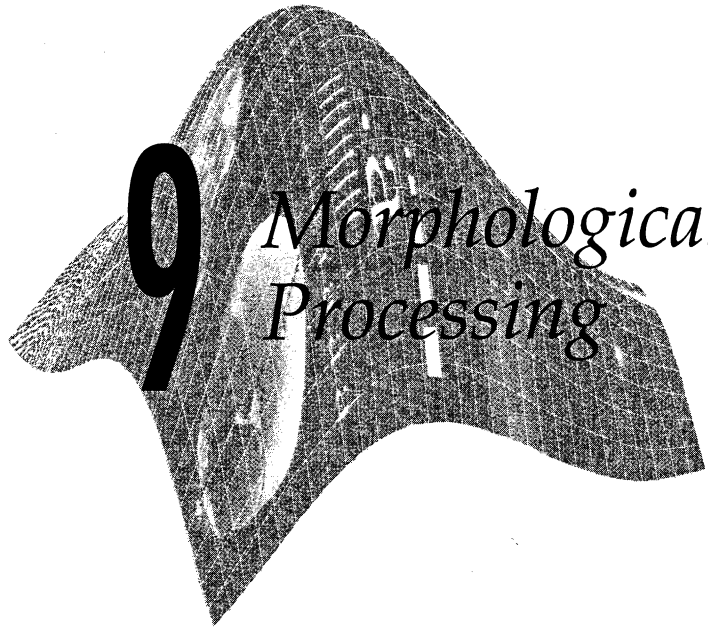


9 Morphological Image Processing

Preview

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

In Section 9.1 we define several set theoretic operations, introduce binary images, and discuss binary sets and logical operators. In Section 9.2 we define two fundamental morphological operations, *dilation* and *erosion*, in terms of the union (or intersection) of an image with a translated shape (*structuring element*). Section 9.3 deals with combining erosion and dilation to obtain more complex morphological operations. Section 9.4 introduces techniques for labeling connected components in an image. This is a fundamental step in extracting objects from an image for subsequent analysis.

Section 9.5 deals with *morphological reconstruction*, a morphological transformation involving two images, rather than a single image and a structuring element, as is the case in Sections 9.1 through 9.4. Section 9.6 extends morphological concepts to gray-scale images by replacing set union and intersection with maxima and minima. Most binary morphological operations have natural extensions to gray-scale processing. Some, like morphological reconstruction, have applications that are unique to gray-scale images, such as peak filtering.

The material in this chapter begins a transition from image-processing methods whose inputs and outputs are images, to image analysis methods, whose outputs in some way describe the contents of the image. Morphology is

a cornerstone of the mathematical set of tools underlying the development of techniques that extract "meaning" from an image. Other approaches are developed and applied in the remaining chapters of the book.

9.1 Preliminaries

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

9.1.1 Some Basic Concepts from Set Theory

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

Let A be a set in Z^2, the elements of which are pixel coordinates (x, y). If $w = (x, y)$ is an element of A, then we write

$$w \in A$$

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

$$w \notin A$$

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

$$B = \{w|\text{condition}\}$$

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

$$A^c = \{w|w \notin A\}$$

This set is called the *complement* of A.

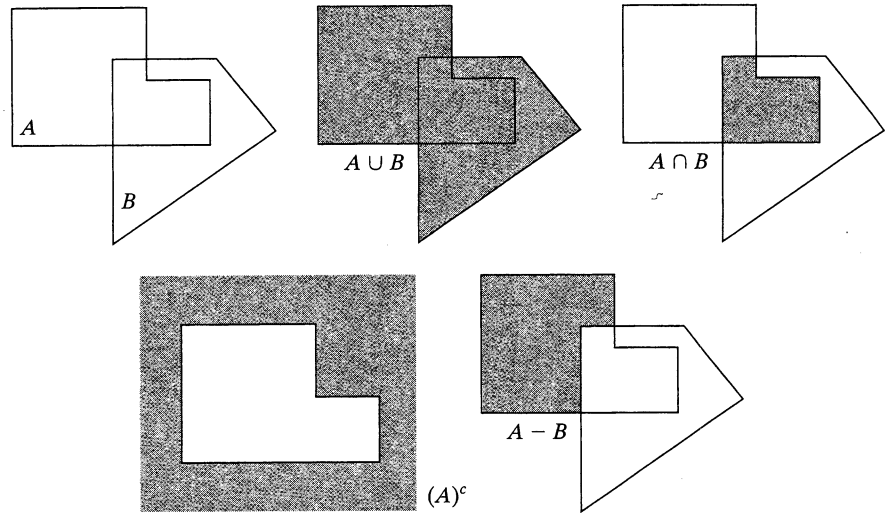
The *union* of two sets, denoted by

$$C = A \cup B$$

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

$$C = A \cap B$$

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



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.

The *difference* of sets A and B, denoted $A - B$, is the set of all elements that belong to A but not to B:

$$A - B = \{w | w \in A, w \notin B\}$$

Figure 9.1 illustrates these basic set operations. The result of each operation is shown in gray.

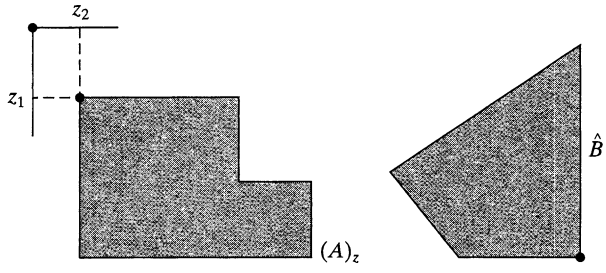
In addition to the preceding basic operations, morphological operations often require two operators that are specific to sets whose elements are pixel coordinates. The *reflection* of set B, denoted $\hat{B}$, is defined as

$$\hat{B} = \{w | w = -b, \text{for } b \in B\}$$

The *translation* of set A by point $z = (z_1, z_2)$, denoted $(A)_z$, is defined as

$$(A)_z = \{c | c = a + z, \text{for } a \in A\}$$

Figure 9.2 illustrates these two definitions using the sets from Fig. 9.1. The black dot identifies the *origin* assigned (arbitrarily) to the sets.



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.

9.1.2 Binary Images, Sets, and Logical Operators

The language and theory of mathematical morphology often present a dual view of binary images. As in the rest of the book, a binary image can be viewed as a bivalued *function* of x and y. Morphological theory views a binary image as the *set* of its foreground (1-valued) pixels, the elements of which are in Z^2. Set operations such as union and intersection can be applied directly to binary image sets. For example, if A and B are binary images, then $C = A \cup B$ is also a binary image, where a pixel in C is a foreground pixel if either or both of the corresponding pixels in A and B are foreground pixels. In the first view, that of a function, C is given by

$$C(x, y) = \begin{cases} 1 & \text{if either } A(x, y) \text{ or } B(x, y) \text{ is 1, or if both are 1} \\ 0 & \text{otherwise} \end{cases}$$

Using the set view, on the other hand, C is given by

$$C = \{(x, y) | (x, y) \in A \text{ or } (x, y) \in B \text{ or } (x, y) \in (A \text{ and } B)\}$$

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.

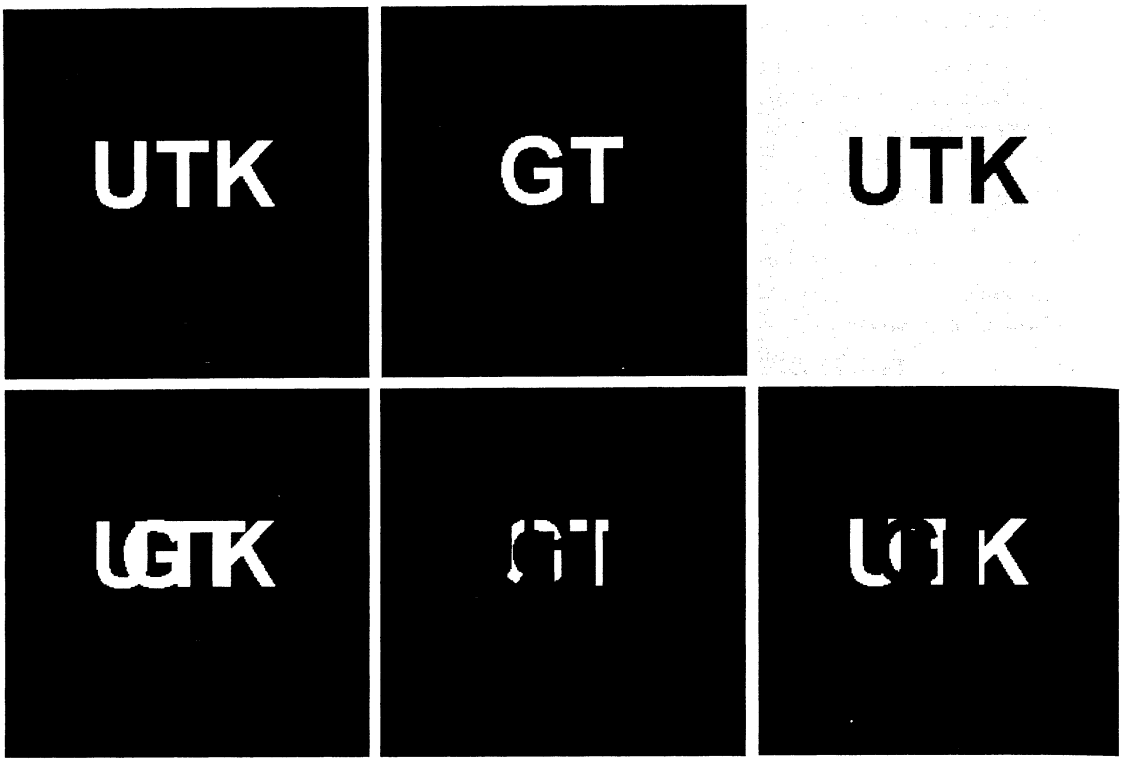
As a simple illustration, Fig. 9.3 shows the results of applying several logical operators to two binary images containing text. (We follow the IPT convention that foreground (1-valued) pixels are displayed as white.) The image in Fig. 9.3(d) is the union of the "UTK" and "GT" images; it contains all the foreground pixels from both. By contrast, the intersection of the two images [Fig. 9.3(e)] consists of the pixels where the letters in "UTK" and "GT" overlap. Finally, the set difference image [Fig. 9.3(f)] shows the letters in "UTK" with the pixels "GT" removed.

9.2 Dilation and Erosion

The operations of *dilation* and *erosion* are fundamental to morphological image processing. Many of the algorithms presented later in this chapter are based on these operations, which are defined and illustrated in the discussion that follows.

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.



a b c
d e f

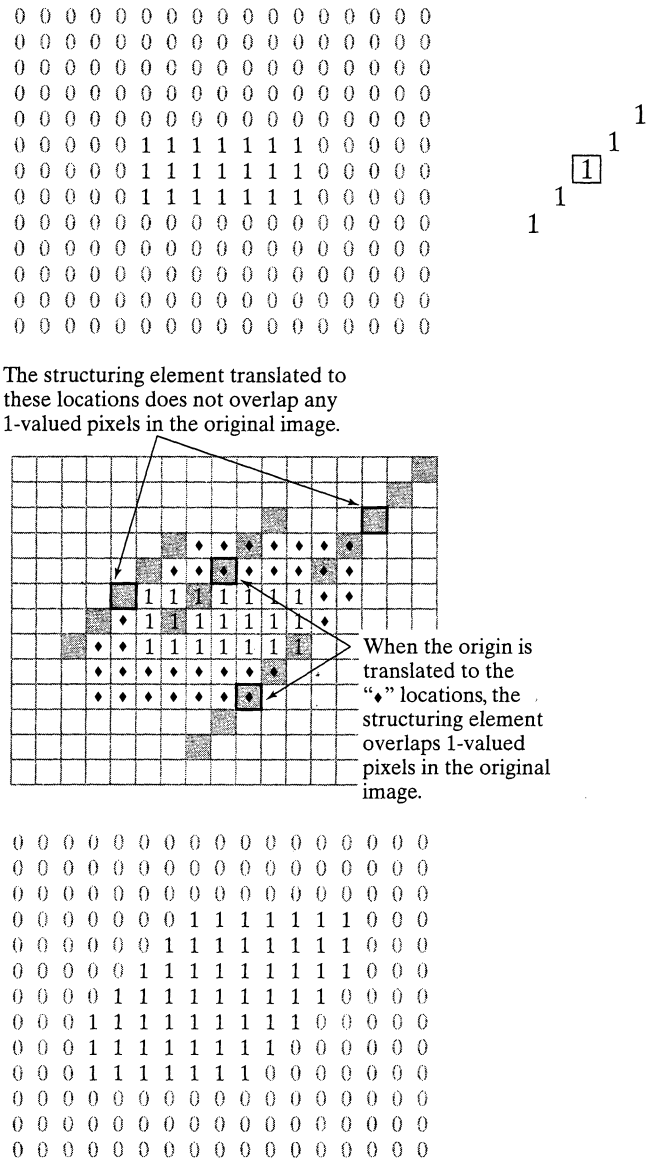
FIGURE 9.3 (a) Binary image A. (b) Binary image B. (c) Complement ~A. (d) Union A | B. (e) Intersection A & B. (f) Set difference A & ~B.

9.2.1 Dilation

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*. 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. Computationally, structuring elements typically are represented by a matrix of 0s and 1s; sometimes it is convenient to show only the 1s, as illustrated in the figure. In addition, the origin of the structuring element must be clearly identified. 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. The output image in 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.

