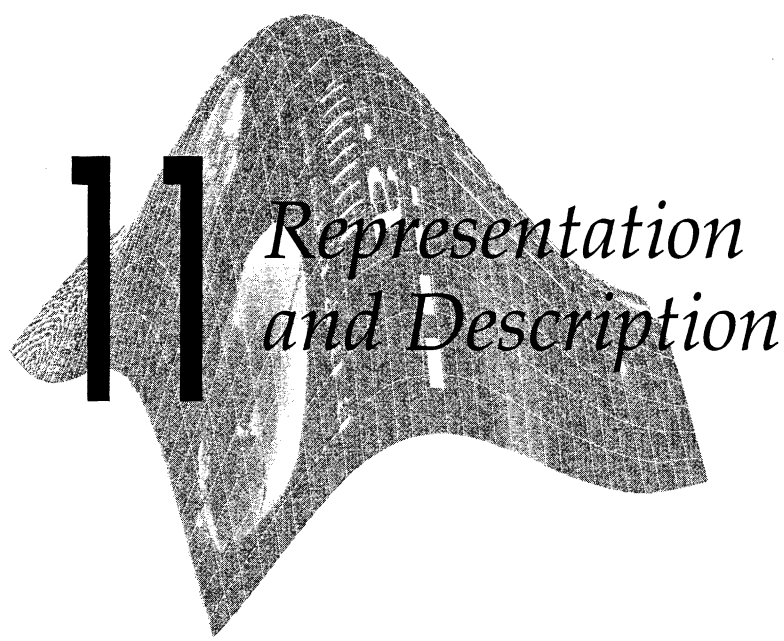


11 Representation and Description

Preview

After an image has been segmented into regions by methods such as those discussed in Chapter 10, the next step usually is to represent and describe the aggregate of segmented, "raw" pixels in a form suitable for further computer processing. Representing a region involves two basic choices: (1) We can represent the region in terms of its external characteristics (its boundary), or (2) we can represent it in terms of its internal characteristics (the pixels comprising the region). Choosing a representation scheme, however, is only part of the task of making the data useful to a computer. The next task is to *describe* the region based on the chosen representation. For example, a region may be *represented* by its boundary, and the boundary may be *described* by features such as its length and the number of concavities it contains.

An external representation is selected when interest is on shape characteristics. An internal representation is selected when the principal focus is on regional properties, such as color and texture. Both types of representations sometimes are used in the same application to solve a problem. In either case, the features selected as descriptors should be as insensitive as possible to variations in region size, translation, and rotation. For the most part, the descriptors discussed in this chapter satisfy one or more of these properties.

11.1 Background

A *region* is a connected component, and the *boundary* (also called the *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. Points not on a boundary or region are called *background* points. Initially we are interested only in binary images, so region or boundary points are represented by 1s and background points by 0s. Later in this chapter we allow pixels to have gray-scale or multispectral values.

From the definition given in the previous paragraph, it follows that a boundary is a connected set of points. The points on a boundary are said to be *ordered* if they form a clockwise or counterclockwise sequence. A boundary is said to be *minimally connected* if each of its points has exactly two 1-valued neighbors that are not 4-adjacent. An *interior point* is defined as a point anywhere in a region, except on its boundary.

The material in this chapter differs significantly from the discussions thus far in the sense that we have to be able to handle a mixture of different types of data such as boundaries, regions, topological data, and so forth. Thus, before proceeding, we pause briefly to introduce some basic MATLAB and IPT concepts and functions for use later in the chapter.

11.1.1 Cell Arrays and Structures

We begin with a discussion of MATLAB's cell arrays and structures, which were introduced briefly in Section 2.10.6.

Cell Arrays

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

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

where the curly braces designate the contents of the cell array. Typing `C` at the prompt would output the following results:

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

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

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

or we can use function `celldisp`:

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

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

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

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

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

Function size gives the size of a cell array:

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

Function cellfun, with syntax

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

See the cellfun *help page for a list of valid entries for* fname.

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

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

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

Finally, keep in mind the comment made in Section 2.10.6 that cell arrays contain copies of the arguments, not pointers to those arguments. Thus, if any of the arguments of C in the preceding example were to change after C was created, that change would not be reflected in C.

EXAMPLE 11.1: A simple illustration of cell arrays.

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

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

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

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

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

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

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

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

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

G =

    [1x2 double]    [1]    [512x1 double]    [1x512 double]

>> H

H =

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

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

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

or

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

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

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

Structures

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

EXAMPLE 11.2: A simple illustration of structures.

■ Continuing with the theme of Example 11.1 will clarify these concepts. Using structures, we write

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

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

Using the same image as in Example 11.1 and typing s and size(s) at the prompt gives the following output:

```
>> s =

s =

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

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

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

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

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

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

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

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

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

Note that the colon is in the first dimension of v and that k is in the second because s is of dimension $1 \times Q$ and AIrows is of dimension $M \times Q$. Thus, because k goes from 1 to Q, v is of dimension $M \times Q$. Had we been interested in extracting the values of AIcols instead, we would have used v(k, :) in the loop.

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

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

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

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

11.1.2 Some Additional MATLAB and IPT Functions Used in This Chapter

Function `imfill` was mentioned briefly in Table 9.3 and in Section 9.5.2. This function performs differently for binary and intensity image inputs, so, to help clarify the notation in this section, we let `fB` and `fI` represent binary and intensity images, respectively. If the output is a binary image, we denote it by `gB`; otherwise we denote simply as `g`. The syntax

```
gB = imfill(fB, locations, conn)
```

performs a flood-fill operation on background pixels (i.e., it changes background pixels to 1) of the input binary image `fB`, starting from the points specified in `locations`. This parameter can be an $n \times 1$ vector (n is the number of locations), in which case it contains the *linear indices* (see Section 2.8.2) of the starting coordinate locations. Parameter `locations` can also be an $n \times 2$ matrix, in which each row contains the 2-D coordinates of one of the starting locations in `fB`. Parameter `conn` specifies the connectivity to be used on the background pixels: 4 (the default), or 8. If both `location` and `conn` are omitted from the input argument, the command `gB = imfill(fB)` displays the binary image, `fB`, on the screen and lets the user select the starting locations using the mouse. Click the left mouse button to add points. Press **BackSpace** or **Delete** to remove the previously selected point. A shift-click, right-click, or double-click selects a final point and then starts the fill operation. Pressing **Return** finishes the selection without adding a point.

Using the syntax

```
gB = imfill(fB, conn, 'holes')
```

fills holes in the input binary image. A *hole* is a set of background pixels that cannot be reached by filling the background from the edge of the image. As before, `conn` specifies connectivity: 4 (the default) or 8.

The syntax

```
g = imfill(fI, conn, 'holes')
```

fills holes in an input intensity image, `fI`. In this case, a hole is an area of dark pixels surrounded by lighter pixels. Parameter `conn` is as before.

See Section 5.2.2 for a discussion of function `find` *and Section 9.4 for a discussion of* `bwlabel`.

Function `find` can be used in conjunction with `bwlabel` to return vectors of coordinates for the pixels that make up a specific object. For example, if `[gB, num] = bwlabel(fB)` yields more than one connected region (i.e., `num > 1`), we obtain the coordinates of, say, the second region using

```
[r, c] = find(g == 2)
```

The 2-D *coordinates* of regions or boundaries are organized in this chapter in the form of $np \times 2$ arrays, where each row is an (x, y) coordinate pair, and np is the number of points in the region or boundary. In some cases it is necessary to sort these arrays. Function `sortrows` can be used for this purpose:

```
z = sortrows(S)
```

This function sorts the rows of `S` in ascending order. Argument `S` must be either a matrix or a column vector. In this chapter, `sortrows` is used only with $np \times 2$ arrays. If several rows have identical first coordinates, they are sorted in ascending order of the second coordinate. If we want to sort the rows of `S` and also eliminate duplicate rows, we use function `unique`, which has the syntax

```
[z, m, n] = unique(S, 'rows')
```

where `z` is the sorted array with no duplicate rows, and `m` and `n` are such that `z = S(m, :)` and `S = z(n, :)`. For example, if `S = [1 2; 6 5; 1 2; 4 3]`, then `z = [1 2; 4 3; 6 5]`, `m = [3; 4; 2]`, and `n = [1; 3; 1; 2]`. Note that `z` is arranged in ascending order and that `m` indicates which rows of the original array were kept.

Frequently, it is necessary to shift the rows of an array up, down, or sideways a specified number of positions. For this we use function `circshift`:

```
z = circshift(S, [ud lr])
```

where `ud` is the number of elements by which `S` is shifted up or down. If `ud` is positive, the shift is down; otherwise it is up. Similarly, if `lr` is positive, the array is shifted to the right `lr` elements; otherwise it is shifted to the left. If only up and down shifting is needed, we can use a simpler syntax

```
z = circshift(S, ud)
```

If `S` is an image, `circshift` is really nothing more than the familiar *scrolling* (up and down) or *panning* (right and left), with the image wrapping around.

11.1.3 Some Basic Utility M-Functions

Tasks such as converting between regions and boundaries, ordering boundary points in a contiguous chain of coordinates, and subsampling a boundary to simplify its representation and description are typical of the processes that are employed routinely in this chapter. The following utility M-functions are used for these purposes. To avoid a loss of focus on the main topic of this chapter, we discuss only the syntax of these functions. The documented code for each non-MATLAB function is included in Appendix C. As noted earlier, boundaries are represented as $np \times 2$ arrays in which each row represents a 2-D pair of coordinates. Many of these functions automatically convert $2 \times np$ coordinate arrays to arrays of size $np \times 2$.

Function

boundaries

