega_j)$$

Substituting the expression for the Gaussian PDF gives the equation:

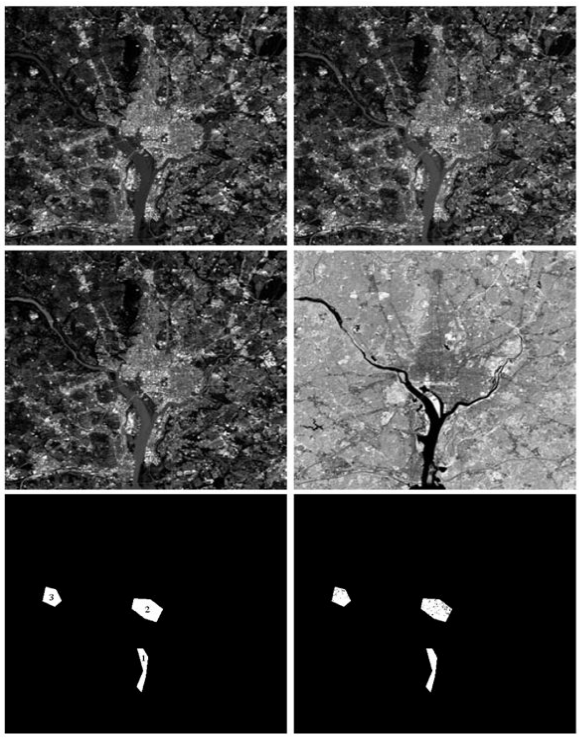
$$d_j(x) = \ln P(\omega_j) - \frac{n}{2}\ln 2\pi - \frac{1}{2}\ln\left|C_j\right| - \frac{1}{2}[(x-m_j)^T C_j^{-1}(x-m_j)]$$

The term(n/2) can be deleted, yielding the decision functions:

$$d_j(x) = \ln P(\omega_j) - \frac{1}{2}\ln\left|C_j\right| - \frac{1}{2}[(x-m_j)^T C_j^{-1}(x-m_j)]$$

Examples12.2　Bayes classification of multispectral data

```
B=roipoly(f)
Stack=cat(3,f1,f2,f3,f4)
[X,R]=imstack2vectors(stack,B)
[C1,m1]=covmatrix(Y1)
CA=cat(3,C1,C2,C3)
MA=cat(2,m1,m2,m3)
dY1=bayesgauss(Y1,CA,MA)
IY1=find(dY1~=1)
```



a b
c d
e f

FIGURE 12.2 Bayes classification of multispectral data. (a)–(c) Images in the blue, green, and red visible wavelengths. (d) Infrared image. (e) Mask showing sample regions of water (1), urban development (2), and vegetation (3). (f) Results of classification. The black dots denote points classified incorrectly. The other (white) points in the regions were classified correctly. (Original images courtesy of NASA.)

Table 12.1 summarizes the recognition results obtained with the training and independent pattern sets.

TABLE 12.1 Bayes classification of multispectral image data.

Training Patterns						Independent Patterns					
Class	No. of Samples	Classified into Class 1	2	3	% Correct	Class	No. of Samples	Classified into Class 1	2	3	% Correct
1	484	482	2	0	99.6	1	483	478	3	2	98.9
2	933	0	885	48	94.9	2	932	0	880	52	94.4
3	483	0	19	464	96.1	3	482	0	16	466	96.7

12.3.5 Adaptive Learning Systems

The principal approach in use today for this type of classification is based on neural network.

12.4 Structural Recognition

Structural recognition techniques are based on representing objects of interest as strings, trees, or graphs and then defining descriptors and recognition rules based on those representations.

The key difference between decision-theoretic and structural methods is that the former uses quantitative descriptors expressed in the form of numeric vectors. Structural techniques , on the other hand, deal principally with symbolic information.

Strings are by far the most common representation used in structural recognition, so we focus on this approach in this section.