Mathematically, dilation is defined in terms of set operations. The dilation of A by B, denoted $A \oplus B$, is defined as

$$A \oplus B = \{z | (\hat{B})_z \cap A \neq \emptyset\}$$



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.

where $\emptyset$ is the empty set and B is the structuring element. In words, the dilation of A by B is the set consisting of all the structuring element origin locations where the reflected and translated B overlaps at least some portion of A. The translation of the structuring element in dilation is similar to the mechanics of spatial convolution discussed in Chapter 3. Figure 9.4 does not show the structuring element's reflection explicitly because the structuring element is symmetrical with respect to its origin in this case. Figure 9.5 shows a nonsymmetric structuring element and its reflection.

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

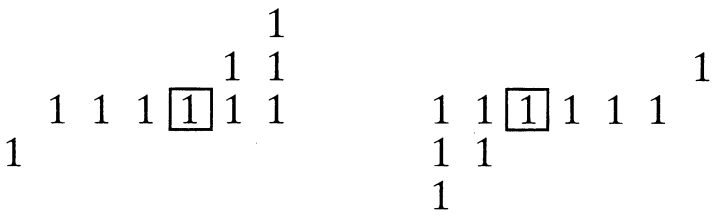


FIGURE 9.5 Structuring element reflection. (a) Nonsymmetric structuring element. (b) Structuring element reflected about its origin.

operand be the structuring element, which usually is much smaller than the image. We follow this convention from this point on.

EXAMPLE 9.1: A simple application of dilation.

■ IPT function `imdilate` performs dilation. Its basic calling syntax is

```
A2 = imdilate(A, B)
```

where `A` and `A2` are binary images, and `B` is a matrix of 0s and 1s that specifies the structuring element. Figure 9.6(a) shows a sample binary image containing text with broken characters. We want to use `imdilate` to dilate the image with the structuring element:

$$\begin{matrix} 0 & 1 & 0 \\ 1 & \boxed{1} & 1 \\ 0 & 1 & 0 \end{matrix}$$

The following commands read the image from a file, form the structuring element matrix, perform the dilation, and display the result.

```
>> A = imread('broken_text.tif');
>> B = [0 1 0; 1 1 1; 0 1 0];
>> A2 = imdilate(A, B);
>> imshow(A2)
```

Figure 9.6(b) shows the resulting image. ■

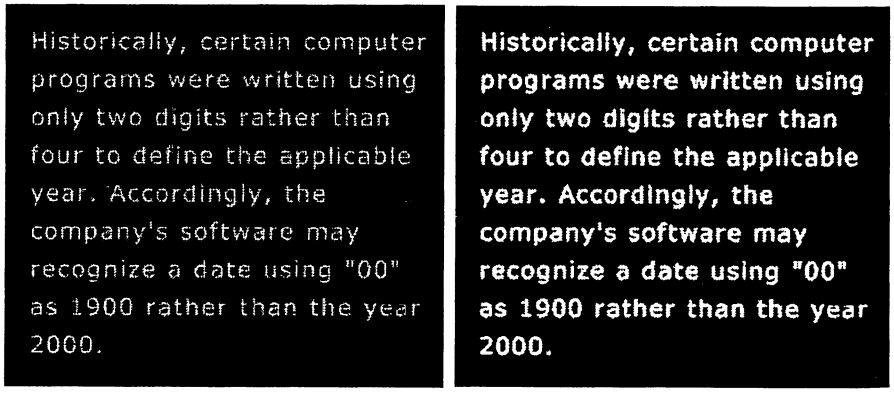


FIGURE 9.6 A simple example of dilation. (a) Input image containing broken text. (b) Dilated image.

9.2.2 Structuring Element Decomposition

Dilation is *associative*. That is,

$$A \oplus (B \oplus C) = (A \oplus B) \oplus C$$

Suppose that a structuring element B can be represented as a dilation of two structuring elements B_1 and B_2:

$$B = B_1 \oplus B_2$$

Then $A \oplus B = A \oplus (B_1 \oplus B_2) = (A \oplus B_1) \oplus B_2$. In other words, dilating A with B is the same as first dilating A with B_1, and then dilating the result with B_2. We say that B can be *decomposed* into the structuring elements B_1 and B_2.

The associative property is important because the time required to compute dilation is proportional to the number of nonzero pixels in the structuring element. Consider, for example, dilation with a 5×5 array of 1s:

$$\begin{array}{ccccc} 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & \boxed{1} & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \end{array}$$

This structuring element can be decomposed into a five-element row of 1s and a five-element column of 1s:

$$[1\ 1\ \boxed{1}\ 1\ 1] \oplus \begin{bmatrix} 1 \\ 1 \\ \boxed{1} \\ 1 \\ 1 \end{bmatrix}$$

The number of elements in the original structuring element is 25, but the total number of elements in the row-column decomposition is only 10. This means that dilation with the row structuring element first, followed by dilation with the column element, can be performed 2.5 times faster than dilation with the 5×5 array of 1s. In practice, the speed-up will be somewhat less because there is usually some overhead associated with each dilation operation, and at least two dilation operations are required when using the decomposed form. However, the gain in speed with the decomposed implementation is still significant.

9.2.3 The `strel` Function

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, such as its size. For example, `strel('diamond', 5)` returns a diamond-shaped structuring element that extends ±5 pixels along the horizontal and vertical axes. Table 9.2 summarizes the various shapes that `strel` can create.

In addition to simplifying the generation of common structuring element shapes, function `strel` also has the important property of producing structuring elements in decomposed form. Function `imdilate` automatically uses the decomposition information to speed up the dilation process. The following example illustrates how `strel` returns information related to the decomposition of a structuring element.

EXAMPLE 9.2: An illustration of structuring element decomposition using `strel`.

■ Consider again the creation of a diamond-shaped structuring element using `strel`:

```
>> se = strel('diamond', 5)
se =
Flat STREL object containing 61 neighbors.
Decomposition: 4 STREL objects containing a total of 17 neighbors
Neighborhood:
     0     0     0     0     0     1     0     0     0     0     0
     0     0     0     0     1     1     1     0     0     0     0
     0     0     0     1     1     1     1     1     0     0     0
     0     0     1     1     1     1     1     1     1     0     0
     0     1     1     1     1     1     1     1     1     1     0
     1     1     1     1     1     1     1     1     1     1     1
     0     1     1     1     1     1     1     1     1     1     0
     0     0     1     1     1     1     1     1     1     0     0
     0     0     0     1     1     1     1     1     0     0     0
     0     0     0     0     1     1     1     0     0     0     0
     0     0     0     0     0     1     0     0     0     0     0
```

We see that `strel` does not display as a normal MATLAB matrix; it returns instead a special quantity called an *strel object*. The command-window display of an strel object includes the neighborhood (a matrix of 1s in a diamond-shaped pattern in this case); the number of 1-valued pixels in the structuring element (61); the number of structuring elements in the decomposition (4); and the total number of 1-valued pixels in the decomposed structuring elements (17). Function `getsequence` can be used to extract and examine separately the individual structuring elements in the decomposition.


```
>> decomp = getsequence(se);
>> whos
  Name          Size                    Bytes  Class

  decomp        4x1                      1716  strel object
  se            1x1                      3309  strel object

Grand total is 495 elements using 5025 bytes
```

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.)

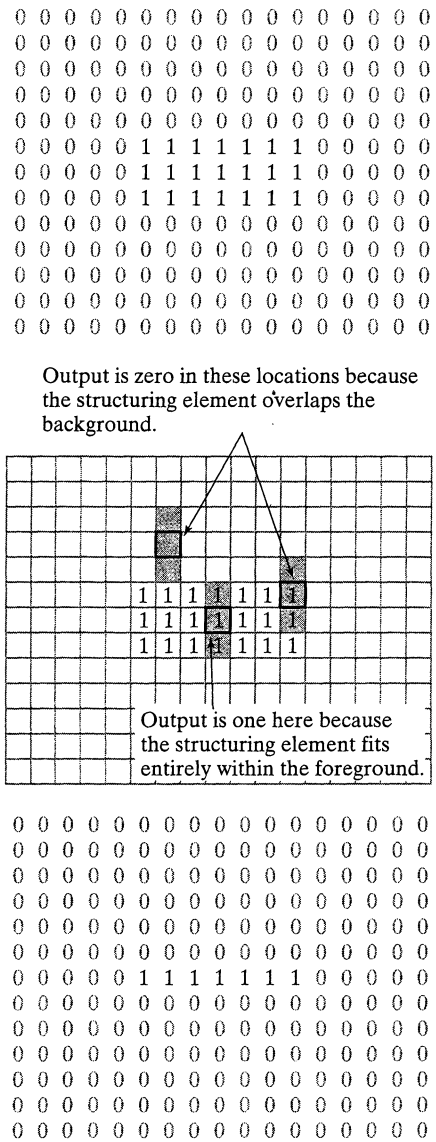
The output of `whos` shows that `se` and `decomp` are both strel objects, and, further, that `decomp` is a four-element vector of strel objects. The four structuring elements in the decomposition can be examined individually by indexing into `decomp`:

```
>> decomp(1)

ans =

Flat STREL object containing 5 neighbors.

Neighborhood:
     0     1     0
     1     1     1
     0     1     0

>> decomp(2)

ans =

Flat STREL object containing 4 neighbors.

Neighborhood:
     0     1     0
     1     0     1
     0     1     0

>> decomp(3)

ans =

Flat STREL object containing 4 neighbors.

Neighborhood:
     0     0     1     0     0
     0     0     0     0     0
     1     0     0     0     1
     0     0     0     0     0
     0     0     1     0     0

>> decomp(4)

ans =

Flat STREL object containing 4 neighbors.

Neighborhood:
     0     1     0
     1     0     1
     0     1     0
```

Function `imdilate` uses the decomposed form of a structuring element automatically, performing dilation approximately three times faster ($\approx 61/17$) than with the non-decomposed form. ■

9.2.4 Erosion

Erosion "shrinks" or "thins" objects in a binary image. As in dilation, the manner and extent of shrinking is controlled by a structuring element. Figure 9.7 illustrates the erosion process. Figure 9.7(a) is the same as Fig. 9.4(a). Figure 9.7(b) is the structuring element, a short vertical line. Figure 9.7(c) graphically depicts erosion as a process of translating the structuring element throughout the domain of the image and checking to see where it fits entirely



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.

within the foreground of the image. The output image in Fig. 9.7(d) has a value of 1 at each location of the origin of the structuring element, such that the element overlaps *only* 1-valued pixels of the input image (i.e., it does not overlap any of the image background).

The mathematical definition of erosion is similar to that of dilation. The erosion of A by B, denoted $A \ominus B$, is defined as

$$A \ominus B = \{z | (B)_z \cap A^c \neq \emptyset\}$$

In other words, erosion of A by B is the set of all structuring element origin locations where the translated B has no overlap with the background of A.

EXAMPLE 9.3: An illustration of erosion.

■ Erosion is performed by IPT function `imerode`. Suppose that we want to remove the thin wires in the image in Fig. 9.8(a), but we want to preserve the other structures. We can do this by choosing a structuring element small enough to fit within the center square and thicker border leads but too large to fit entirely within the wires. Consider the following commands:

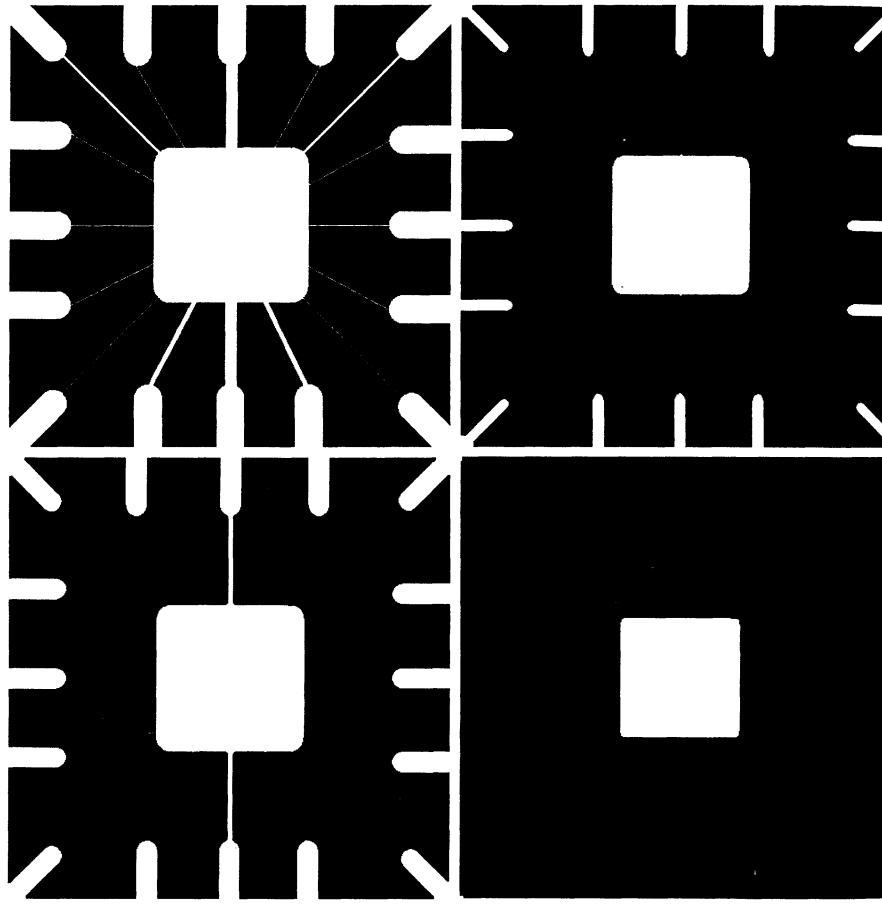


FIGURE 9.8 An illustration of erosion. (a) Original image. (b) Erosion with a disk of radius 10. (c) Erosion with a disk of radius 5. (d) Erosion with a disk of radius 20.

```
>> A = imread('wirebond_mask.tif');
>> se = strel('disk', 10);
>> A2 = imerode(A, se);
>> imshow(A2)
```

As Fig. 9.8(b) shows, these commands successfully removed the thin wires in the mask. Figure 9.8(c) shows what happens if we choose a structuring element that is too small:

```
>> se = strel('disk', 5);
>> A3 = imerode(A, se);
>> imshow(A3)
```

Some of the wire leads were not removed in this case. Figure 9.8(d) shows what happens if we choose a structuring element that is too large:

```
>> A4 = imerode(A, strel('disk', 20));
>> imshow(A4)
```

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

9.3 Combining Dilation and Erosion

In practical image-processing applications, dilation and erosion are used most often in various combinations. An image will undergo a series of dilations and/or erosions using the same, or sometimes different, structuring elements. In this section we consider three of the most common combinations of dilation and erosion: opening, closing, and the hit-or-miss transformation. We also introduce lookup table operations and discuss `bwmorph`, an IPT function that can perform a variety of practical morphological tasks.