```
B = boundaries(f, conn, dir)
```

traces the *exterior* boundaries of the objects in `f`, which is assumed to be a binary image with 0s as the background. Parameter `conn` specifies the desired connectivity of the output boundaries; its values can be 4 or 8 (the default). Parameter `dir` specifies the direction in which the boundaries are traced; its values can be `'cw'` (the default) or `'ccw'`, indicating a clockwise or counterclockwise direction. Thus, if 8-connectivity and a `'cw'` direction are acceptable, we can use the simpler syntax

```
B = boundaries(f)
```

Output `B` in both syntaxes is a cell array whose elements are the coordinates of the boundaries found. The first and last points in the boundaries returned by function `boundaries` are the same. This produces a closed boundary.

As an example to fix ideas, suppose that we want to find the boundary of the object with the longest boundary in image `f` (for simplicity we assume that the longest boundary is unique). We do this with the following sequence of commands:

```
>> B = boundaries(f);
>> d = cellfun('length', B);
>> [max_d, k] = max(d);
>> v = B{k(1)};
```

See Section 2.10.2 for an explanation of this use of function `max`.

Vector `v` contains the coordinates of the longest boundary in the input image, and `k` is the corresponding region number; array `v` is of size $np \times 2$. The last statement simply selects the first boundary of maximum length if there is more than one such boundary. As noted in the previous paragraph, the first and last points of every boundary computed using function `boundaries` are the same, so row `v(1, :)` is the same as row `v(end, :)`.

Function `bound2eight` with syntax

bound2eight

```
b8 = bound2eight(b)
```

removes from `b` pixels that are necessary for 4-connectedness but not necessary for 8-connectedness, leaving a boundary whose pixels are only 8-connected. Input `b` must be an $np \times 2$ matrix, each row of which contains the (x, y) coordinates of a boundary pixel. It is required that `b` be a closed, connected set of pixels ordered sequentially in the clockwise or counterclockwise direction. The same conditions apply to function `bound2four`:

bound2four

```
b4 = bound2four(b)
```

This function inserts new boundary pixels wherever there is a diagonal connection, thus producing an output boundary in which pixels are only 4-connected. Code listings for both functions can be found in Appendix C.

Function

```
g = bound2im(b, M, N, x0, y0)
```

bound2im

generates a binary image, g, of size $M \times N$, with 1s for boundary points and a background of 0s. Parameters x0 and y0 determine the location of the minimum x- and y-coordinates of b in the image. Boundary b must be an $np \times 2$ (or $2 \times np$) array of coordinates, where, as mentioned earlier, np is the number of points. If x0 and y0 are omitted, the boundary is centered approximately in the $M \times N$ array. If, in addition, M and N are omitted, the vertical and horizontal dimensions of the image are equal to the height and width of boundary b. If function boundaries finds multiple boundaries, we can get all the coordinates for use in function bound2im by concatenating the various elements of cell array B:

```
b = cat(1, B{:})
```

See Section 6.1.1 for an explanation of the cat *operator. See also Example 11.13.*

where the 1 indicates concatenation along the first (vertical) dimension of the array.

Function

```
[s, su] = bsubsamp(b, gridsep)
```

bsubsamp

subsamples a (single) boundary b onto a grid whose lines are separated by gridsep pixels. The output s is a boundary with fewer points than b, the number of such points being determined by the value of gridsep, and su is the set of boundary points scaled so that transitions in their coordinates are unity. This is useful for coding the boundary using chain codes, as discussed in Section 11.1.2. It is required that the points in b be ordered in a clockwise or counterclockwise direction.

When a boundary is subsampled using bsubsamp, its points cease to be connected. They can be reconnected by using

```
z = connectpoly(s(:, 1), s(:, 2))
```

connectpoly

where the rows of s are the coordinates of a subsampled boundary. It is required that the points in s be ordered, either in a clockwise or counterclockwise direction. The rows of output z are the coordinates of a connected boundary formed by connecting the points in s with the shortest possible path consisting of 4- or 8-connected straight segments. This function is useful for producing a polygonal, fully connected boundary that is generally smoother (and simpler) than the original boundary, b, from which s was obtained. Function connectpoly also is quite useful when working with functions that generate only the vertices of a polygon, such as minperpoly, discussed in Section 11.2.3.

Computing the integer coordinates of a straight line joining two points is a basic tool when working with boundaries (for example, function `connectpoly` requires a subfunction that does this). IPT function `intline` is well suited for this purpose. Its syntax is

```
[x, y] = intline(x1, x2, y1, y2)
```

`intline` *is an undocumented IPT utility function. Its code is included in Appendix C.*

where `(x1, y1)` and `(x2, y2)` are the integer coordinates of the two points to be connected. The outputs `x` and `y` are column vectors containing the integer x- and y-coordinates of the straight line joining the two points.

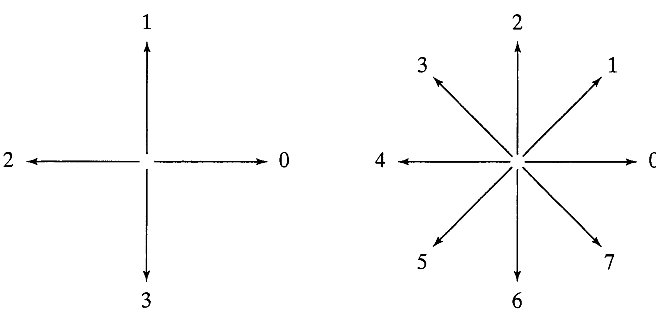
11.2 Representation

As noted at the beginning of this chapter, the segmentation techniques discussed in Chapter 10 yield raw data in the form of pixels along a boundary or pixels contained in a region. Although these data sometimes are used directly to obtain descriptors (as in determining the texture of a region), standard practice is to use schemes that compact the data into representations that are considerably more useful in the computation of descriptors. In this section we discuss the implementation of various representation approaches.

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. Typically, this representation is based on 4- or 8-connectivity of the segments. The direction of each segment is coded by using a numbering scheme such as the ones shown in Figs. 11.1(a) and (b). Chain codes based on this scheme are referred to as *Freeman chain codes*.

The chain code of a boundary depends on the starting point. However, the code can be normalized with respect to the starting point by treating it as a circular sequence of direction numbers and redefining the starting point so that the resulting sequence of numbers forms an integer of minimum magnitude. We can normalize for rotation [in increments of 90° or 45°, as shown in Figs. 11.1(a) and (b)] by using the *first difference* of the chain code instead of the code itself. This difference is obtained by counting the number of direction changes (in a coun-



a b

FIGURE 11.1 (a) Direction numbers for (a) a 4-directional chain code, and (b) an 8-directional chain code.

terclockwise direction in Fig. 11.1) that separate two adjacent elements of the code. For instance, the first difference of the 4-direction chain code 10103322 is 3133030. If we elect to treat the code as a circular sequence, then the first element of the difference is computed by using the transition between the last and first components of the chain. Here, the result is 33133030. Normalization with respect to arbitrary rotational angles is achieved by orienting the boundary with respect to some dominant feature, such as its major axis, as discussed in Section 11.3.2.

Function `fchcode`, with syntax

```
c = fchcode(b, conn, dir)
```

fchcode

computes the Freeman chain code of an $np \times 2$ set of ordered boundary points stored in array `b`. The output `c` is a structure with the following fields, where the numbers inside the parentheses indicate array size:

`c.fcc` = Freeman chain code ($1 \times np$)
`c.diff` = First difference of code `c.fcc` ($1 \times np$)
`c.mm` = Integer of minimum magnitude ($1 \times np$)
`c.diffmm` = First difference of code `c.mm` ($1 \times np$)
`c.x0y0` = Coordinates where the code starts (1×2)

Parameter `conn` specifies the connectivity of the code; its value can be 4 or 8 (the default). A value of 4 is valid only when the boundary contains no diagonal transitions.

Parameter `dir` specifies the direction of the output code: If `'same'` is specified, the code is in the same direction as the points in `b`. Using `'reverse'` causes the code to be in the opposite direction. The default is `'same'`. Thus, writing `c = fchcode(b, conn)` uses the default direction, and `c = fchcode(b)` uses the default connectivity and direction.

EXAMPLE 11.3: Freeman chain code and some of its variations.

■ Figure 11.2(a) shows an image, `f`, of a circular stroke embedded in specular noise. The objective of this example is to obtain the chain code and first difference of the object's boundary. It is obvious by looking at Fig. 11.2(a) that the noise fragments attached to the object would result in a very irregular boundary, not truly descriptive of the general shape of the object. Smoothing is a routine process when working with noisy boundaries. Figure 11.2(b) shows the result, `g`, of using a 9×9 averaging mask:

```
>> h = fspecial('average', 9);
>> g = imfilter(f, h, 'replicate');
```

The binary image in Fig. 11.2(c) was then obtained by thresholding:

```
>> g = im2bw(g, 0.5);
```

The boundary of this image was computed using function `boundaries` discussed in the previous section:

```
>> B = boundaries(g);
```

As in the illustration in Section 11.1.3, we are interested in the longest boundary:

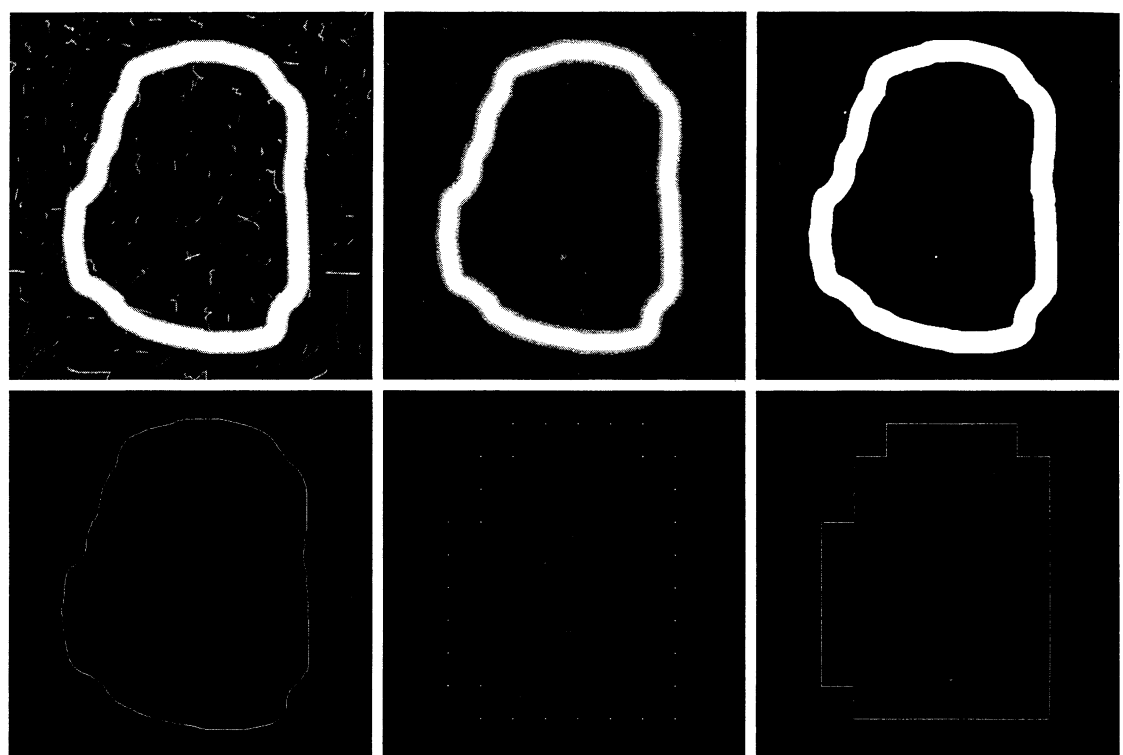
```
>> d = cellfun('length', B);
>> [max_d, k] = max(d);
>> b = B{1};
```

The boundary image in Fig. 11.2(d) was generated using the commands:

```
>> [M N] = size(g);
>> g = bound2im(b, M, N, min(b(:, 1)), min(b(:, 2)));
```

Obtaining the chain code of b directly would result in a long sequence with small variations that are not necessarily representative of the general shape of the image. Thus, as is typical in chain-code processing, we subsample the boundary using function bsubsamp discussed in the previous section:

```
>> [s, su] = bsubsamp(b, 50);
```



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).

Here, we used a grid separation equal to approximately 10% the width of the image, which in this case was of size 570 × 570 pixels. The resulting points can be displayed as an image [Fig. 11.2(e)]:

```
>> g2 = bound2im(s, M, N, min(s(:, 1)), min(s(:, 2)));
```

or as a connected sequence [Fig. 11.2(f)] by using the commands

```
>> cn = connectpoly(s(:, 1), s(:, 2));
>> g2 = bound2im(cn, M, N, min(cn(:, 1)), min(cn(:, 2)));
```

The advantage of using this representation, as opposed to Fig. 11.2(d), for chain-coding purposes is evident by comparing this figure with Fig. 11.2(f). The chain code is obtained from the scaled sequence `su`:

```
>> c = fchcode(su);
```

This command resulted in the following outputs:

```
>> c.x0y0

ans =

     7     3
>> c.fcc

ans =

2 2 0 2 2 0 2 0 0 0 0 6 0 6 6 6 6 6 6 6 6 4 4 4 4 4 4 2 4 2 2 2
>> c.mm

ans =

0 0 0 0 6 0 6 6 6 6 6 6 6 6 4 4 4 4 4 4 2 4 2 2 2 2 2 0 2 2 0 2
>> c.diff

ans =

0 6 2 0 6 2 6 0 0 0 6 2 6 0 0 0 0 0 0 0 6 0 0 0 0 0 6 2 6 0 0 0
>> c.diffmm

ans =

0 0 0 6 2 6 0 0 0 0 0 0 0 6 0 0 0 0 0 6 2 6 0 0 0 0 6 2 0 6 2 6
```

By examining `c.fcc`, Fig. 11.2(f), and `c.x0y0` we see that the code starts on the left of the figure and proceeds in the clockwise direction, which is the same direction as the coordinates of the boundary. ■

11.2.2 Polygonal Approximations Using Minimum-Perimeter Polygons

A digital boundary can be approximated with arbitrary accuracy by a polygon. For a closed curve, the approximation is exact when the number of segments in the polygon is equal to the number of points in the boundary, so that each pair

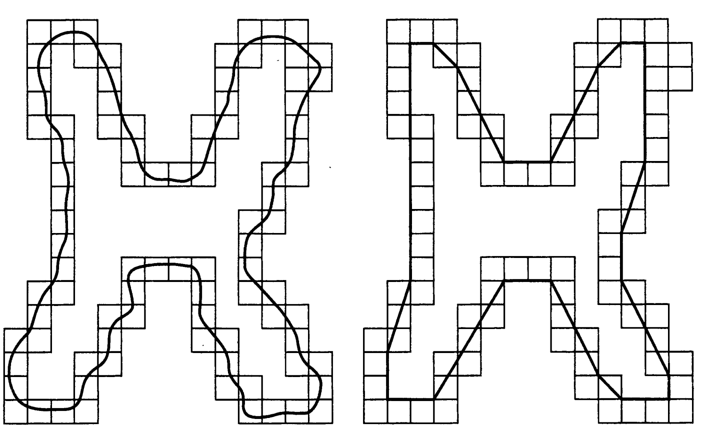
of adjacent points defines an edge of the polygon. In practice, the goal of a polygonal approximation is to use the fewest vertices possible to capture the "essence" of the boundary shape.

A particularly attractive approach to polygonal approximation is to find the *minimum-perimeter polygon* (MPP) of a region or boundary. The theoretical underpinnings and an algorithm for finding MPPs are discussed in the classic paper by Sklansky et al. [1972] (see also Kim and Sklansky [1982]). In this section we present the fundamentals of the algorithm and give an M-function implementation of the procedure. The method is restricted to *simple* polygons (i.e., polygons with no self-intersections). Also, regions with peninsular protrusions that are one pixel thick are excluded. Such protrusions can be extracted using morphological methods and then reappended after the polygonal approximation has been computed.

Foundation

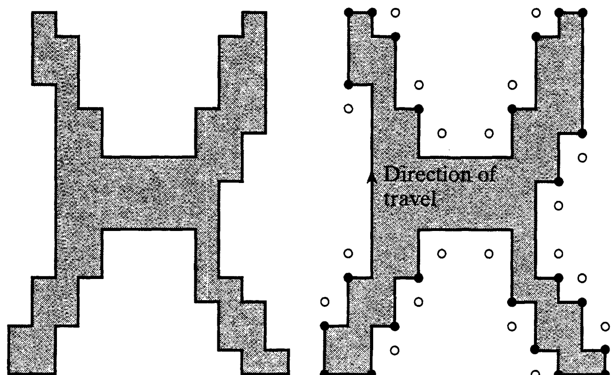
We begin with a simple example to fix ideas. Suppose that we enclose a boundary by a set of concatenated cells, as shown in Fig. 11.3(a). It helps to visualize this enclosure as two walls corresponding to the outside and inside boundaries of the strip of cells and think of the object boundary as a rubber band contained within the two walls. If the rubber band is allowed to shrink, it takes the shape shown in Fig. 11.3(b), producing a polygon of minimum perimeter that fits the geometry established by the cell strip.

Sklansky's approach uses a so-called *cellular complex* or *cellular mosaic*, which, for our purposes, is the set of *square* cells used to enclose a boundary, as in Fig. 11.3(a). Figure 11.4(a) shows the region (shaded) enclosed by the cellular complex. Note that the boundary of this region forms a 4-connected path. As we traverse this path in a clockwise direction, we assign a black dot (•) to the convex corners (those with interior angles equal to 90°) and a white dot (∘) to the concave corners (those with interior angles equal to 270°). As Fig. 11.4(b) shows, the black dots are placed on the convex corners themselves. The white dots are placed diagonally opposite their corresponding concave corners. This corresponds to the cellular complex and vertex definitions of the algorithm.



a b

FIGURE 11.3 (a) Object boundary enclosed by cells. (b) Minimum-perimeter polygon.



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.

The following properties are basic in formulating an approach for finding MPPs:

1. The MPP corresponding to a simply connected cellular complex is not self-intersecting. Let P denote this MPP.
2. Every *convex* vertex of P coincides with a • (but not every • is a vertex of P).
3. Every *concave* vertex of P coincides with a ∘ (but not every ∘ is a vertex of P).
4. If a • in fact is part of P, but it is not a convex vertex of P, then it lies on the edge of P.

In our discussion, a vertex of a polygon is defined to be *convex* if its *interior* angle is in the range $0° < \theta < 180°$; otherwise the vertex is *concave.* As in the previous paragraph, convexity is measured with respect to the interior region as we travel in a clockwise direction.

The condition $\theta = 0°$ is not allowed, and $\theta = 180°$ is treated as a special case.

An Algorithm for Finding MPPs

Properties 1 through 4 are the basis for finding the vertices of an MPP. There are various ways to do this (e.g., see Sklansky et al. [1972], and Kim and Sklansky [1982]). The approach we follow here is designed to take advantage of two basic IPT/MATLAB functions. The first is `qtdecomp`, which performs quadtree decompositions that lead to the cellular wall enclosing the data of interest. The second is function `inpolygon`, used to determine which points lie outside, on, or inside the boundary of a polygon defined by a given set of vertices.

It will be helpful to develop the procedure for finding MPPs in the context of an illustration. We use Figs. 11.3 and 11.4 again for this purpose. An approach for finding the 4-connected boundary of the shaded inner region in Fig. 11.4(a) is discussed later in this section. After the boundary has been obtained, the next step is to find its corners, which we do by obtaining its Freeman chain code. Changes in code direction indicate a corner in the boundary. By analyzing direction changes as we travel in a clockwise direction through the boundary, it becomes a fairly easy task to determine and mark the convex and concave corners, as in Fig. 11.4(b). The specific approach for obtaining the

markers is documented in M-function `minperpoly` discussed later in this section. The corners determined in this manner are as in Fig. 11.4(b), which we show again in Fig. 11.5(a). The shaded region and background grid are included for easy reference. The boundary of the shaded region is not shown to avoid confusion with the polygonal boundaries shown throughout Fig. 11.5.

Next, we form an initial polygon using only the initial convex vertices (the black dots), as Fig. 11.5(b) shows. We know from property 2 that the set of MPP convex vertices is a subset of this initial set of convex vertices. We see that all the concave vertices (white dots) lying *outside* the initial polygon do not form concavities in the polygon. For those particular vertices to become convex at a later stage in the algorithm, the polygon would have to pass through them. But, we know that they can never become convex because all possible convex vertices are accounted for at this point (it is possible that their angle could become 180° later, but that would have no effect on the shape of the polygon). Thus, the white dots outside the initial polygon can be eliminated from further analysis, as Fig. 11.5(c) shows.

The concave vertices (white dots) inside the polygon are associated with concavities in the boundary that were ignored in the first pass. Thus, these vertices must be incorporated into the polygon, as shown in Fig. 11.5(d). At this point generally there are vertices that are black dots but that have ceased to be convex in the new polygon [see the black dots marked with arrows in Fig. 11.5(d)]. There are two possible reasons for this. The first reason may be that these vertices are part of the starting polygon in Fig. 11.5(b), which includes *all* convex (black) vertices. The second reason could be that they have become convex as a result of our having incorporated additional (white) vertices into the polygon as in Fig. 11.5(d). Therefore, all black dots in the polygon must be tested to see if any of the vertex angles at those points now exceed 180°. All those that do are deleted. The procedure in then repeated.

Figure 11.5(e) shows only one new black vertex that has become concave during the second pass through the data. The procedure terminates when no further vertex changes take place, at which time all vertices with angles of 180° are deleted because they are on an edge, and thus do not affect the shape of the final polygon. The boundary in Fig. 11.5(f) is the MPP for our example. This polygon is the same as the polygon in Fig. 11.3(b). Finally, Fig. 11.4(g) shows the original cellular complex superimposed on the MPP.

The preceding discussion is summarized in the following steps for finding the MPP of a region:

1. Obtain the cellular complex (the approach is discussed later in this section).
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. Form an initial polygon using the black dots as vertices, and delete from further analysis any white dots that are outside this polygon (white dots on the polygon boundary are kept).

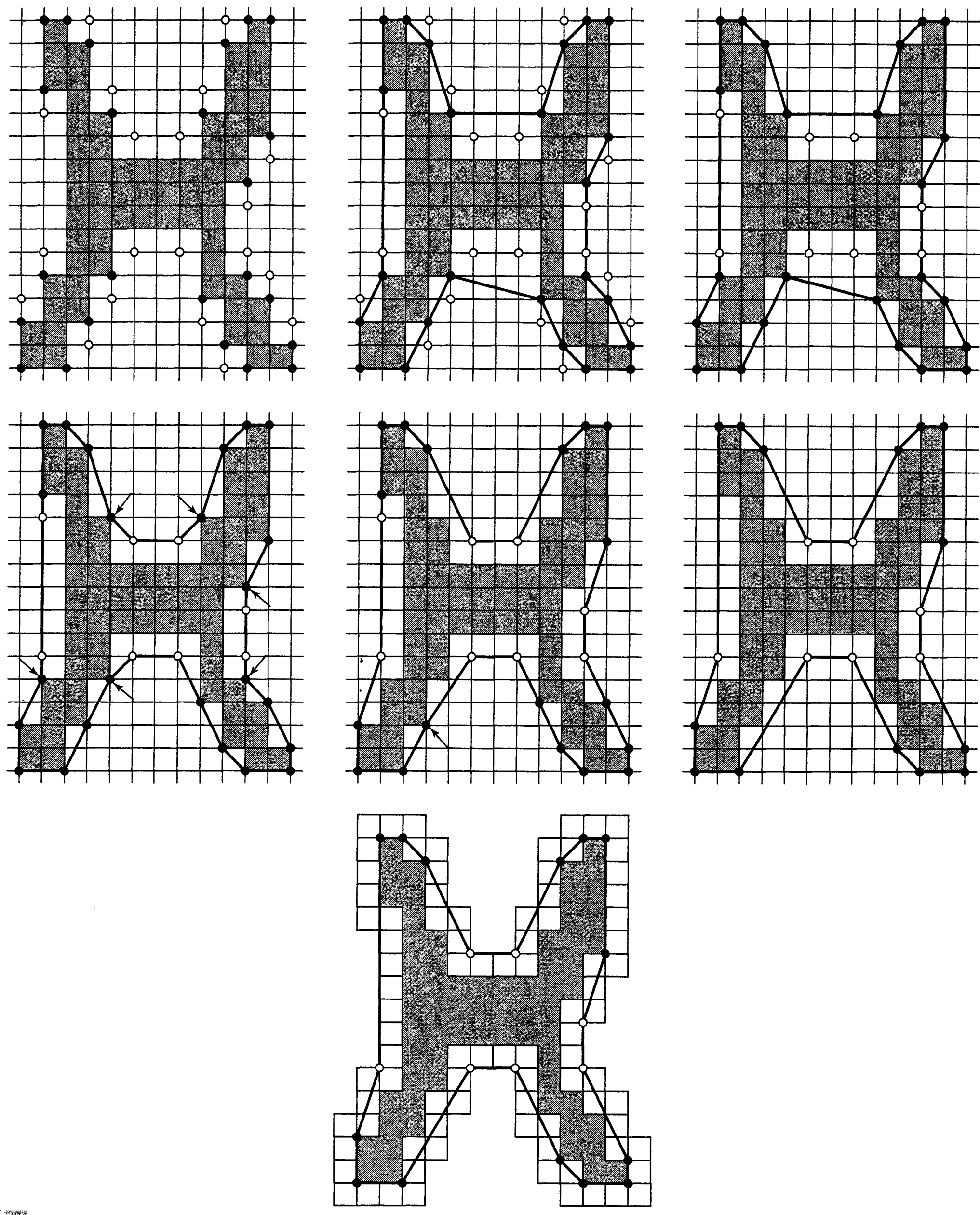


FIGURE 11.5 (a) Convex (black) and concave (white) vertices of the boundary in Fig. 11.4(a). (b) Initial polygon joining all convex vertices. (c) Result after deleting concave vertices outside of the polygon. (d) Result of incorporating the remaining concave vertices into the polygon (the arrows indicate 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.

7. Form 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° are deleted. The remaining dots are the vertices of the MPP.

Some of the M-Functions Used in Implementing the MPP Algorithm

We use function `qtdecomp` introduced in Section 10.4.2 as the first step in obtaining the cellular complex enclosing a boundary. As usual, we consider the region, `B`, in question to be composed of 1s and the background of 0s. The `qtdecomp` syntax applicable to our work here is

```
Q = qtdecomp(B, threshold, [mindim maxdim])
```

where `Q` is a sparse matrix containing the quadtree structure. If `Q(k, m)` is nonzero, then `(k, m)` is the upper-left corner of a block in the decomposition and the size of the block is `Q(k, m)`.

A block is split if the maximum value of the block elements minus the minimum value of the block elements is greater than `threshold`. The value of this parameter is specified between 0 and 1, regardless of the class of the input image. Using the preceding syntax, function `qtdecomp` will not produce blocks smaller than `mindim` or larger than `maxdim`. Blocks larger than `maxdim` are split even if they do not meet the threshold condition. The ratio `maxdim/mindim` must be a power of 2.

If only one of the two values is specified (without the brackets), the function assumes that it is `mindim`. This is the formulation we use in this section. Image `B` must be of size `K` × `K`, such that the ratio of `K/mindim` is an integer power of 2. Clearly, the smallest possible value of `K` is the largest dimension of `B`. The size requirements generally are met by padding `B` with zeros with option `'post'` in function `padarray`. For example, suppose that `B` is of size 640 × 480 pixels, and we specify `mindim = 3`. Parameter `K` has to satisfy the conditions `K >= max(size(B))` and `K/mindim = 2^p`, or `K = mindim*(2^p)`. Solving for `p` gives `p = 8`, in which case `K = 768`.

To get the block values in a quadtree decomposition we use function `qtgetblk`, discussed in Section 10.4.2:

```
[vals, r, c] = qtgetblk(B, Q, mindim)
```

where `vals` is an array containing the values of the `mindim` × `mindim` blocks in the quadtree decomposition of `B`, and `Q` is the sparse matrix returned by `qtdecomp`. Parameters `r` and `c` are vectors containing the row and column coordinates of the upper-left corners of the blocks.

EXAMPLE 11.4: Obtaining the cellular wall of the boundary of a region.

■ To see how steps 1 through 4 of the MPP algorithm are implemented, consider the image in Fig. 11.6(a), and suppose that we specify `mindim = 2`. We show individual pixels as small squares to facilitate explanation of function `qtdecomp`. The image is of size 32 × 32, and it is easily verified that no addi-

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`.

tional padding is required for the specified value of `mindim`. The 4-connected boundary of the region is obtained using the following command (the margin note in the next page explains why 8 is used in the function call):