12.4.1Working with Strings in MATLAB

In MATLAB, a string is a one-dimensional array whose components are the numeric codes for the characters in the string. A string is defined by enclosing its characters in single quotes. Table 12.2 lists the principal MATLAB functions that deal with strings.

Table 12.2

TABLE 12.2 MATLAB's string-manipulation functions.

Category	Function Name	Explanation
General	`blanks`	String of blanks.
	`cellstr`	Create cell array of strings from character array. Use function `char` to convert back to a character string.
	`char`	Create character array (string).
	`deblank`	Remove trailing blanks.
	`eval`	Execute string with MATLAB expression.
String tests	`iscellstr`	True for cell array of strings.
	`ischar`	True for character array.
	`isletter`	True for letters of the alphabet.
	`isspace`	True for whitespace characters.
String operations	`lower`	Convert string to lowercase.
	`regexp`	Match regular expression.
	`regexpi`	Match regular expression, ignoring case.
	`regexprep`	Replace string using regular expression.
	`strcat`	Concatenate strings.
	`strcmp`	Compare strings (see Section 2.10.5).
	`strcmpi`	Compare strings, ignoring case.
	`strfind`	Find one string within another.
	`strjust`	Justify string.
	`strmatch`	Find matches for string.
	`strncmp`	Compare first n characters of strings.
	`strncmpi`	Compare first n characters, ignoring case.
	`strread`	Read formatted data from a string. See Section 2.10.5 for a detailed explanation.
	`strrep`	Replace a string within another.
	`strtok`	Find token in string.
	`strvcat`	Concatenate strings vertically.
	`upper`	Convert string to uppercase.
String to number conversion	`double`	Convert string to numeric codes.
	`int2str`	Convert integer to string.
	`mat2str`	Convert matrix to a string suitable for processing with the `eval` function.
	`num2str`	Convert number to string.
	`sprintf`	Write formatted data to string.
	`str2double`	Convert string to double-precision value.
	`str2num`	Convert string to number (see Section 2.10.5).
	`sscanf`	Read string under format control.
Base number conversion	`base2dec`	Convert base B string to decimal integer.
	`bin2dec`	Convert binary string to decimal integer.
	`dec2base`	Convert decimal integer to base B string.
	`dec2bin`	Convert decimal integer to binary string.
	`dec2hex`	Convert decimal integer to hexadecimal string.
	`hex2dec`	Convert hexadecimal string to decimal integer.
	`hex2num`	Convert IEEE hexadecimal to double-precision number.

Table 12.3

Metacharacters	Usage
.	Matches any one character.
[ab...]	Matches any one of the characters, (a, b, . . .), contained within the brackets.
[^ab...]	Matches any character except those contained within the brackets.
?	Matches any character zero or one times.
*	Matches the preceding element zero or more times.
+	Matches the preceding element one or more times.
{num}	Matches the preceding element num times.
{min, max}	Matches the preceding element at least min times, but not more than max times.
|	Matches either the expression preceding or following the metacharacter |.
^chars	Matches when a string begins with chars.
chars$	Matches when a string ends with chars.
\<chars	Matches when a word begins with chars.
chars\>	Matches when a word ends with chars.
\<word\>	Exact word match.

TABLE 12.3 Some of the metacharacters used in regular expressions for matching. See the `regular expressions` help page for a complete list.

12.4.2 String Matching

In addition to the string matching and comparing functions in Table 12.2, it is often useful to have available measures of similarity that behave much like the distance measures discussed in Section 12.2. We illustrate this approach using a measure defined as follows.

Suppose that two region boundaries, a and b, are coded into strings $a_1a_2...a_m$ and $b_1b_2...b_n$, respectively. Let α denote the number of matches between these two strings, where a match is said to occur in the kth position if $a_k = b_k$. The number of symbols that do not match is

$$\beta = \max(|a|, |b|) - \alpha$$

where $|\arg|$ is the length of the string in the argument.