9.3.1 Opening and Closing

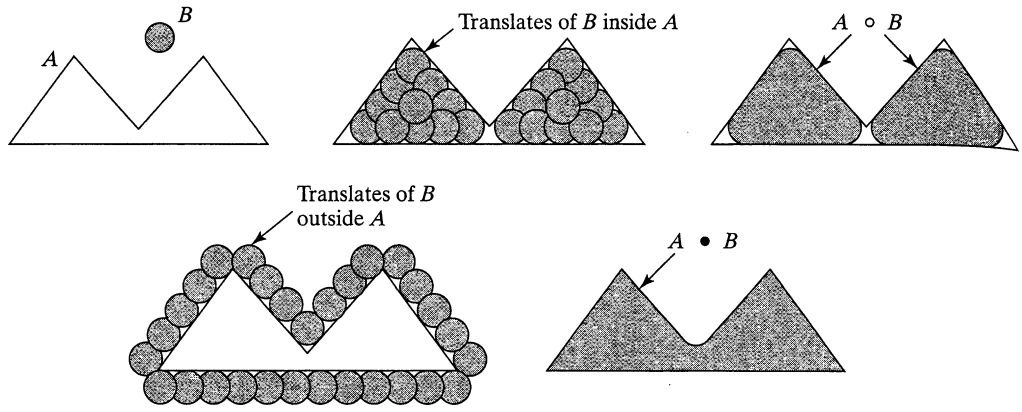
The *morphological opening* of A by B, denoted $A \circ B$, is simply erosion of A by B, followed by dilation of the result by B:

$$A \circ B = (A \ominus B) \oplus B$$

An alternative mathematical formulation of opening is

$$A \circ B = \cup \{(B)_z | (B)_z \subseteq A\}$$

where $\cup \{\bullet\}$ denotes the union of all sets inside the braces, and the notation $C \subseteq D$ means that C is a subset of D. This formulation has a simple geometric interpretation: $A \circ B$ is the union of all translations of B that fit entirely within A. Figure 9.9 illustrates this interpretation. Figure 9.9(a) shows a set A and a disk-shaped structuring element B. Figure 9.9(b) shows some of the translations of B that fit *entirely* within A. The union of all such translations is the shaded region in Fig. 9.9(c); this region is the complete opening. The white regions in this figure are areas where the structuring element could not fit



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).

completely within A, and, therefore, are not part of the opening. Morphological opening removes completely regions of an object that cannot contain the structuring element, smoothes object contours, breaks thin connections, and removes thin protrusions.

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

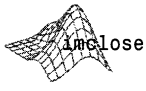
$$A \cdot B = (A \oplus B) \ominus B$$

Geometrically, $A \cdot B$ is the complement of the union of all translations of B that do not overlap A. Figure 9.9(d) illustrates several translations of B that do not overlap A. By taking the complement of the union of all such translations, we obtain the shaded region if Fig. 9.9(e), which is the complete closing. Like opening, morphological closing tends to smooth the contours of objects. Unlike opening, however, it generally joins narrow breaks, fills long thin gulfs, and fills holes smaller than the structuring element.

Opening and closing are implemented in the toolbox with functions `imopen` and `imclose`. These functions have the simple syntax forms

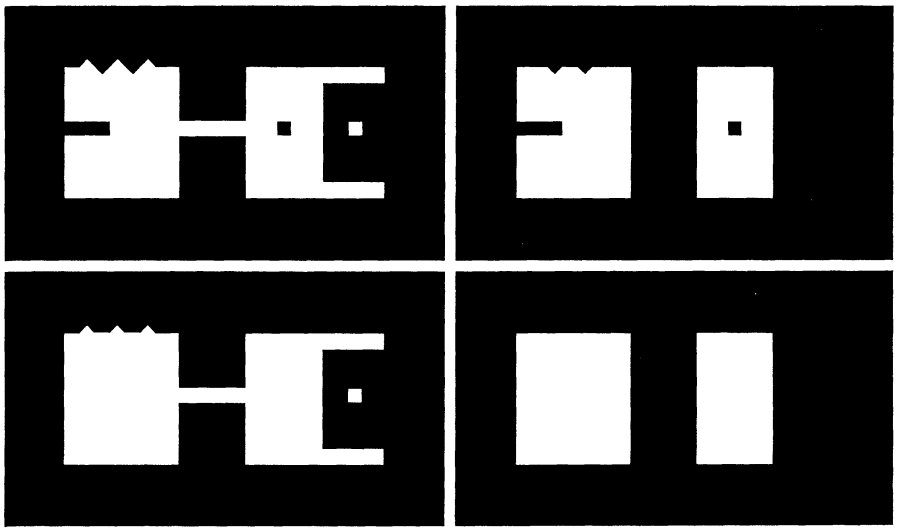

```
C = imopen(A, B)
```

and




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

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



a b
c d

FIGURE 9.10 Illustration of opening and closing. (a) Original image. (b) Opening. (c) Closing. (d) Closing of (b).

EXAMPLE 9.4: Working with functions `imopen` and `imclose`.

■ This example illustrates the use of functions `imopen` and `imclose`. The image `shapes.tif` shown in Fig. 9.10(a) has several features designed to illustrate the characteristic effects of opening and closing, such as thin protrusions, joins, gulfs, an isolated hole, a small isolated object, and a jagged boundary. The following commands open the image with a 20×20 structuring element:

```
>> f = imread('shapes.tif');
>> se = strel('square', 20);
>> fo = imopen(f, se);
>> imshow(fo)
```

Figure 9.10(b) shows the result. Note that the thin protrusions and outward-pointing boundary irregularities were removed. The thin join and the small isolated object were removed also. The commands

```
>> fc = imclose(f, se);
>> imshow(fc)
```

produced the result in Fig. 9.10(c). Here, the thin gulf, the inward-pointing boundary irregularities, and the small hole were removed. As the next paragraph shows, combining a closing and an opening can be quite effective in removing noise. In terms of Fig. 9.10, performing a closing on the result of the earlier opening has the net effect of smoothing the object quite significantly. We close the opened image as follows:

```
>> foc = imclose(fo, se);
>> imshow(foc)
```

Figure 9.10(d) shows the resulting smoothed objects.

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.)

Figure 9.11 further illustrates the usefulness of closing and opening by applying these operations to a noisy fingerprint [Fig. 9.11(a)]. The commands

```
>> f = imread('fingerprint.tif');
>> se = strel('square', 3);
>> fo = imopen(f, se);
>> imshow(fo)
```

produced the image in Fig. 9.11(b). Note that noisy spots were removed by opening the image, but this process introduced numerous gaps in the ridges of the fingerprint. Many of the gaps can be filled in by following the opening with a closing:

```
>> foc = imclose(fo,se);
>> imshow(foc)
```

Figure 9.11(c) shows the final result. ■

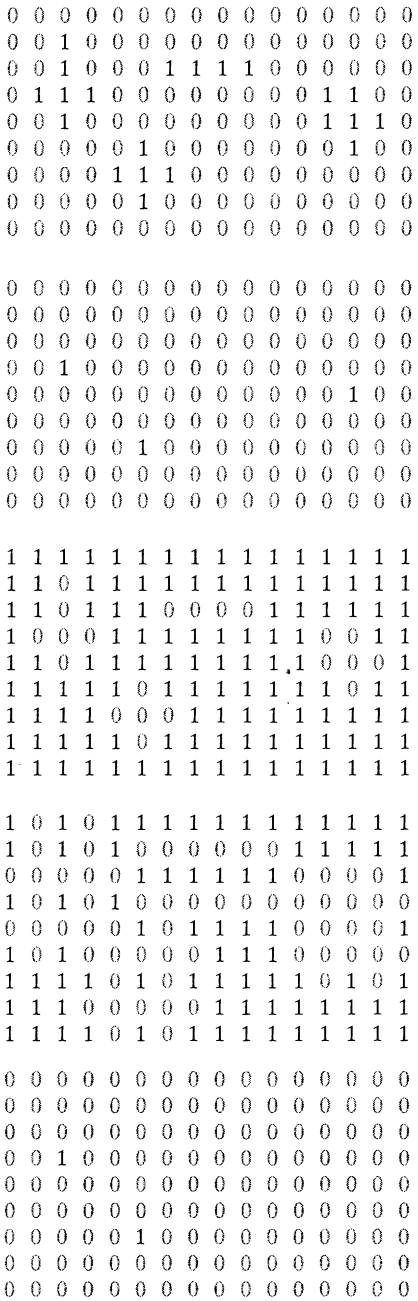
9.3.2 The Hit-or-Miss Transformation

Often, it 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. The *hit-or-miss transformation* is useful for applications such as these. The hit-or-miss transformation of A by B is denoted $A \otimes B$. Here, B is a structuring element pair, $B = (B_1, B_2)$, rather than a single element, as before. The hit-or-miss transformation is defined in terms of these two structuring elements as

$$A \otimes B = (A \ominus B_1) \cap (A^c \ominus B_2)$$

Figure 9.12 shows how the hit-or-miss transformation can be used to identify the locations of the following cross-shaped pixel configuration:

$$\begin{matrix} 0 & 1 & 0 \\ 1 & 1 & 1 \\ 0 & 1 & 0 \end{matrix}$$



B_1

1
1 1 1
1

B_2

1 1
□
1 1

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.

Figure 9.12(a) contains this configuration of pixels in two different locations. Erosion with structuring element B_1 determines the locations of foreground pixels that have north, east, south, and west foreground neighbors. Erosion of the complement with structuring element B_2 determines the locations of all the pixels whose northeast, southeast, southwest, and northwest neighbors

belong to the background. Figure 9.12(g) shows the intersection (logical AND) of these two operations. Each foreground pixel of Fig. 9.12(g) is the location of a set of pixels having the desired configuration.

The name "hit-or-miss transformation" is based on how the result is affected by the two erosions. For example, the output image in Fig. 9.12 consists of all locations that match the pixels in B_1 (a "hit") and that have none of the pixels in B_2 (a "miss"). Strictly speaking, *hit-and-miss transformation* is a more accurate name, but *hit-or-miss transformation* is used more frequently.

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.

EXAMPLE 9.5: Using IPT function `bwhitmiss`.

■ Consider the task of locating upper-left-corner pixels of objects in an image using the hit-or-miss transformation. Figure 9.13(a) shows a simple image containing square shapes. We want to locate foreground pixels that have east and south neighbors (these are "hits") and that have no northeast, north, northwest, west, or southwest neighbors (these are "misses"). These requirements lead to the two structuring elements:

```
>> B1 = strel([0 0 0; 0 1 1; 0 1 0]);
>> B2 = strel([1 1 1; 1 0 0; 1 0 0]);
```

Note that neither structuring element contains the southeast neighbor, which is called a *don't care* pixel. We use function `bwhitmiss` to compute the transformation, where `f` is the input image shown in Fig. 9.13(a):

```
>> g = bwhitmiss(f, B1 ,B2);
>> imshow(g)
```

a b

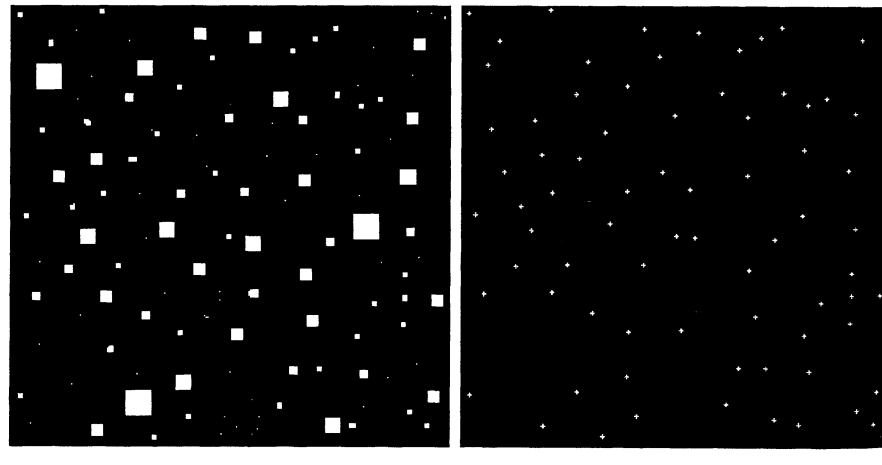


FIGURE 9.13 (a) Original image. (b) Result of applying the hit-or-miss transformation (the dots shown were enlarged to facilitate viewing).