```
>> B = bwperim(B, 8);
```

The syntax for `bwperim` *is* `g = bwperim(f, conn)` *where* `conn` *identifies the desired connectivity: 4 (the default) or 8. The connectivity is with respect to the background pixels. Thus, to obtain 4-connected object boundaries we specify 8 for* `conn`. *Conversely, 8-connected boundaries result from specifying a value of 4 for* `conn`. *Output* `g` *is a binary image containing the boundaries of the objects in* `f`. *This function is discussed in detail in Section 11.3.1.*

Figure 11.6(b) shows the result. Note that `B` is still an image, which now contains only a 4-connected boundary (keep in mind that the small squares are individual pixels).

Figure 11.6(c) shows the quadtree decomposition of `B`, obtained using the command

```
>> Q = qtdecomp(B, 0, 2);
```

where 0 was used for the threshold so that blocks were split down to the minimum 2 × 2 size, regardless of the mixture of 1s and 0s they contained (each such block is capable of containing between zero and four pixels). Note that there are numerous blocks of size greater than 2 × 2, but they are all homogeneous.

Next we used `qtgetblk(B, Q, 2)` to extract the values and top-left corner coordinates of all the blocks of size 2 × 2. Then all the blocks that contained at least one pixel valued 1 were filled with 1s. This result, which we denote by `BF`, is shown in Fig. 11.6(d). The dark cells in this image constitute the cellular complex. In other words, these cells enclose the boundary in Fig. 11.6(b).

The region bounded by the cellular complex in Fig. 11.6(d) was obtained using the command

```
>> R = imfill(BF, 'holes') & ~BF;
```

Figure 11.6(e) shows the result. We are interested in the 4-connected boundary of this region, which we obtain using the commands

```
>> b = boundaries(b, 4, 'cw');
>> b  = b{1};
```

Figure 11.6(f) shows the result. The Freeman chain code shown in this figure was obtained using function `fchcode`. This completes steps 1 through 4 of the MPP algorithm. ■

Function `inpolygon` is used in function `minperpoly` (discussed in the next section) to determine whether a point is outside, on the boundary, or inside a polygon; the syntax is

```
IN = inpolygon(X, Y, xv, yv)
```

where `X` and `Y` are vectors containing the x- and y-coordinates of the points to be tested, and `xv` and `yv` are vectors containing the the x- and y-coordinates of the polygon vertices, arranged in a clockwise or counterclockwise sequence. Array `IN` is a vector whose length is equal to the number of points being tested. Its values are 1 for points inside or on the boundary of the polygon, and 0 for points outside the boundary.

An M-Function for Computing MPPs

Steps 1 through 9 of the MPP algorithm are implemented in function `minperpoly`, whose listing is included in Appendix C. The syntax is

```
[x, y] = minperpoly(B, cellsize)
```

minperpoly

where `B` is an input binary image containing a single region or boundary, and `cellsize` is the size of the square cells in the cellular complex used to enclose the boundary. Column vectors `x` and `y` contain the x- and y-coordinates of the MPP vertices.

EXAMPLE 11.5: Using function `minperpoly`.

■ Figure 11.7(a) shows an image, `B`, of a maple leaf, and Fig. 11.7(b) is the boundary obtained using the commands

```
>> 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)
```

This is the reference boundary against which various MMPs are compared in this example. Figure 11.7(c) is the result of using the commands

```
>> [x, y] = minperpoly(B, 2);
>> b2 = connectpoly(x, y);
>> B2 = bound2im(b2, M, N, xmin, ymin);
>> imshow(B2)
```

Similarly, Figs. 11.7(d) through (f) show the MPPs obtained using square cells of sizes 3, 4, and 8. The thin stem is lost with cells larger than 2×2 due to a loss of resolution. The second major shape characteristic of the leaf is its set of three main lobes. These are preserved reasonably well even for cells of size 8, as Fig. 11.7(f) shows. Further increases in the size of the cells to 10 and even to 16 still preserve this feature, as Figs. 11.8(a) and (b) show. However, as shown in Figs. 11.8(c) and (d), values of 20 and higher cause this characteristic to be lost.

The arrows in Figs. 11.7(c) and (e) point to nodes formed by self-intersecting lines. These nodes can arise if the size of the indentation in the boundary with respect to the cell size is such that when the concave vertices are created, their positions "cross" each other, altering the clockwise sequence of the vertices. One approach for solving this problem is to delete one of the vertices. The other is to increase or decrease the cell size. For example, Fig. 11.7(d), which corresponds to a cell size of 3, does not have the problem exhibited by the vertices generated with cells of sizes 2 and 4. ■

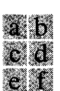
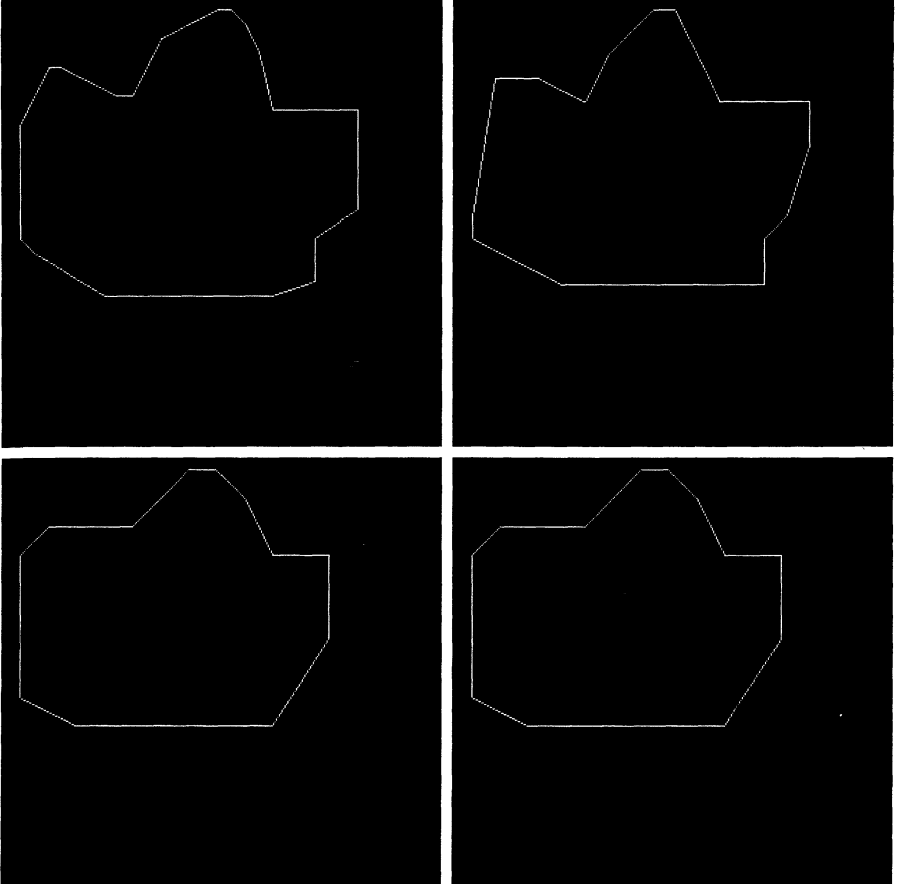


FIGURE 11.7
(a) Original image.
(b) 4-connected boundary.
(c) MPP obtained using square bounding cells of size 2. (d) through (f) MPPs obtained using square cells of sizes 3, 4, and 8, respectively.



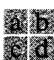
a b
c d

FIGURE 11.8 MPPs obtained with even larger bounding square cells of sizes (a) 10, (b) 16, (c) 20, and (d) 32.

11.2.3 Signatures

A *signature* is a 1-D functional representation of a boundary and may be generated in various ways. One of the simplest is to plot the distance from an interior point (e.g., the centroid) to the boundary as a function of angle, as illustrated in Fig. 11.9. Regardless of how a signature is generated, however, the basic idea is to reduce the boundary representation to a 1-D function, which presumably is easier to describe than the original 2-D boundary. Keep in mind that it makes sense to consider using signatures only when it can be guaranteed that the vector extending from its origin to the boundary intersects the boundary only once, thus yielding a single-valued function of increasing angle. This excludes boundaries with self-intersections, and it also typically excludes boundaries with deep, narrow concavities or thin, long protrusions.

Signatures generated by the approach just described are invariant to translation, but they do depend on rotation and scaling. Normalization with respect to



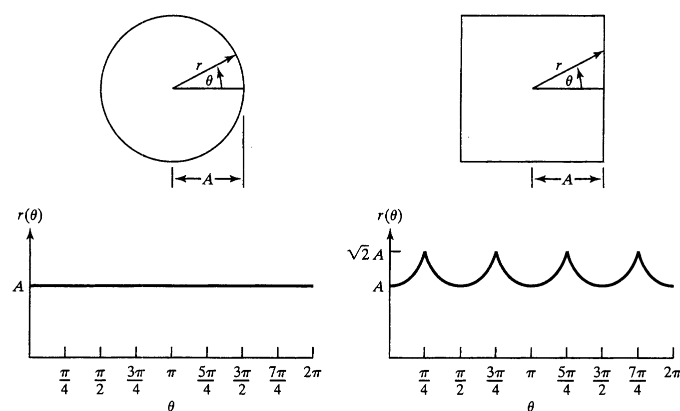


FIGURE 11.9 (a) and (b) Circular and square objects. (c) and (d) Corresponding distance versus angle signatures.

rotation can be achieved by finding a way to select the same starting point to generate the signature, regardless of the shape's orientation. One way to do so is to select the starting point as the point farthest from the origin of the vector (see Section 11.3.1), if this point happens to be unique and independent of rotational aberrations for each shape of interest.

Another way is to select a point on the major eigen axis (see Section 11.5). This method requires more computation but is more rugged because the direction of the eigen axes is determined by using all contour points. Yet another way is to obtain the chain code of the boundary and then use the approach discussed in Section 11.1.2, assuming that the rotation can be approximated by the discrete angles in the code directions defined in Fig. 11.1.

Based on the assumptions of uniformity in scaling with respect to both axes, and that sampling is taken at equal intervals of θ, changes in size of a shape result in changes in the amplitude values of the corresponding signature. One way to normalize for this dependence is to scale all functions so that they always span the same range of values, say, $[0, 1]$. The main advantage of this method is simplicity, but it has the potentially serious disadvantage that scaling of the entire function is based on only two values: the minimum and maximum. If the shapes are noisy, this can be a source of error from object to object. A more rugged approach is to divide each sample by the variance of the signature, assuming that the variance is not zero—as in the case of Fig. 11.9(a)—or so small that it creates computational difficulties. Use of the variance yields a variable scaling factor that is inversely proportional to changes in size and works much as automatic gain control does. Whatever the method used, keep in mind that the basic idea is to remove dependency on size while preserving the fundamental shape of the waveforms.

Function `signature`, included in Appendix C, finds the signature of a given boundary. Its syntax is

signature