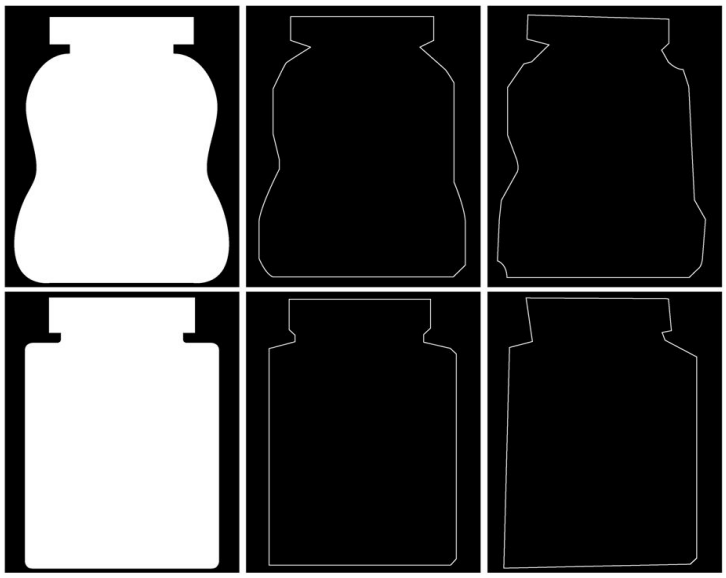
A simple measure of similarity between a and b is the ratio

$$R = \frac{\alpha}{\beta} = \frac{\alpha}{\max(|a|, |b|) - \alpha}$$

It is infinite for a perfect match an 0 when none of the corresponding symbols in a and b match (α is 0 in this case).

One way to register two strings is to shift one string with respect to the other until a maximum value of R is obtained. A more efficient approach is to define the same starting point for all strings based on normalizing the boundaries with respect to size and orientation before their string representation is extracted.

Examples12.3 Object recognition based on string matching



a b c
d e f

FIGURE 12.3 (a) An object. (b) Its minimum perimeter polygon obtained using function `minperpoly` with a cell size of 8. (c) A typical noisy boundary. (d)–(f) The same sequence for another object.

[xn,yn]=randvertex(x,y,npix)

Angles=polyangles(x,y)

S=floor(angles/45)+1

S=int2str(s)

R=strsimilarity(sl1,sl2)

The results obtained using five typical strings are summarized in Table 12.4. Table12.5 shows the same type of computation involving five strings of class 2 against themselves. Table 12.6 shows values of the similarity measure between the strings of class 1and class 2.

R	s_{11}	s_{12}	s_{13}	s_{14}	s_{15}
s_{11}	Inf				
s_{12}	9.33	Inf			
s_{13}	26.25	12.31	Inf		
s_{14}	16.36	9.33	14.16	Inf	
s_{15}	22.22	14.17	14.01	19.02	Inf

TABLE 12.4
Values of similarity measure, R, between the strings of class 1. (All values shown are ×10.)

R	s_{21}	s_{22}	s_{23}	s_{24}	s_{25}
s_{21}	Inf				
s_{22}	10.00	Inf			
s_{23}	13.33	13.33	Inf		
s_{24}	7.50	13.31	18.00	Inf	
s_{25}	13.33	7.51	18.12	10.01	Inf

TABLE 12.5
Values of similarity measure, R, between the strings of class 2. (All values shown are ×10.)

R	s_{11}	s_{12}	s_{13}	s_{14}	s_{15}
s_{21}	2.03	0.01	1.15	1.17	0.75
s_{22}	1.15	1.61	1.16	0.75	2.07
s_{23}	2.08	1.15	2.08	2.06	2.08
s_{24}	1.60	1.62	1.59	1.14	2.61
s_{25}	1.61	0.36	0.74	1.60	1.16

TABLE 12.6
Values of similarity measure, R, between the strings of classes 1 and 2. (All values shown are ×10.)

Summary

Although the material in the present chapter is introductory in nature, the topics covered are fundamental to understanding the state of the art in object recognition.

Having finished study of the material in the preceding twelve chapters, the reader is now in the position of being able to master the fundamentals of how to prototype software solutions of image-processing problems using MATLAB and Image Processing Toolbox functions.