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. ■

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. For instance, there are $2^9 = 512$ different 3×3 configurations of pixel values in a binary image.

To make the use of lookup tables practical, we must assign a unique index to each possible configuration. A simple way to do this for, say, the 3×3 case, is to multiply each 3×3 configuration element-wise by the matrix

$$\begin{matrix} 1 & 8 & 64 \\ 2 & 16 & 128 \\ 4 & 32 & 256 \end{matrix}$$

and then sum all the products. This procedure assigns a unique value in the range [0, 511] to each different 3×3 neighborhood configuration. For example, the value assigned to the neighborhood

$$\begin{matrix} 1 & 1 & 0 \\ 1 & 0 & 1 \\ 1 & 0 & 1 \end{matrix}$$

is $1(1) + 2(1) + 4(1) + 8(1) + 16(0) + 32(0) + 64(0) + 128(1) + 256(1) = 399$, where the first number in these products is a coefficient from the preceding matrix and the numbers in parentheses are the pixel values, taken columnwise.

The toolbox provides two functions, `makelut` and `applylut` (illustrated later in this section), that can be used to implement this technique. Function `makelut` constructs a lookup table based on a user-supplied function, and `applylut` processes binary images using this lookup table. Continuing with the 3×3 case, using `makelut` requires writing a function that accepts a 3×3 binary matrix and returns a single value, typically either a 0 or 1. Function `makelut` calls the user-supplied function 512 times, passing it each possible 3×3 neighborhood. It records and returns all the results in the form of a 512-element vector.

As an illustration, we write a function, `endpoints.m`, that uses `makelut` and `applylut` to detect end points in a binary image. We define an *end point* as a foreground pixel that has exactly one foreground neighbor. Function `endpoints` computes and then applies a lookup table for detecting end points in an input image. The line of code

```
persistent lut
```

used in function `endpoints` establishes a variable called `lut` and declares it to be *persistent*. MATLAB "remembers" the value of persistent variables in between function calls. The first time function `endpoints` is called, variable `lut`

is automatically initialized to the empty matrix ([]). When `lut` is empty, the function calls `makelut`, passing it a handle to subfunction `endpoint_fcn`. Function `applylut` then finds the end points using the lookup table. The lookup table is saved in persistent variable `lut` so that, the next time `endpoints` is called, the lookup table does not need to be recomputed.

`endpoints`

See Section 3.4.2 for a discussion of function handle, @.

```
function g = endpoints(f)
%ENDPOINTS Computes end points of a binary image.
%   G = ENDPOINTS(F) computes the end points of the binary image F
%   and returns them in the binary image G.

persistent lut

if isempty(lut)
   lut = makelut(@endpoint_fcn, 3);
end

g = applylut(f, lut);

%--------------------------------------------------------------------%
function is_end_point = endpoint_fcn(nhood)
%   Determines if a pixel is an end point.
%   IS_END_POINT = ENDPOINT_FCN(NHOOD) accepts a 3-by-3 binary
%   neighborhood, NHOOD, and returns a 1 if the center element is an
%   end point; otherwise it returns a 0.

is_end_point = nhood(2, 2) & (sum(nhood(:)) == 2);
```

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`.

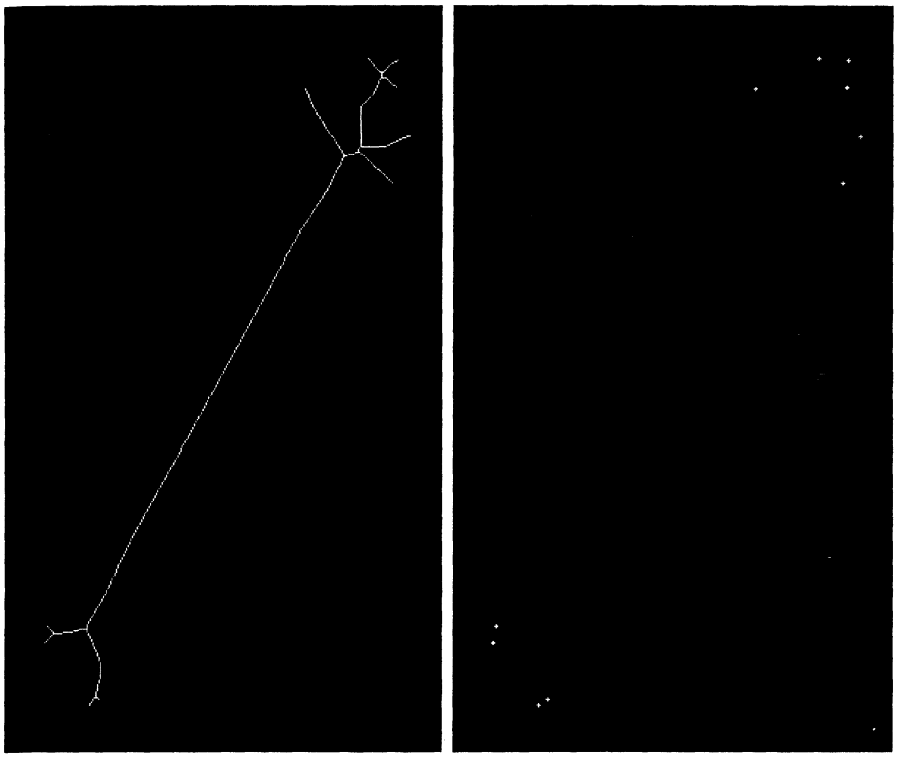
EXAMPLE 9.6: Playing Conway's Game of Life using binary images and lookup-table-based computation.

■ An interesting application of lookup tables is Conway's "Game of Life," which involves "organisms" arranged on a rectangular grid. We include it here as another illustration of the power and simplicity of lookup tables. There are simple rules for how the organisms in Conway's game are born, survive, and die from one "generation" to the next. A binary image is a convenient representation for the game, where each foreground pixel represents a living organism in that location.

Conway's genetic rules describe how to compute the next generation (or next binary image) from the current one:

1. Every foreground pixel with two or three neighboring foreground pixels survives to the next generation.
2. Every foreground pixel with zero, one, or at least four foreground neighbors "dies" (becomes a background pixel) because of "isolation" or "overpopulation."
3. Every background pixel adjacent to exactly three foreground neighbors is a "birth" pixel and becomes a foreground pixel.

All births and deaths occur simultaneously in the process of computing the next binary image depicting the next generation.



a b

FIGURE 9.14 (a) Image of a morphological skeleton. (b) Output of function `endpoints`. The pixels in (b) were enlarged for clarity.

To implement the game of life using `makelut` and `applylut`, we first write a function that applies Conway's genetic laws to a single pixel and its 3×3 neighborhood:

`conwaylaws`

```
function out = conwaylaws(nhood)
%CONWAYLAWS Applies Conway's genetic laws to a single pixel.
%   OUT = CONWAYLAWS(NHOOD) applies Conway's genetic laws to a single
%   pixel and its 3-by-3 neighborhood, NHOOD.
num_neighbors = sum(nhood(:)) - nhood(2, 2);
if nhood(2, 2) == 1
   if num_neighbors <= 1
      out = 0; % Pixel dies from isolation.
   elseif num_neighbors > = 4
      out = 0; % Pixel dies from overpopulation.
   else
      out = 1; % Pixel survives.
    end
else
    if num_neighbors == 3
       out = 1; % Birth pixel.
    else
       out = 0; % Pixel remains empty.
    end
end
```

The lookup table is constructed next by calling `makelut` with a function handle to `conwaylaws`:

```
>> lut = makelut(@conwaylaws, 3);
```

Various starting images have been devised to demonstrate the effect of Conway's laws on successive generations (see Gardner, [1970, 1971]). Consider, for example, an initial image called the "Cheshire cat configuration,"

```
>> bw1 = [0 0 0 0 0 0 0 0 0 0
          0 0 0 0 0 0 0 0 0 0
          0 0 0 1 0 0 1 0 0 0
          0 0 0 1 1 1 1 0 0 0
          0 0 1 0 0 0 0 1 0 0
          0 0 1 0 1 1 0 1 0 0
          0 0 1 0 0 0 0 1 0 0
          0 0 0 1 1 1 1 0 0 0
          0 0 0 0 0 0 0 0 0 0
          0 0 0 0 0 0 0 0 0 0];
```

The following commands perform the computation and display up to the third generation:

```
>> imshow(bw1, 'n'), title('Generation 1')
>> bw2 = applylut(bw1, lut);
>> figure, imshow(bw2, 'n'); title('Generation 2')
>> bw3 = applylut(bw2, lut);
>> figure, imshow(bw3, 'n'); title('Generation 3')
```

We leave it as an exercise to show that after a few generations the cat fades to a "grin" before finally leaving a "paw print." ■

9.3.4 Function bwmorph

IPT function `bwmorph` implements a variety of useful operations based on combinations of dilations, erosions, and lookup table operations. Its calling syntax is


```
g = bwmorph(f, operation, n)
```

where `f` is an input binary image, `operation` is a string specifying the desired operation, and `n` is a positive integer specifying the number of times the operation is to be repeated. Input argument `n` is optional and can be omitted, in which case the operation is performed once. Table 9.3 describes the set of valid operations for `bwmorph`. In the rest of this section we concentrate on two of these: *thinning* and *skeletonization*.

Thinning means reducing binary objects or shapes in an image to strokes that are a single pixel wide. For example, the fingerprint ridges shown in

TABLE 9.3 Operations supported by function `bwmorph`.

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.

Fig. 9.11(c) are fairly thick. It may be desirable for subsequent shape analysis to thin the ridges so that each is one pixel thick. Each application of `bwmorph`'s thinning operation removes one or two pixels from the thickness of binary image objects. The following commands, for example, display the results of applying the thinning operation one and two times.

```
>> f = imread('fingerprint_cleaned.tif');
>> g1 = bwmorph(f, 'thin', 1);
>> g2 = bwmorph(f, 'thin', 2);
>> imshow(g1), figure, imshow(g2)
```

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.

Figures 9.15(a) and 9.15(b), respectively, show the results. A key question is how many times to apply the thinning operation. For several operations, including thinning, `bwmorph` allows `n` to be set to infinity (`Inf`). Calling `bwmorph` with `n = Inf` instructs `bwmorph` to repeat the operation until the image stops changing. Sometimes this is called repeating an operation *until stability*. For example,

```
>> ginf = bwmorph(f, 'thin', Inf);
>> imshow(ginf)
```

As Fig. 9.15(c) shows, this is a significant improvement over Fig. 9.11(c).

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. (Skeletonization is described in more detail in Gonzalez and Woods [2002].) Function `bwmorph` performs skeletonization when `operation` is set to `'skel'`. Let `f` denote the image of the bonelike object in Fig. 9.16(a). To compute its skeleton, we call `bwmorph`, with `n = Inf`:

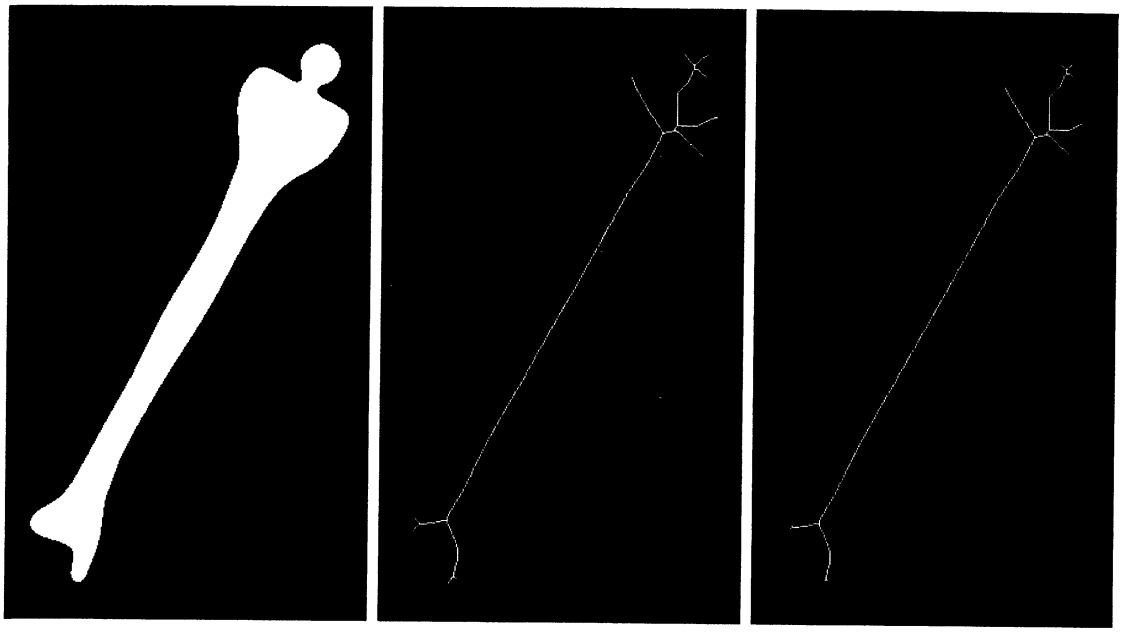
```
>> fs = bwmorph(f, 'skel', Inf);
>> imshow(f), figure, imshow(fs)
```

Figure 9.16(b) shows the resulting skeleton, which is a reasonable likeness of the basic shape of the object.