```
[st, angle, x0, y0] = signature(b, x0, y0)
```

where `b` is an $np \times 2$ array containing the xy-coordinates of a boundary ordered in a clockwise or counterclockwise direction. The amplitude of the

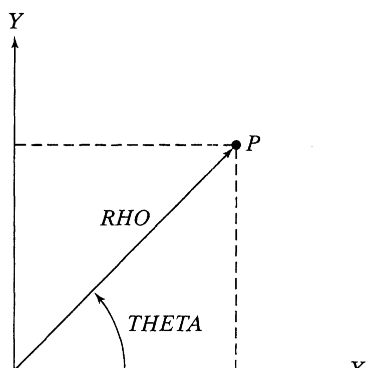


FIGURE 11.10 Axis convention used by MATLAB for performing conversions between polar and Cartesian coordinates, and vice versa.

signature as a function of increasing `angle` is output in `st`. Coordinates `(x0, y0)` in the input are the coordinates of the origin of the vector extending to the boundary. If these coordinates are not included in the argument, the function uses the coordinates of the centroid of the boundary by default. In either case, the values of `(x0, y0)` used by the function are included in the output. The size of arrays `st` and `angle` is 360×1, indicating a resolution of one degree. The input must be a one-pixel-thick boundary obtained, for example, using function `boundaries` (see Section 11.1.3). As before, we assume that a boundary is a closed curve.

Function `signature` utilizes MATLAB's function `cart2pol` to convert Cartesian to polar coordinates. The syntax is

```
[THETA, RHO] = cart2pol(X, Y)
```

where `X` and `Y` are vectors containing the coordinates of the Cartesian points. The vectors `THETA` and `RHO` contain the corresponding angle and length of the polar coordinates. If `X` and `Y` are row vectors, so are `THETA` and `RHO`, and similarly in the case of columns. Figure 11.10 shows the convention used by MATLAB for coordinate conversions. Note that the MATLAB coordinates `(X, Y)` in this situation are related to our image coordinates `(x, y)` as `X = y` and `Y = -x` [see Fig. 2.1(a)]. Function `pol2cart` is used for converting back to Cartesian coordinates:

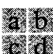
```
[X, Y] = pol2cart(THETA, RHO)
```

EXAMPLE 11.6: Signatures.

■ Figures 11.11(a) and (b) show the boundaries, `bs` and `bt`, of an irregular square and triangle, respectively, embedded in arrays of size 674×674 pixels. Figure 11.11(c) shows the signature of the square, obtained using the commands

```
>> [st, angle, x0, y0] = signature(bs);
>> plot(angle, st)
```

The values of `x0` and `y0` obtained in the preceding command were [342, 326]. A similar pair of commands yielded the plot in Fig. 11.11(d), whose centroid is



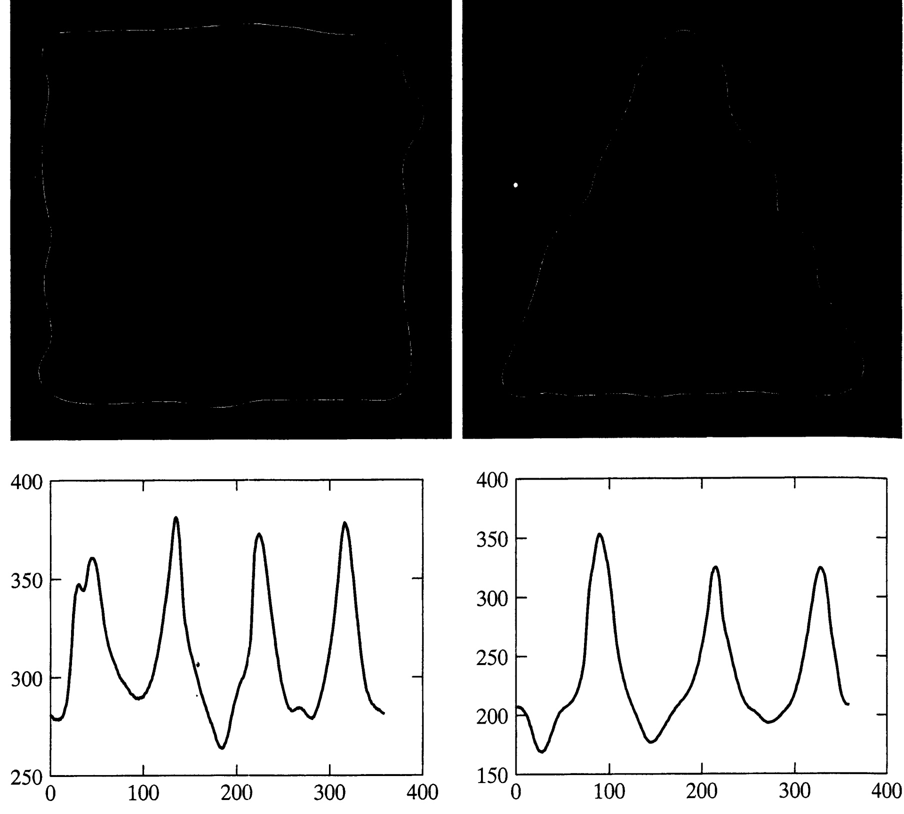


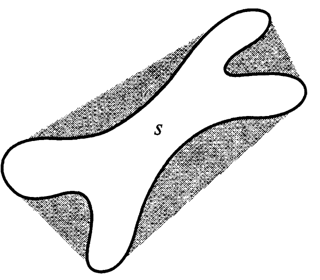
FIGURE 11.11 (a) and (b) Boundaries of an irregular square and triangle. (c) and (d) Corresponding signatures.

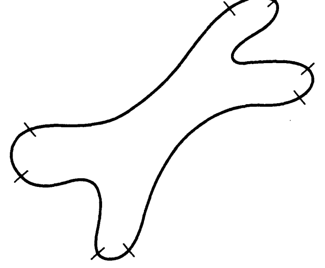
located at [416, 335]. Note that simply counting the number of prominent peaks in the two signatures is sufficient to differentiate between the two boundaries. ■

11.2.4 Boundary Segments

Decomposing a boundary into segments often is useful. Decomposition reduces the boundary's complexity and thus simplifies the description process. This approach is particularly attractive when the boundary contains one or more significant concavities that carry shape information. In this case use of the convex hull of the region enclosed by the boundary is a powerful tool for robust decomposition of the boundary.

The *convex hull* H of an arbitrary set S is the smallest convex set containing S. The set difference $H - S$ is called the *convex deficiency*, D, of the set S. To see how these concepts might be used to partition a boundary into meaningful segments, consider Fig. 11.12(a), which shows an object (set S) and its convex deficiency (shaded regions). The region boundary can be partitioned by following the contour of S and marking the points at which a transition is made into or out of a component of the convex deficiency. Figure 11.12(b) shows the result in this case. In principle, this scheme is independent of region size and





a b

FIGURE 11.12 (a) A region S and its convex deficiency (shaded). (b) Partitioned boundary.

orientation. In practice, this type of processing is preceded typically by aggressive image smoothing to reduce the number of "insignificant" concavities. The MATLAB tools necessary to implement boundary decomposition in the manner just described are contained in function `regionprops`, which is discussed in Section 11.4.1.

11.2.5 Skeletons

An important approach for representing the structural shape of a plane region is to reduce it to a graph. This reduction may be accomplished by obtaining the *skeleton* of the region via a thinning (also called *skeletonizing*) algorithm.

The skeleton of a region may be defined via the medial axis transformation (MAT). The MAT of a region R with border b is as follows. For each point p in R, we find its closest neighbor in b. If p has more than one such neighbor, it is said to belong to the *medial axis* (skeleton) of R.

Although the MAT of a region is an intuitive concept, direct implementation of this definition is expensive computationally, as it involves calculating the distance from every interior point to every point on the boundary of a region. Numerous algorithms have been proposed for improving computational efficiency while at the same time attempting to approximate the medial axis representation of a region.

As noted in Section 9.3.4, IPT generates the skeleton of all regions contained in a binary image `B` via function `bwmorph`, using the following syntax:

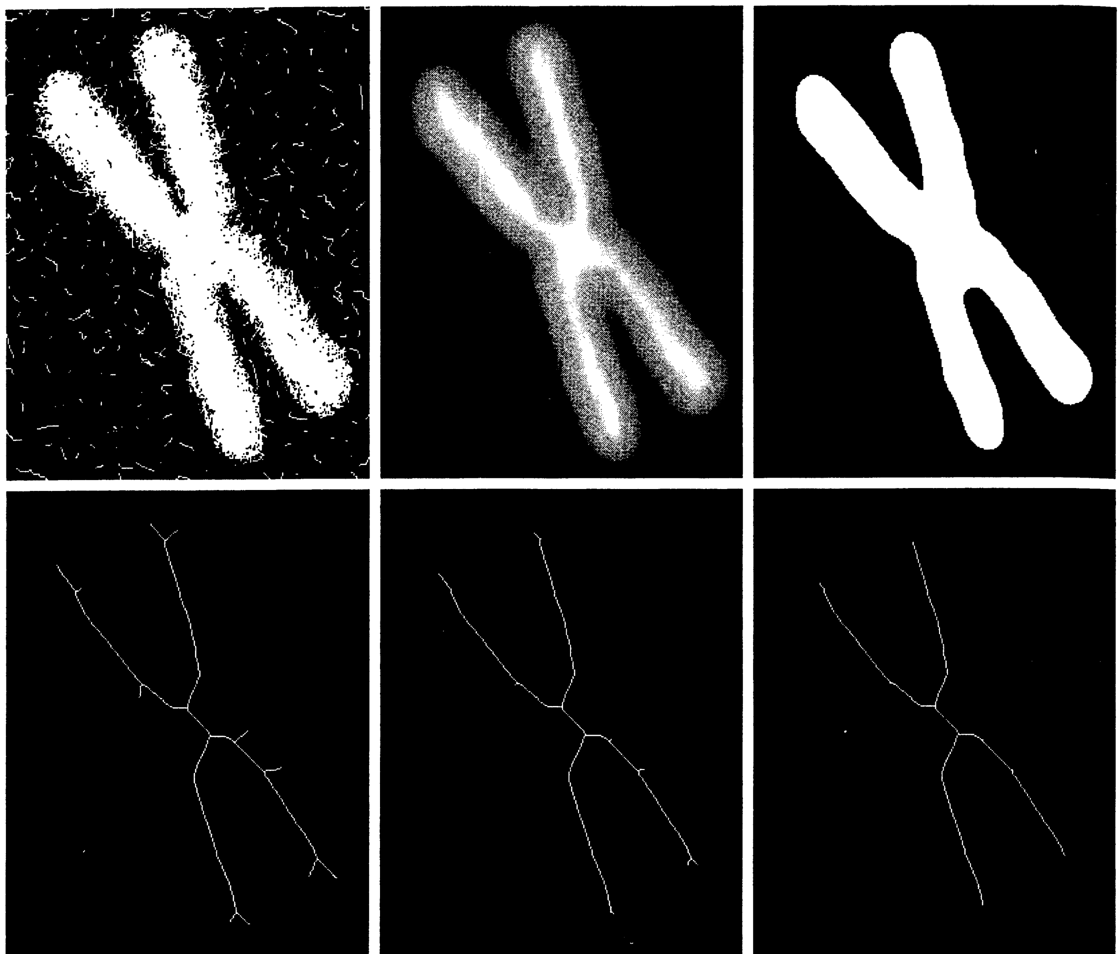
```
S = bwmorph(B, 'skel', Inf)
```

This function removes pixels on the boundaries of objects but does not allow objects to break apart. The pixels remaining make up the image skeleton. This option preserves the Euler number (defined in Table 11.1).

EXAMPLE 11.7: Computing the skeleton of a region.

■ Figure 11.13(a) shows an image, `f`, representative of what a human chromosome looks like after it has been segmented out of an electron microscope image with magnification on the order of 30,000X. The objective of this example is to compute the skeleton of the chromosome.

Clearly, the first step in the process must be to isolate the chromosome from the background of irrelevant detail. One approach is to smooth the image and then threshold it. Figure 11.13(b) shows the result of smoothing `f` with a 25×25 Gaussian spatial mask with `sig = 15`:



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.