Skeletonization and thinning often produce short extraneous spurs, sometimes called *parasitic components*. The process of cleaning up (or removing) these spurs is called *pruning*. Function `endpoints` (Section 9.3.3) can be used for this purpose. The method is to iteratively identify and remove endpoints. The following simple commands, for example, postprocesses the skeleton image `fs` through five iterations of endpoint removals:

```
>> for k = 1:5
    fs = fs & ~endpoints(fs);
end
```

Figure 9.16(c) shows the result.



a b c

FIGURE 9.16 (a) Bone image. (b) Skeleton obtained using function `bwmorph`. (c) Resulting skeleton after pruning with function `endpoints`.

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. In this section we consider the important "middle ground" between individual foreground pixels and the set of all foreground pixels. This leads to the notion of *connected components*, also referred to as *objects* in the following discussion.

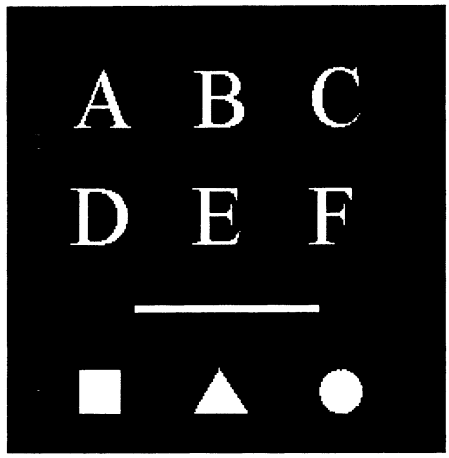
When asked to count the objects in Fig. 9.17(a), most people would identify ten: six characters and four simple geometric shapes. Figure 9.17(b) shows a small rectangular section of pixels in the image. How are the sixteen foreground pixels in Fig. 9.17(b) related to the ten objects in the image? Although they appear to be in two separate groups, all sixteen pixels actually belong to the letter "E" in Fig. 9.17(a). To develop computer programs that locate and operate on objects, we need a more precise set of definitions for key terms.

A pixel p at coordinates (x, y) has two horizontal and two vertical neighbors whose coordinates are $(x + 1, y)$, $(x - 1, y)$, $(x, y + 1)$ and $(x, y - 1)$. This set of 4-*neighbors* of p, denoted $N_4(p)$, 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 $N_D(p)$. The union of $N_4(p)$ and $N_D(p)$ in Fig. 9.18(c) are the 8-*neighbors* of p, denoted $N_8(p)$.

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

a b

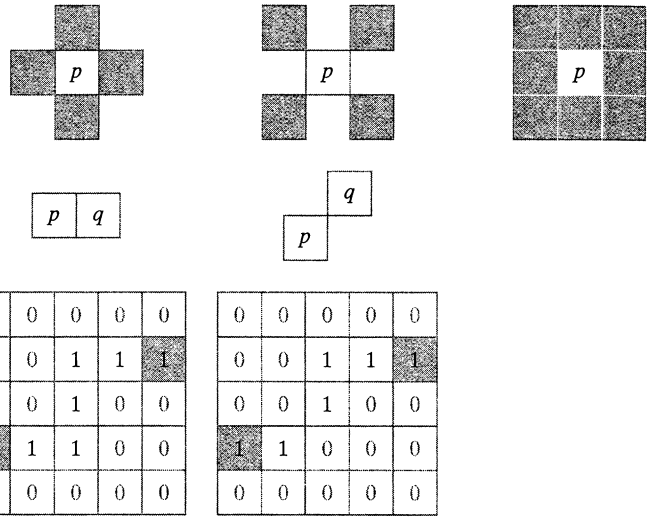
FIGURE 9.17
(a) Image containing ten objects. (b) A subset of pixels from the image.



0	1	1	1	0	0	0	0	0	0
0	0	1	1	0	0	0	0	0	0
0	0	0	1	0	0	0	0	0	0
0	0	0	1	0	0	0	0	0	0
0	0	0	1	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	1	0
0	0	0	0	0	0	0	1	0	0
0	0	0	0	0	0	1	1	0	0
0	0	0	0	0	0	1	1	0	0
0	0	0	0	0	1	1	0	0	0

a b c
d e
f g

FIGURE 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.



0	0	0	0	0
0	0	1	1	1
0	0	1	0	0
1	1	1	0	0
0	0	0	0	0

0	0	0	0	0
0	0	1	1	1
0	0	1	0	0
1	1	0	0	0
0	0	0	0	0

these concepts. A *path* between pixels p_1 and p_n is a sequence of pixels $p_1, p_2, \ldots, p_{n-1}, p_n$ such that p_k is adjacent to p_{k+1}, for $1 \leq k < n$. A path can be 4-*connected* or 8-*connected*, depending on the definition of adjacency used.

Two foreground pixels p and q are said to be 4-connected if there exists a 4-connected path between them, consisting entirely of foreground pixels [Fig. 9.18(f)]. They are 8-connected if there exists an 8-connected path between them [Fig. 9.18(g)]. For any foreground pixel, p, the set of all foreground pixels connected to it is called the *connected component* containing p.

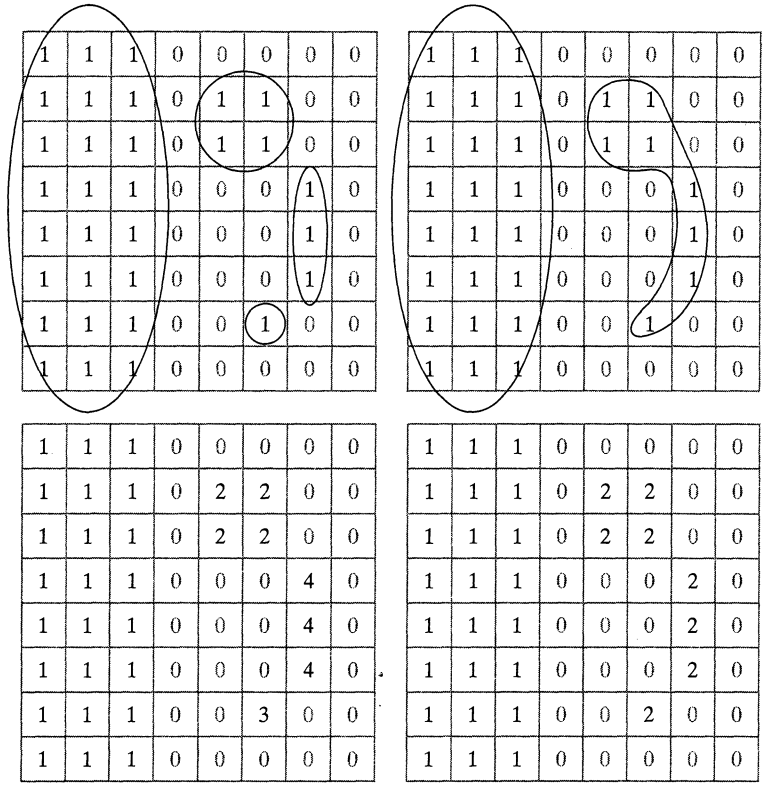


FIGURE 9.19 Connected components (a) Four 4-connected components. (b) Two 8-connected components. (c) Label matrix obtained using 4-connectivity (d) Label matrix obtained using 8-connectivity.

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 illustrates the effect that adjacency can have on determining the number of connected components in an image. 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.

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). Output `L` is called a *label matrix*, and `num` (optional) gives the total number of connected components found. If parameter `conn` is omitted, its value defaults to 8. Figure 9.19(c) shows the label matrix corresponding to Fig. 9.19(a), computed using `bwlabel(f, 4)`. The pixels in each different connected component are assigned a unique integer, from 1 to the total number of connected components. In other words, the pixels labeled 1 belong to the

first connected component; the pixels labeled 2 belong to the second connected component; and so on. Background pixels are labeled 0. Figure 9.19(d) shows the label matrix corresponding to Fig. 9.19(a), computed using `bwlabel(f, 8)`.

EXAMPLE 9.7: Computing and displaying the center of mass of connected components.

■ This example shows how to compute and display the center of mass of each connected component in Fig. 9.17(a). First, we use `bwlabel` to compute the 8-connected components:

```
>> f = imread('objects.tif');
>> [L, n] = bwlabel(f);
```

Function `find` (Section 5.2.2) is useful when working with label matrices. For example, the following call to `find` returns the row and column indices for all the pixels belonging to the third object:

```
>> [r, c] = find(L == 3);
```

Function `mean` with `r` and `c` as inputs then computes the center of mass of this object.

If `A` *is a vector,* `mean(A)` *computes the average value of its elements. If* `A` *is a matrix,* `mean(A)` *treats the columns of* `A` *as vectors, returning a row vector of mean values. The syntax* `mean(A, dim)` *returns the mean values of the elements along the dimension of* `A` *specified by scalar* `dim`.

```
>> rbar = mean(r);
>> cbar = mean(c);
```

A loop can be used to compute and display the centers of mass of all the objects in the image. To make the centers of mass visible when superimposed on the image, we display them using a white " * " marker on top of a black-filled circle marker, as follows:

```
>> imshow(f)
>> hold on   % So later plotting commands plot on top of the image.
>> for k = 1:n
   [r, c] = find(L == k);
   rbar = mean(r);
   cbar = mean(c);
   plot(cbar, rbar, 'Marker', 'o', 'MarkerEdgeColor', 'k',...
        'MarkerFaceColor', 'k', 'MarkerSize', 10)
   plot(cbar, rbar, 'Marker', '*', 'MarkerEdgeColor', 'w')
end
```

Figure 9.20 shows the result. ■

9.5 Morphological Reconstruction

Reconstruction is a morphological transformation involving two images and a structuring element (instead of a single image and structuring element). One image, the *marker*, is the starting point for the transformation. The other image, the *mask*, constrains the transformation. The structuring element used

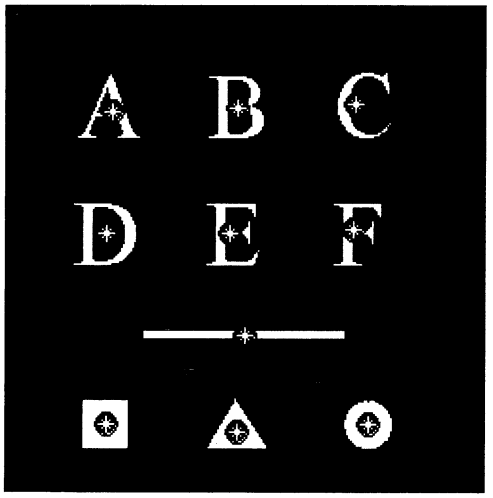


FIGURE 9.20 Centers of mass (white asterisks) shown superimposed on their corresponding connected components.

defines connectivity. In this section we use 8-connectivity (the default), which implies that B in the following discussion is a 3×3 matrix of 1s, with the center defined at coordinates $(2, 2)$.

If g is the mask and f is the marker, the reconstruction of g from f, denoted $R_g(f)$, is defined by the following iterative procedure:

1. Initialize h_1 to be the marker image f.
2. Create the structuring element: `B = ones(3)`.
3. Repeat:

$$h_{k+1} = (h_k \oplus B) \cap g$$

until $h_{k+1} = h_k$.

Marker f must be a subset of g; that is,

$$f \subseteq g$$

Figure 9.21 illustrates the preceding iterative procedure. Note that, although this iterative formulation is useful conceptually, much faster computational algorithms exist. IPT function `imreconstruct` uses the "fast hybrid reconstruction" algorithm described in Vincent [1993]. The calling syntax for `imreconstruct` is

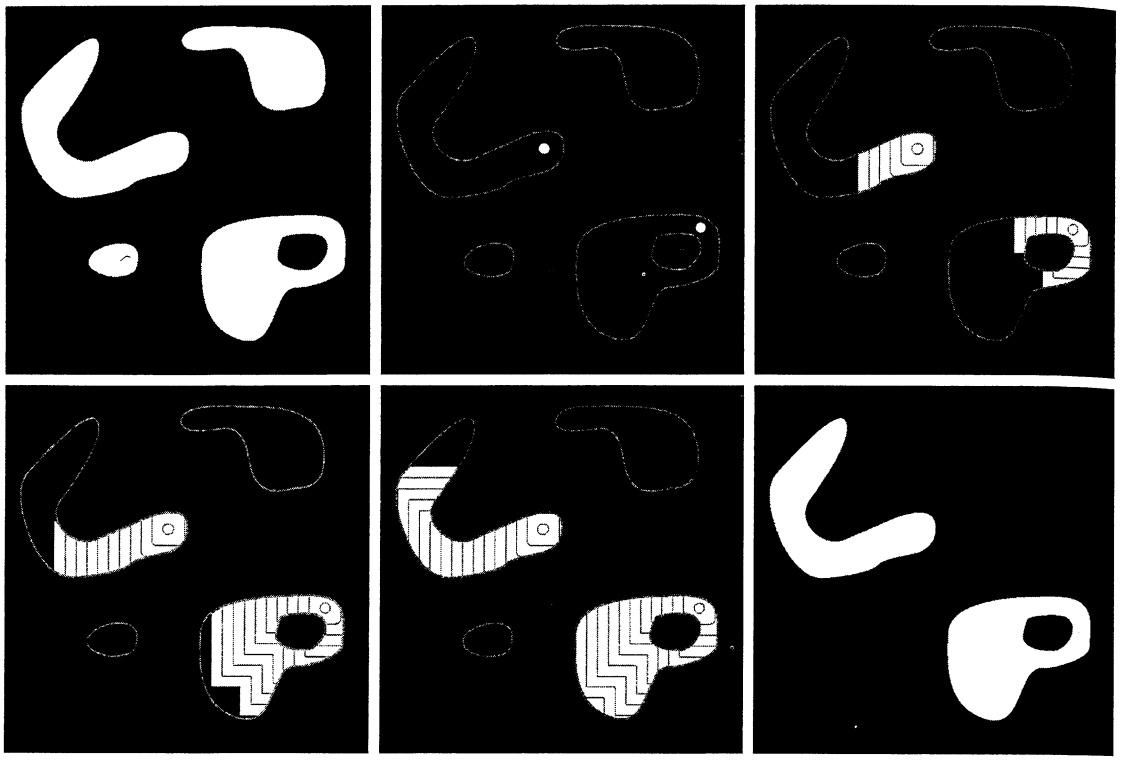
```
out = imreconstruct(marker, mask)
```

where `marker` and `mask` are as defined at the beginning of this section.

9.5.1 Opening by Reconstruction

In morphological opening, erosion typically removes small objects, and the subsequent dilation tends to restore the shape of the objects that remain. However, the accuracy of this restoration depends on the similarity between

See Sections 10.4.2 and 10.4.3 for additional applications of morphological reconstruction.



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.]

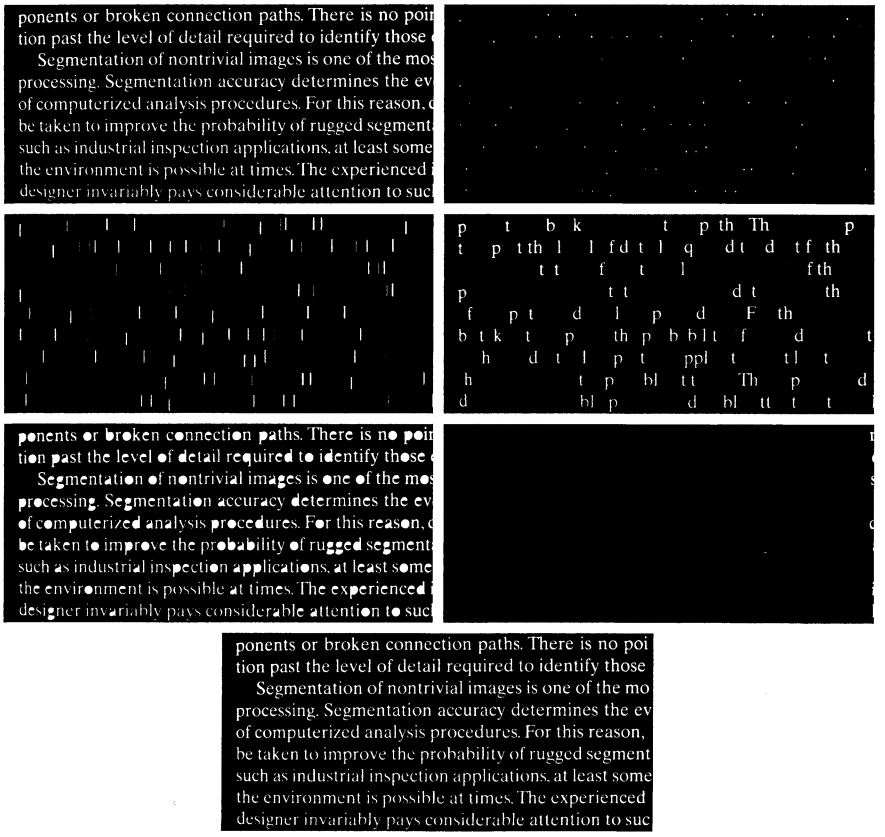
the shapes and the structuring element. The method discussed in this section, *opening by reconstruction*, restores exactly the shapes of the objects that remain after erosion. The opening by reconstruction of f, using structuring element B, is defined as $R_f(f \ominus B)$.

EXAMPLE 9.8: Opening by reconstruction.

■ A comparison between opening and opening by reconstruction for an image containing text is shown in Fig. 9.22. In this example, we are interested in extracting from Fig. 9.22(a) the characters that contain long vertical strokes. Since opening by reconstruction requires an eroded image, we perform that step first, using a thin, vertical structuring element of length proportional to the height of the characters:

```
>> f = imread('book_text_bw.tif');
>> fe = imerode(f, ones(51, 1));
```

Figure 9.22(b) shows the result. The opening, shown in Fig. 9.22(c), is computed using `imopen`:



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.

```
>> fo = imopen(f, ones(51, 1));
```

Note that the vertical strokes were restored, but not the rest of the characters containing the strokes. Finally, we obtain the reconstruction:

```
>> fobr = imreconstruct(fe, f);
```

The result in Fig. 9.22(d) shows that characters containing long vertical strokes were restored exactly; all other characters were removed. The remaining parts of Fig. 9.22 are explained in the following two sections. ■

9.5.2 Filling Holes

Morphological reconstruction has a broad spectrum of practical applications, each determined by the selection of the marker and mask images. For example, suppose that we choose the marker image, f_m, to be 0 everywhere except on the image border, where it is set to $1 - f$:

$$f_m(x, y) = \begin{cases} 1 - f(x, y) & \text{if } (x, y) \text{ is on the border of } f \\ 0 & \text{otherwise} \end{cases}$$

Then $g = [R_f{}^c(f_m)]^c$ has the effect of filling the holes in f, as illustrated in Fig. 9.22(e). IPT function `imfill` performs this computation automatically when the optional argument `'holes'` is used:


```
g = imfill(f, 'holes')
```

This function is discussed in more detail in Section 11.1.2.

9.5.3 Clearing Border Objects

Another useful application of reconstruction is removing objects that touch the border of an image. Again, the key task is to select the appropriate marker and mask images to achieve the desired effect. In this case, we use the original image as the mask, and the marker image, f_m, is defined as

$$f_m(x, y) = \begin{cases} f(x, y) & \text{if } (x, y) \text{ is on the border of } f \\ 0 & \text{otherwise} \end{cases}$$

Figure 9.22(f) shows that the reconstruction, $R_f(f_m)$, contains only the objects touching the border. The set difference $f - R_f(f_m)$, shown in Fig. 9.22(g), contains only the objects from the original image that do not touch the border. IPT function `imclearborder` performs this entire procedure automatically. Its syntax is


```
g = imclearborder(f, conn)
```

where `f` is the input image and `g` is the result. The value of `conn` can be either 4 or 8 (the default). This function suppresses structures that are lighter than their surroundings and that are connected to the image border. Input `f` can be a gray-scale or binary image. The output image is a gray-scale or binary image, respectively.

9.6 Gray-Scale Morphology

All the binary morphological operations discussed in this chapter, with the exception of the hit-or-miss transform, have natural extensions to gray-scale images. In this section, as in the binary case, we start with dilation and erosion, which for gray-scale images are defined in terms of minima and maxima of pixel neighborhoods.

9.6.1 Dilation and Erosion

The *gray-scale dilation* of f by structuring element b, denoted $f \oplus b$, is defined as

$$(f \oplus b)(x, y) = \max\{f(x - x', y - y') + b(x', y') \mid (x', y') \in D_b\}$$

where D_b is the domain of b, and $f(x, y)$ is assumed to equal $-\infty$ outside the domain of f. This equation implements a process similar to the concept of spatial convolution, explained in Section 3.4.1. Conceptually, we can think of

rotating the structuring element about its origin and translating it to all locations in the image, just as the convolution kernel is rotated and then translated about the image. At each translated location, the rotated structuring element values are added to the image pixel values and the maximum is computed.

One important difference between convolution and gray-scale dilation is that, in the latter, D_b, a binary matrix, defines which locations in the neighborhood are included in the max operation. In other words, for an arbitrary pair of coordinates (x_0, y_0) in the domain of D_b, the sum $f(x - x_0, y - y_0) + b(x_0, y_0)$ is included in the max computation only if D_b is 1 at those coordinates. If D_b is 0 at (x_0, y_0), the sum is not considered in the max operation. This is repeated for all coordinates $(x', y') \in D_b$ each time that coordinates (x, y) change. Plotting $b(x', y')$ as a function of coordinates x' and y' would look like a digital "surface" with the height at any pair of coordinates being given by the value of b at those coordinates.

In practice, gray-scale dilation usually is performed using *flat* structuring elements (see Table 9.2) in which the value (height) of b is 0 at all coordinates over which D_b is defined. That is,

$$b(x', y') = 0 \quad \text{for } (x', y') \in D_b$$

In this case, the max operation is specified completely by the pattern of 0s and 1s in binary matrix D_b, and the gray-scale dilation equation simplifies to

$$(f \oplus b)(x, y) = \max\{f(x - x', y - y') \mid (x', y') \in D_b\}$$

Thus, flat gray-scale dilation is a local-maximum operator, where the maximum is taken over a set of pixel neighbors determined by the shape of D_b.

Nonflat structuring elements are created with `strel` by passing it two matrices: (1) a matrix of 0s and 1s specifying the structuring element domain, D_b, and (2) a second matrix specifying height values, $b(x', y')$. For example,

```
>> b = strel([1 1 1], [1 2 1])
    b =
    Nonflat STREL object containing 3 neighbors.
    Neighborhood:
      1     1     1
    Height:
      1     2     1
```

creates a 1×3 structuring element whose height values are $b(0, -1) = 1$, $b(0, 0) = 2$, and $b(0, 1) = 1$.

Flat structuring elements for gray-scale images are created using `strel` in the same way as for binary images. For example, the following commands show how to dilate the image `f` in Fig. 9.23(a) using a flat 3×3 structuring element:

```
>> se = strel('square', 3);
>> gd = imdilate(f, se);
```

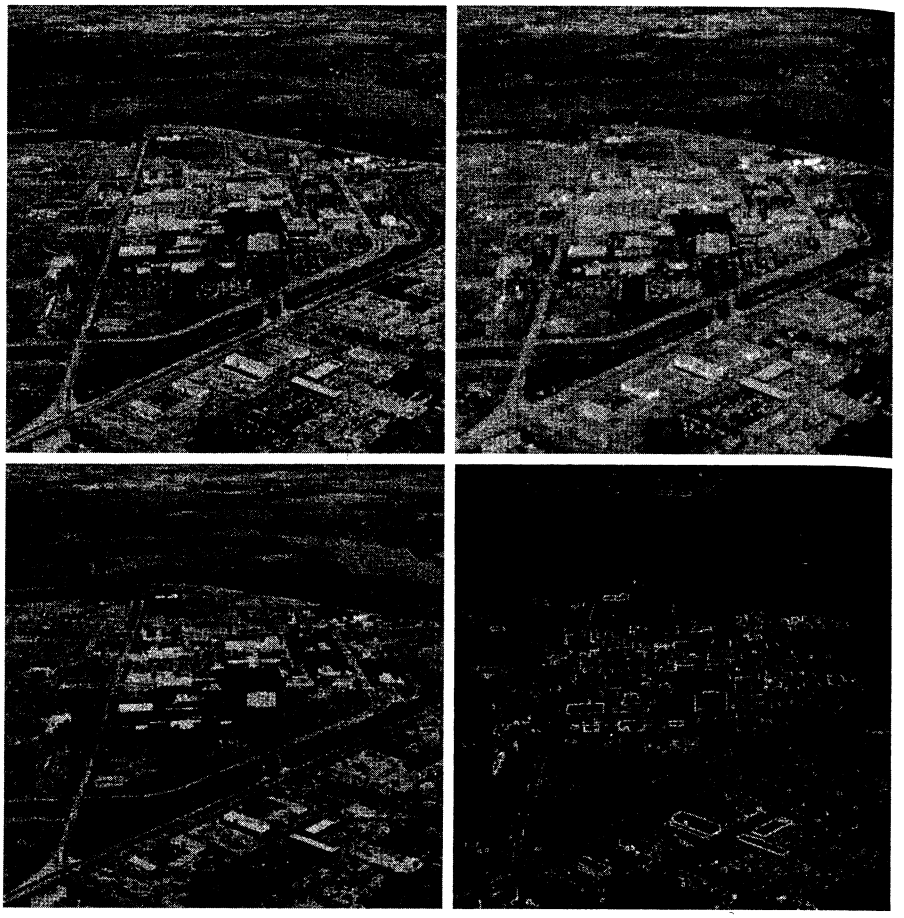


FIGURE 9.23 Dilation and erosion. (a) Original image. (b) Dilated image. (c) Eroded image. (d) Morphological gradient. (Original image courtesy of NASA.)

Figure 9.23(b) shows the result. As expected, the image is slightly blurred. The rest of this figure is explained in the following discussion.

The *gray-scale erosion* of f by structuring element b, denoted $f \ominus b$, is defined as

$$(f \ominus b)(x, y) = \min\{f(x + x', y + y') - b(x', y') \mid (x', y') \in D_b\}$$

where D_b is the domain of b and $f(x, y)$ is assumed to be $+\infty$ outside the domain of f. Conceptually, we again can think of translating the structuring element to all locations in the image. At each translated location, the structuring element values are subtracted from the image pixel values and the minimum is taken.

As with dilation, gray-scale erosion is most often performed using flat structuring elements. The equation for flat gray-scale erosion can then be simplified to

$$(f \ominus b)(x, y) = \min\{f(x + x', y + y') \mid (x', y') \in D_b\}$$

Thus, flat gray-scale erosion is a local-minimum operator, in which the minimum is taken over a set of pixel neighbors determined by the shape of D_b.

Figure 9.23(c) shows the result of using `imerode` with the same structuring element used for Fig. 9.23(b):

```
>> ge = imerode(f, se);
```

Dilation and erosion can be combined to achieve a variety of effects. For instance, subtracting an eroded image from its dilated version produces a "morphological gradient," which is a measure of local gray-level variation in the image. For example, letting

Computing the morphological gradient requires a different procedure for nonsymmetric structuring elements. Specifically, a reflected structuring element must be used in the dilation step.

```
>> morph_grad = imsubtract(gd, ge);
```

produced the image in Fig. 9.23(d), which is the morphological gradient of the image in Fig. 9.23(a). This image has edge-enhancement characteristics similar to those that would be obtained using the gradient operations discussed in Sections 6.6.1 and later in Section 10.1.3.

9.6.2 Opening and Closing

The expressions for opening and closing gray-scale images have the same form as their binary counterparts. The opening of image f by structuring element b, denoted $f \circ b$, is defined as

$$f \circ b = (f \ominus b) \oplus b$$

As before, this is simply the erosion of f by b, followed by the dilation of the result by b. Similarly, the closing of f by b, denoted $f \bullet b$, is dilation followed by erosion:

$$f \bullet b = (f \oplus b) \ominus b$$

Both operations have simple geometric interpretations. Suppose that an image function $f(x, y)$ is viewed as a 3-D surface; that is, its intensity values are interpreted as height values over the xy-plane. Then the opening of f by b can be interpreted geometrically as pushing structuring element b up against the underside of the surface and translating it across the entire domain of f. The opening is constructed by finding the highest points reached by any part of the structuring element as it slides against the undersurface of f.

Figure 9.24 illustrates the concept in one dimension. Consider the curve in Fig. 9.24(a) to be the values along a single row of an image. Figure 9.24(b) shows a flat structuring element in several positions, pushed up against the bottom of the curve. The complete opening is shown as the curve along the top of the shaded region in Fig. 9.24(c). Since the structuring element is too large to fit inside the upward peak on the middle of the curve, that peak is removed by the opening. In general, openings are used to remove small bright details while leaving the overall gray levels and larger bright features relatively undisturbed.

Figure 9.24(d) provides a graphical illustration of closing. Note that the structuring element is pushed down on top of the curve while being translated

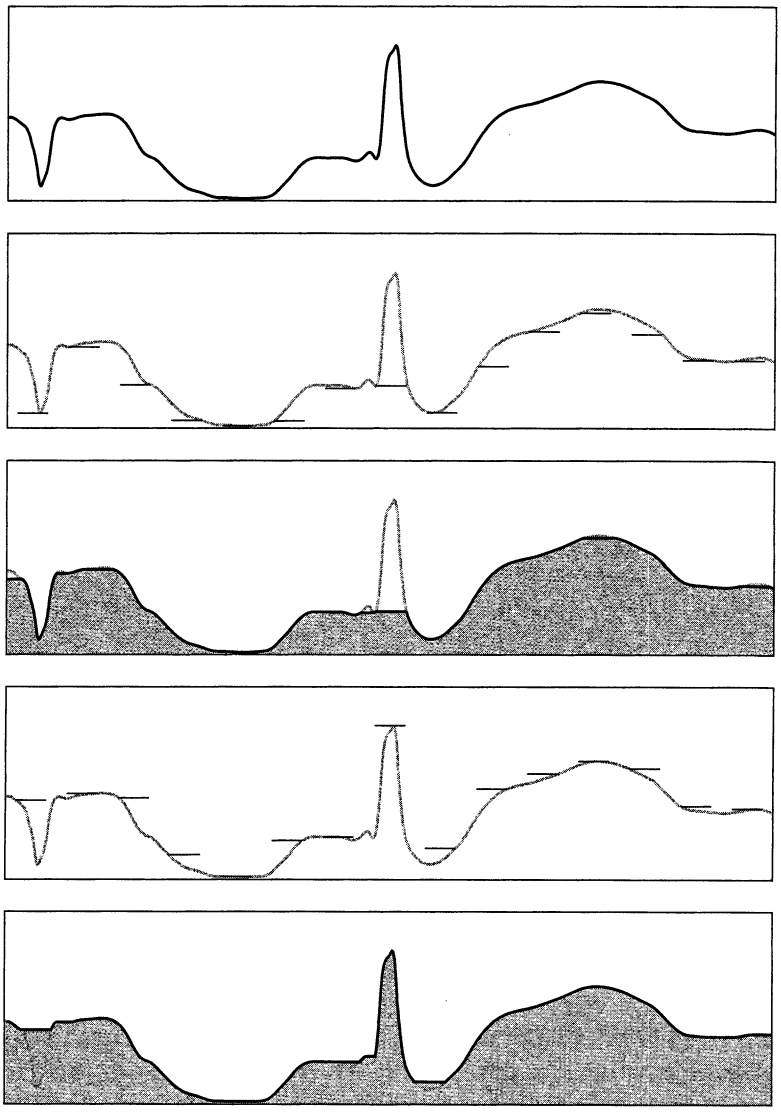
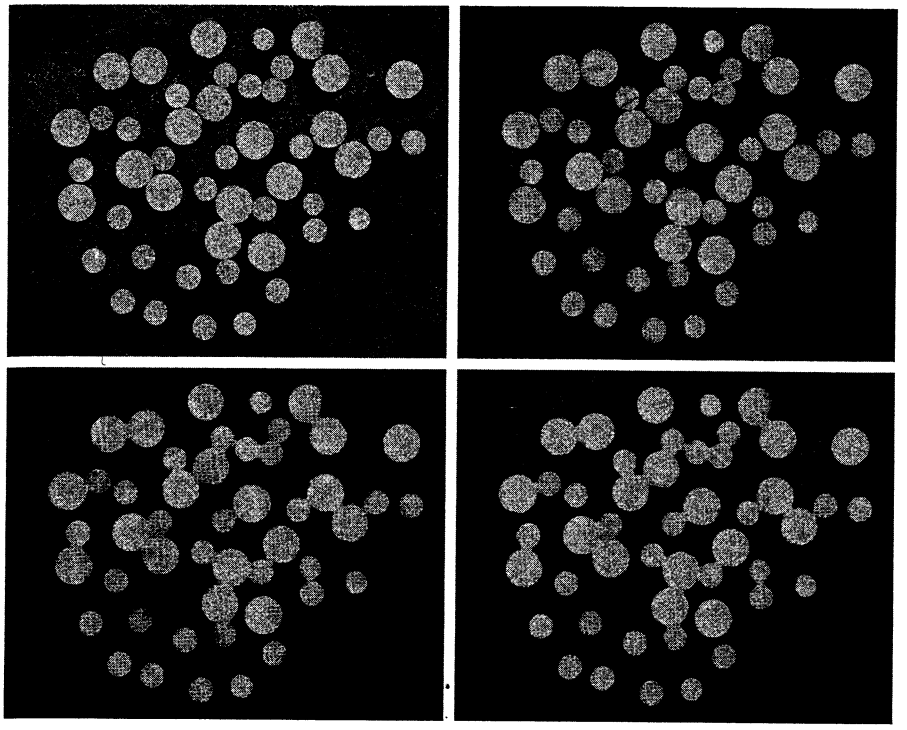


FIGURE 9.24 Opening and closing in one dimension. (a) Original 1-D signal. (b) Flat structuring element pushed up underneath the signal. (c) Opening. (d) Flat structuring element pushed down along the top of the signal. (e) Closing.

to all locations. The closing, shown in Fig. 9.24(e), is constructed by finding the lowest points reached by any part of the structuring element as it slides against the upper side of the curve. Here, we see that closing suppresses dark details smaller than the structuring element.

EXAMPLE 9.9: Morphological smoothing using openings and closings.

■ Because opening suppresses bright details smaller than the structuring element, and closing suppresses dark details smaller than the structuring element, they are used often in combination for image smoothing and noise removal. In this example we use `imopen` and `imclose` to smooth the image of wood dowel plugs shown in Fig. 9.25(a):



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.

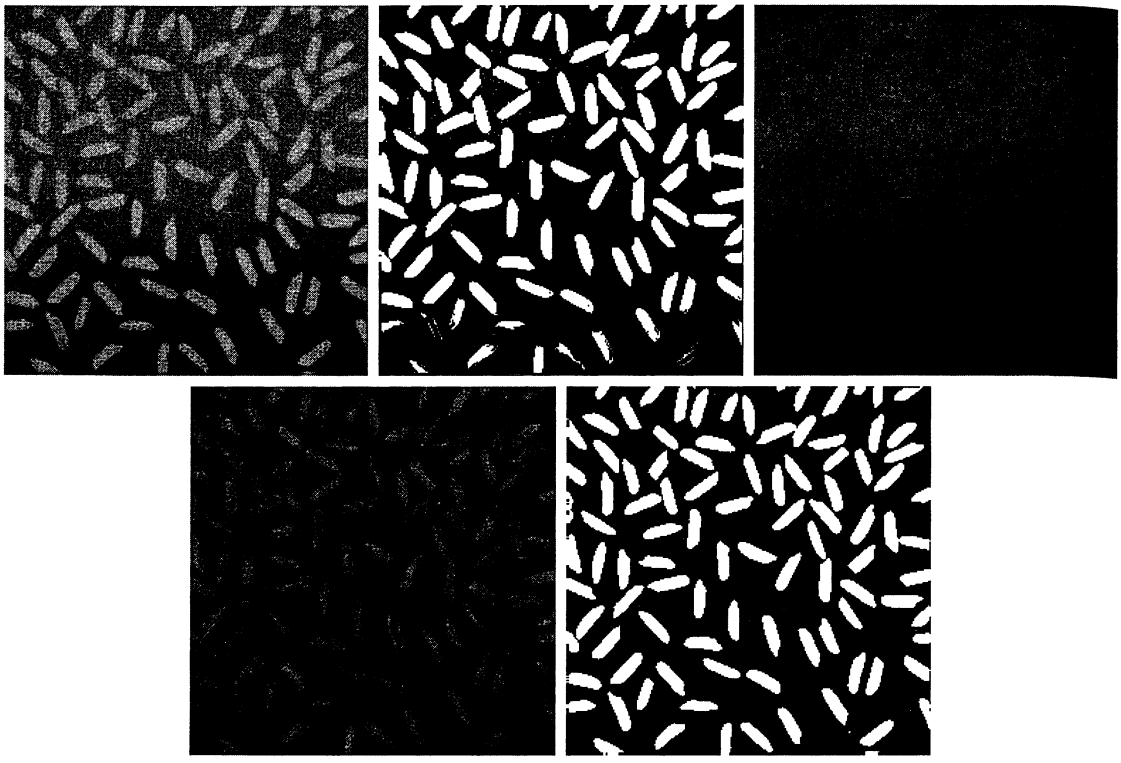
```
>> f = imread('plugs.jpg');
>> se = strel('disk', 5);
>> fo = imopen(f, se);
>> foc = imclose(fo, se);
```

Figure 9.25(b) shows the opened image, `fo`, and Fig. 9.25(c) shows the closing of the opening, `foc`. Note the smoothing of the background and of the details in the objects. This procedure is often called *open-close filtering*. *Close-open filtering* produces similar results.

Another way to use openings and closings in combination is in *alternating sequential filtering*. One form of alternating sequential filtering is to perform open-close filtering with a series of structuring elements of increasing size. The following commands illustrate this process, which begins with a small structuring element and increases its size until it is the same as the structuring element used to obtain Figs. 9.25(b) and (c):

```
>> fasf = f;
>> for k = 2:5
      se = strel('disk', k);
      fasf = imclose(imopen(fasf, se), se);
   end
```

The result, shown in Fig. 9.25(d), yielded slightly smoother results than using a single open-close filter, at the expense of additional processing. ■



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.)

EXAMPLE 9.10: Using the tophat transformation.

■ 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. The uneven illumination makes image thresholding (Section 10.3) difficult. 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. Opening the image can produce a reasonable estimate of the background across the image, as long as the structuring element is large enough so that it does not fit entirely within the rice grains. For example, the commands

```
>> 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);
```

Figure 9.26(d) shows the result, and Fig. 9.26(e) shows the new thresholded image. The improvement is apparent.

Subtracting an opened image from the original is called a *top-hat* transformation. IPT function `imtophat` performs this operation in a single step:

```
>> f2 = imtophat(f, se);
```

Function `imtophat` can also be called as `g = imtophat(f, NHOOD)`, where `NHOOD` is an array of 0s and 1s that specifies the size and shape of the structuring element. This syntax is the same as using the call `imtophat(f, strel(NHOOD))`.

A related function, `imbothat`, performs a *bottom-hat* transformation, defined as the closing of the image minus the image. Its syntax is the same as for function `imtophat`. These two functions can be used together for contrast enhancement using commands such as


```
>> se = strel('disk', 3);
>> g = imsubtract(imadd(f, imtophat(f, se)), imbothat(f , se));
```

■

EXAMPLE 9.11: Granulometry.

■ Techniques for determining the size distribution of particles in an image are an important part of the field of *granulometry*. Morphological techniques can be used to measure particle size distribution indirectly; that is, without identifying explicitly and measuring every particle. For particles with regular shapes that are lighter than the background, the basic approach is to apply morphological openings of increasing size. For each opening, the sum of all the pixel values in the opening is computed; this sum sometimes is called the *surface area* of the image. The following commands apply disk-shaped openings with radii 0 to 35 to the image in Fig. 9.25(a):

```
>> f = imread('plugs.jpg');
>> sumpixels = zeros(1, 36);
>> for k = 0:35
      se = strel('disk', k);
      fo = imopen(f, se);
      sumpixels(k + 1) = sum(fo(:));
   end
>> plot(0:35, sumpixels), xlabel('k'), ylabel('Surface area')
```

If `v` *is a vector, then* `diff(v)` *returns a vector, one element shorter than* `v`*, of differences between adjacent elements. If* `X` *is a matrix, then* `diff(X)` *returns a matrix of row differences:* `X(2:end, :) - X(1:end-1, :)`.

Figure 9.27(a) shows the resulting plot of `sumpixels` versus `k`. More interesting is the reduction in surface area between successive openings:


```
>> plot(-diff(sumpixels))
>> xlabel('k')
>> ylabel('Surface area reduction')
```

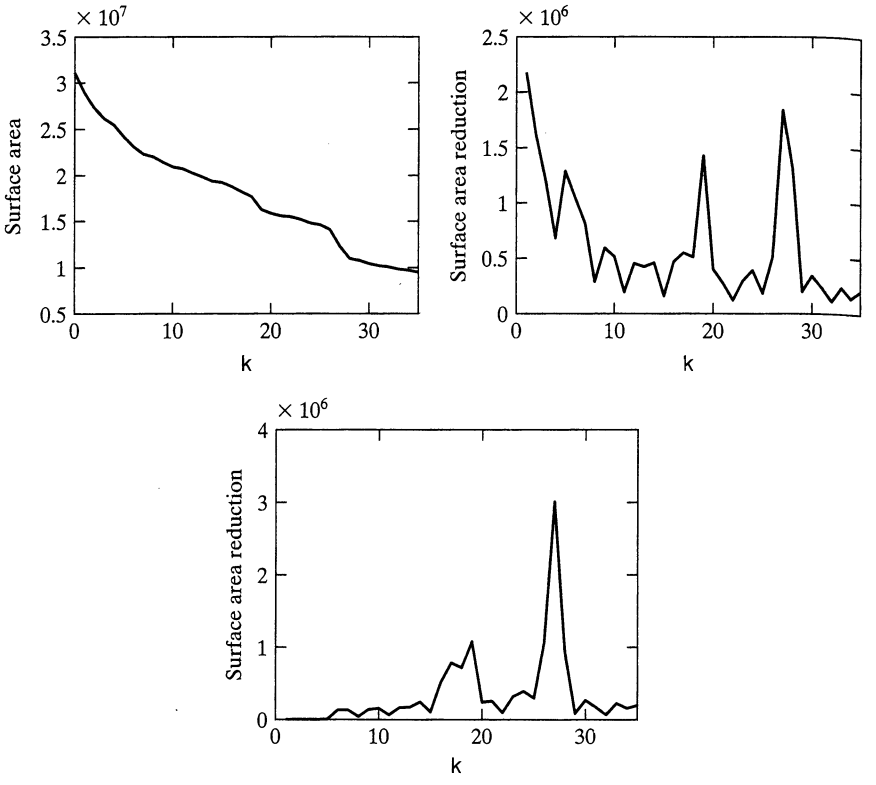


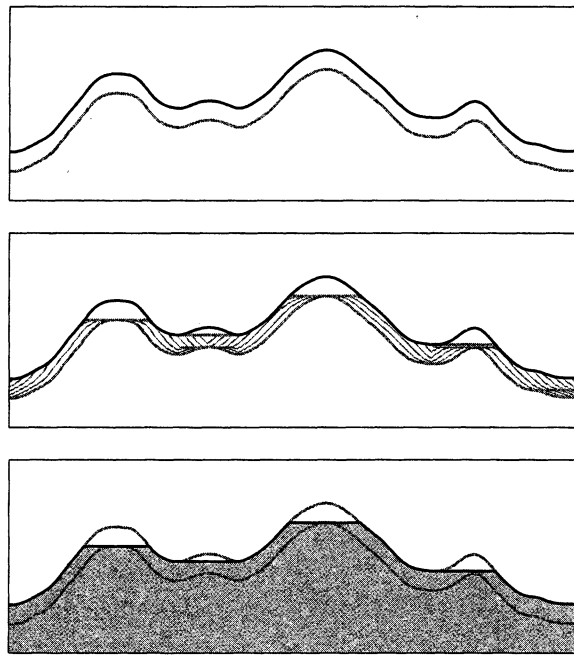
FIGURE 9.27 Granulometry. (a) Surface area versus structuring element radius. (b) Reduction in surface area versus radius. (c) Reduction in surface area versus radius for a smoothed image.

Peaks in the plot in Fig. 9.27(b) indicate the presence of a large number of objects having that radius. Since the plot is quite noisy, we repeat this procedure with the smoothed version of the plugs image in Fig. 9.25(d). The result, shown in Fig. 9.27(c), more clearly indicates the two different sizes of objects in the original image. ■

9.6.3 Reconstruction

Gray-scale morphological reconstruction is defined by the same iterative procedure given in Section 9.5. Figure 9.28 shows how reconstruction works in one dimension. The top curve of Fig. 9.28(a) is the mask while the bottom, gray curve is the marker. In this case the marker is formed by subtracting a constant from the mask, but in general any signal can be used for the marker as long as none of its values exceed the corresponding value in the mask. Each iteration of the reconstruction procedure spreads the peaks in the marker curve until they are forced downward by the mask curve [Fig. 9.28(b)].

The final reconstruction is the black curve in Fig. 9.28(c). Notice that the two smaller peaks were eliminated in the reconstruction, but the two taller peaks, although they are now shorter, remain. When a marker image is formed by subtracting a constant h from the mask image, the reconstruction is called the *h-minima transform*. The h-minima transform, computed by IPT function `imhmin`, is used to suppress small peaks.



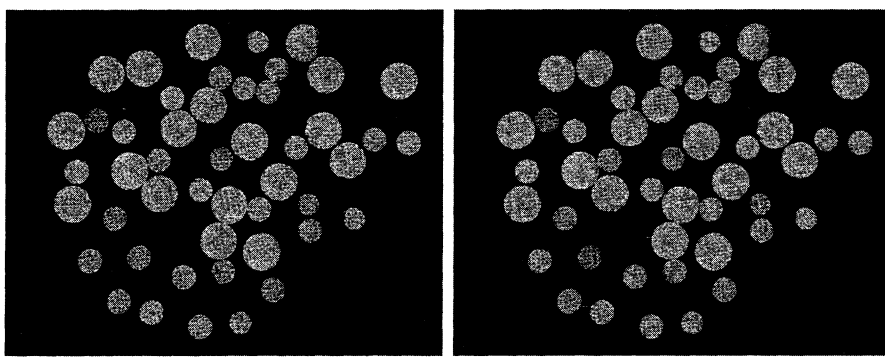
a
b
c

FIGURE 9.28 Gray-scale morphological reconstruction in one dimension. (a) Mask (top) and marker curves. (b) Iterative computation of the reconstruction. (c) Reconstruction result (black curve).

Another useful gray-scale reconstruction technique is *opening-by-reconstruction*, in which an image is first eroded, just as in standard morphological opening. However, instead of following the opening by a closing, the eroded image is used as the marker image in a reconstruction. The original image is used as the mask. Figure 9.29(a) shows an example of opening-by-reconstruction, obtained using the commands

```
>> f = imread('plugs.jpg');
>> se = strel('disk', 5);
>> fe = imerode(f, se);
>> fobr = imreconstruct(fe, f);
```

Reconstruction can be used to clean up image `fobr` further by applying to it a technique called *closing-by-reconstruction*. Closing-by-reconstruction is



a b

FIGURE 9.29 (a) Opening-by-reconstruction. (b) Opening-by-reconstruction followed by closing-by-reconstruction.

implemented by complementing an image, computing its opening-by-reconstruction, and then complementing the result. The steps are as follows:

```
>> fobrc = imcomplement(fobr);
>> fobrce = imerode(fobrc, se);
>> fobrcbr = imcomplement(imreconstruct(fobrce, fobrc));
```

Figure 9.29(b) shows the result of opening-by-reconstruction followed by closing-by-reconstruction. Compare it with the open-close filter and alternating sequential filter results in Fig. 9.25.

EXAMPLE 9.12: Using reconstruction to remove a complex image background.

■ Our concluding example uses gray-scale reconstruction in several steps. The objective is to isolate the text out of the image of calculator keys shown in Fig. 9.30(a). The first step is to suppress the horizontal reflections along the top of each key. To accomplish this, we take advantage of the fact that these reflections are wider than any single text character in the image. We perform opening-by-reconstruction using a structuring element that is a long horizontal line:

```
>> f = imread('calculator.jpg');
>> f_obr = imreconstruct(imerode(f, ones(1, 71)), f);
>> f_o = imopen(f, ones(1, 71));  % For comparison.
```

The opening-by-reconstruction (`f_obr`) is shown in Fig. 9.30(b). For comparison, Fig. 9.30(c) shows the standard opening (`f_o`). Opening-by-reconstruction did a better job of extracting the background between horizontally adjacent keys. Subtracting the opening-by-reconstruction from the original image is called *tophat-by-reconstruction*, and is shown in Fig. 9.30(d):

```
>> f_thr = imsubtract(f, f_obr);
>> f_th = imsubtract(f, f_o);  % Or imtophat(f, ones(1, 71))
```

Figure 9.30(e) shows the standard top-hat computation (i.e., `f_th`).

Next, we suppress the vertical reflections on the right edges of the keys in Fig. 9.30(d). This is done by performing opening-by-reconstruction with a small horizontal line:

```
>> g_obr = imreconstruct(imerode(f_thr, ones(1, 11)), f_thr);
```

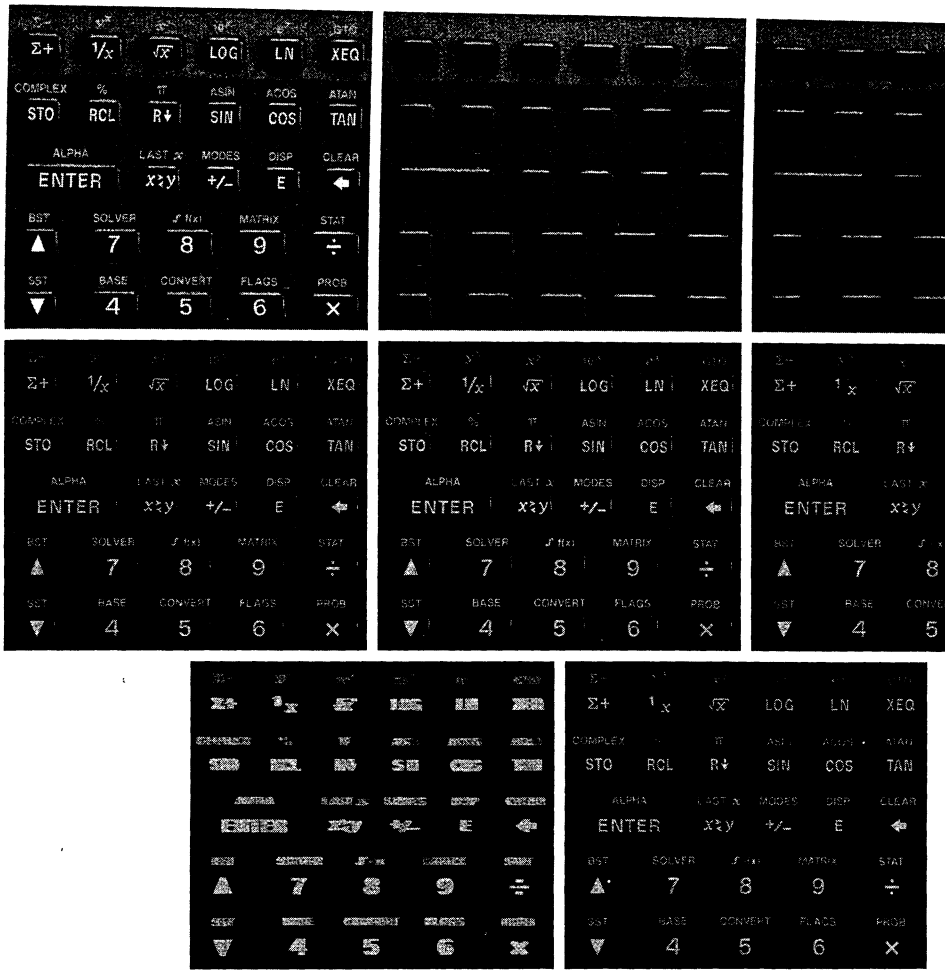
In the result [Fig. 9.30(f)], the vertical reflections are gone, but so are thin-vertical-stroke characters, such as the slash on the percent symbol and the "I" in ASIN. We take advantage of the fact that the characters that have been suppressed in error are very close to other characters still present by first performing a dilation [Fig. 9.30(g)],

```
>> g_obrd = imdilate(g_obr, ones(1, 21));
```

followed by a final reconstruction with `f_thr` as the mask and `min(g_obrd, f_thr)` as the marker:

```
>> f2 = imreconstruct(min(g_obrd, f_thr), f_thr);
```

Figure 9.30(h) shows the final result. Note that the shading and reflections on the background and keys were removed successfully. ■



a b c
d e f
g h

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.

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.