```
>> f = im2double(f);
>> h = fspecial('gaussian', 25, 15);
>> g = imfilter(f, h, 'replicate');
>> imshow(g) % Fig. 11.13(b)
```

Next, we threshold the smoothed image:

```
>> g = im2bw(g, 1.5*graythresh(g));
>> figure, imshow(g) % Fig. 11.13(c)
```

where the automatically determined threshold, graythresh(g), was multiplied by 1.5 to increase by 50% the amount of thresholding. The reasoning for

this is that increasing the threshold value increases the amount of data removed from the boundary, thus achieving additional smoothing. The skeleton of Fig. 11.13(d) was obtained using the command

```
>> s = bwmorph(g, 'skel', Inf); % Fig. 11.13(d)
```

The spurs in the skeleton were reduced using the command

```
>> s1 = bwmorph(s, 'spur', 8); % Fig. 11.13(e)
```

where we repeated the operation 8 times, which in this case is equal to the approximately one-half the value of `sig` in the smoothing filter. Several small spurs still remain in the skeleton. However, applying the previous function an additional 7 times (to complete the value of `sig`) yielded the result in Fig. 11.13(f), which is a reasonable skeleton representation of the input. As a rule of thumb, the value of `sig` of a Gaussian smoothing mask is a good guideline for the selection of the number of times a spur removal algorithm is applied. ■

11.3 Boundary Descriptors

In this section we discuss a number of descriptors that are useful when working with region boundaries. As will become evident shortly, many of these descriptors can be used for boundaries and/or regions, and the grouping of these descriptors in IPT does not make a distinction regarding their applicability. Therefore, some of the concepts introduced here are mentioned again in Section 11.4 when we discuss regional descriptors.

11.3.1 Some Simple Descriptors

The *length* of a boundary is one of its simplest descriptors. The length of a 4-connected boundary is simply the number of pixels in the boundary, minus 1. If the boundary is 8-connected, we count vertical and horizontal transitions as 1, and diagonal transitions as $\sqrt{2}$.

We extract the boundary of objects contained in image `f` using function `bwperim`, introduced in Section 11.2.2:

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

where `g` is a binary image containing the boundaries of the objects in `f`. For 2-D connectivity, which is our focus, `conn` can have the values 4 or 8, depending on whether 4- or 8-connectivity (the default) is desired (see the margin note in Example 11.4 concerning the interpretation of these connectivity values). The objects in `f` can have any pixel values consistent with the image class, but all background pixels have to be 0. By definition, the perimeter pixels are nonzero and are connected to at least one other nonzero pixel.

Connectivity can be defined in a more general way in IPT by using a 3×3 matrix of 0s and 1s for `conn`. The 1-valued elements define neighborhood

locations relative to the center element of conn. For example, conn = ones(3) defines 8-connectivity. Array conn must be symmetric about its center element. The input image can be of any class. The output image containing the boundary of each object in the input is of class logical.

The *diameter* of a boundary is defined as the Euclidean distance between the two farthest points on the boundary. These points are not always unique, as in a circle or square, but generally the assumption is that if the diameter is to be a useful descriptor, it is best applied to boundaries with a single pair of farthest points.† The line segment connecting these points is called the *major axis* of the boundary. The *minor axis* of a boundary is defined as the line perpendicular to the major axis and of such length that a box passing through the outer four points of intersection of the boundary with the two axes completely encloses the boundary. This box is called the *basic rectangle*, and the ratio of the major to the minor axis is called the *eccentricity* of the boundary.

Function diameter (see Appendix C for a listing) computes the diameter, major axis, minor axis, and basic rectangle of a boundary or region. Its syntax is

diameter

```
s = diameter(L)
```

where L is a label matrix (Section 9.4) and s is a structure with the following fields:

s.Diameter	A scalar, the maximum distance between any two pixels in the corresponding region.
s.MajorAxis	A 2 × 2 matrix. The rows contain the row and column coordinates for the endpoints of the major axis of the corresponding region.
s.MinorAxis	A 2 × 2 matrix. The rows contain the row and column coordinates for the endpoints of the minor axis of the corresponding region.
s.BasicRectangle	A 4 × 2 matrix. Each row contains the row and column coordinates of a corner of the basic rectangle.

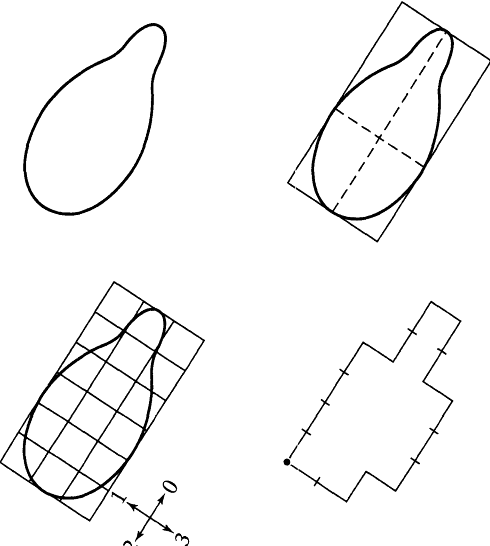
11.3.2 Shape Numbers

The *shape number* of a boundary, generally based on 4-directional Freeman chain codes, is defined as the first difference of smallest magnitude (Bribiesca and Guzman [1980], Bribiesca [1981]). The *order* of a shape number is defined at the number of digits in its representation. Thus, the shape number of a boundary is given by parameter c.diffmm in function fchcode discussed in Section 11.2.1, and the order of the shape number is computed as length(c.diffmm).

As noted in Section 11.2.1, 4-directional Freeman chain codes can be made insensitive to the starting point by using the integer of minimum magnitude, and made insensitive to rotations that are multiples of 90° by using the first

---

†When more than one pair of farthest points exist, they should be near each other and be dominant factors in determining boundary shape.



Chain code: 0 0 0 0 3 0 0 3 2 2 3 2 2 2 1 2 1 1
Difference: 3 0 0 0 3 1 0 3 3 0 1 3 0 0 3 1 3 0
Shape no.: 0 0 0 3 1 0 3 3 0 1 3 0 0 3 1 3 0 3

a b
c d

FIGURE 11.14 Steps in the generation of a shape number.

difference of the code. Thus, shape numbers are insensitive to the starting point and to rotations that are multiples of 90°. An approach used frequently to normalize for arbitrary rotations is to align one of the coordinate axes with the major axis and then extract the 4-code based on the rotated figure. The procedure is illustrated in Fig. 11.14.

The tools required to implement an M-function that calculates shape numbers have been developed already. They consist of function `boundaries` to extract the boundary, function `diameter` to find the major axis, function `bsubsamp` to reduce the resolution of the sampling grid, and function `fchcode` to extract the shape number. Keep in mind when using function `boundaries` to extract 4-connected boundaries that the input image must be labeled using `bwlabel` with 4-connectivity specified. As indicated in Fig. 11.14, compensation for rotation is based on aligning one of the coordinate axes with the major axis. The x-axis can be aligned with the major axis of a region or boundary by using function `x2majoraxis`. The syntax of this function follows; the code is included in Appendix C:

```
[B, theta] = x2majoraxis(A, B)
```

`x2majoraxis`

Here, `A = s.MajorAxis` from function `diameter`, and `B` is an input (binary) image or boundary list. (As before, we assume that a boundary is a connected, closed curve.) On the output, `B` has the same form as the input (i.e., a binary image or a coordinate sequence. Because of possible round-off error, rotations can result in a disconnected boundary sequence, so postprocessing to relink the points (using, for example, `bwmorph`) may be required.

11.3.3 Fourier Descriptors

Figure 11.15 shows a K-point digital boundary in the xy-plane. Starting at an arbitrary point (x_0, y_0), coordinate pairs $(x_0, y_0), (x_1, y_1), (x_2, y_2), \ldots, (x_{K-1}, y_{K-1})$ are encountered in traversing the boundary, say, in the counterclockwise direction. These coordinates can be expressed in the form $x(k) = x_k$ and $y(k) = y_k$. With this notation, the boundary itself can be represented as the sequence of coordinates $s(k) = [x(k), y(k)]$, for $k = 0, 1, 2, \ldots, K - 1$. Moreover, each coordinate pair can be treated as a complex number so that

$$s(k) = x(k) + jy(k)$$

From Section 4.1, the discrete Fourier transform (DFT) of $s(k)$ is

$$a(u) = \sum_{k=0}^{K-1} s(k)e^{-j2\pi uk/K}$$

for $u = 0, 1, 2, \ldots, K - 1$. The complex coefficients $a(u)$ are called the *Fourier descriptors* of the boundary. The inverse Fourier transform of these coefficients restores $s(k)$. That is,

$$s(k) = \frac{1}{K}\sum_{u=0}^{K-1} a(u)e^{j2\pi uk/K}$$

for $k = 0, 1, 2, \ldots, K - 1$. Suppose, however, that instead of all the Fourier coefficients, only the first P coefficients are used. This is equivalent to setting $a(u) = 0$ for $u > P - 1$ in the preceding equation for $a(u)$. The result is the following *approximation* to $s(k)$:

$$\hat{s}(k) = \frac{1}{P}\sum_{u=0}^{P-1} a(u)e^{j2\pi uk/K}$$

for $k = 0, 1, 2, \ldots, K - 1$. Although only P terms are used to obtain each component of $\hat{s}(k)$, k still ranges from 0 to $K - 1$. That is, the *same* number of points exists in the approximate boundary, but not as many terms are used in the reconstruction of each point. Recall from Chapter 4 that high-frequency components account for fine detail, and low-frequency components determine global shape. Thus, loss of detail in the boundary increases as P decreases.

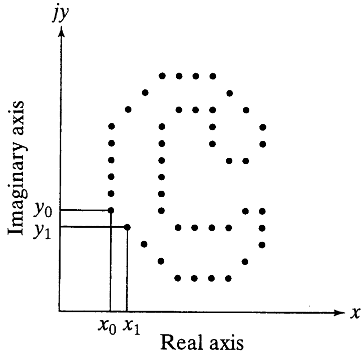


FIGURE 11.15 A digital boundary and its representation as a complex sequence. The points (x_0, y_0) and (x_1, y_1) are (arbitrarily) the first two points in the sequence.

The following function, frdescp, computes the Fourier descriptors of a boundary, s. Similarly, given a set of Fourier descriptors, function ifrdescp computes the inverse using a specified number of descriptor, to yield a closed spatial curve. The documentation section of each function explains its syntax.

frdescp

```
function z = frdescp(s)
%FRDESCP Computes Fourier descriptors.
%   Z = FRDESCP(S) computes the Fourier descriptors of S, which is an
%   np-by-2 sequence of image coordinates describing a boundary.
%
%   Due to symmetry considerations when working with inverse Fourier
%   descriptors based on fewer than np terms, the number of
%   points in S when computing the descriptors must be even. If the
%   number of points is odd, FRDESCP duplicates the end point and
%   adds it at the end of the sequence. If a different treatment is
%   desired, the sequence must be processed externally so that it has
%   an even number of points.
%
%   See function IFRDESCP for computing the inverse descriptors.

% Preliminaries
[np, nc] = size(s);
if nc ~= 2
   error('S must be of size np-by-2.');
end
if np/2 ~= round(np/2);
   s(end + 1, :) = s(end, :);
   np = np + 1;
end

% Create an alternating sequence of 1s and -1s for use in centering
% the transform.
x = 0:(np - 1);
m = ((-1) .^ x)';

% Multiply the input sequence by alternating 1s and -1s to
% center the transform.
s(:, 1) = m .* s(:, 1);
s(:, 2) = m .* s(:, 2);
% Convert coordinates to complex numbers.
s = s(:, 1) + i*s(:, 2);
% Compute the descriptors.
z = fft(s);
```

Function ifrdescp is as follows:

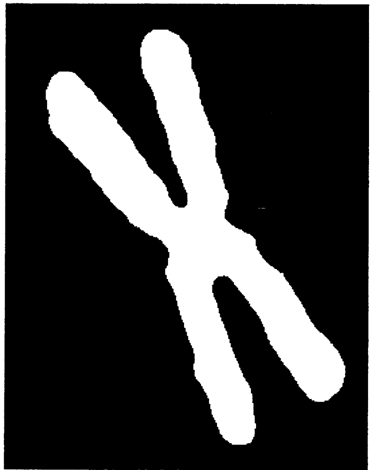
ifrdescp

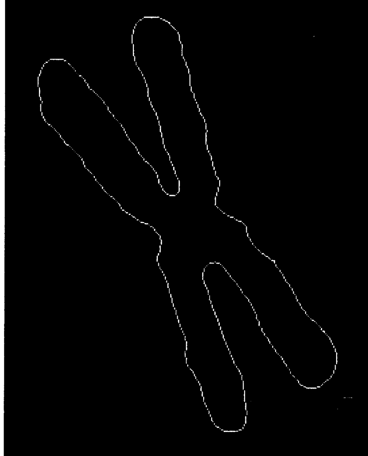
```
function s = ifrdescp(z, nd)
%IFRDESCP Computes inverse Fourier descriptors.
%   I = IFRDESCP(Z, ND) computes the inverse Fourier descriptors of
%   of Z, which is a sequence of Fourier descriptor obtained, for
%   example, by using function FRDESCP. ND is the number of
%   descriptors used to computing the inverse; ND must be an even
```

```
%   integer no greater than length(Z). If ND is omitted, it defaults
%   to length(Z). The output, S, is an ND-by-2 matrix containing the
%   coordinates of a closed boundary.

% Preliminaries.
np = length(z);
% Check inputs.
if nargin == 1 | nd > np
   nd = np;
end

% Create an alternating sequence of 1s and -1s for use in centering
% the transform.
x = 0:(np - 1);
m = ((-1) .^ x)';

% Use only nd descriptors in the inverse. Since the
% descriptors are centered, (np - nd)/2 terms from each end of
% the sequence are set to 0.
d = round((np - nd)/2); % Round in case nd is odd.
z(1:d) = 0;
z(np - d + 1:np) = 0;
% Compute the inverse and convert back to coordinates.
zz = ifft(z);
s(:, 1) = real(zz);
s(:, 2) = imag(zz);
% Multiply by alternating 1 and -1s to undo the earlier
% centering.
s(:, 1) = m.*s(:, 1);
s(:, 2) = m.*s(:, 2);
```

EXAMPLE 11.8: Fourier descriptors.

■ Figure 11.16(a) shows a binary image, `f`, similar to the one in Fig. 11.13(c), but obtained using a Gaussian mask of size 15 × 15 with `sigma = 9`, and thresholded at 0.7. The purpose was to generate an image that was not overly





a b

FIGURE 11.16 (a) Binary image. (b) Boundary extracted using function `boundaries`. The boundary has 1090 points.

smooth in order to illustrate the effect that reducing the number of descriptors has on the shape of a boundary. The image in Fig. 11.16(b) was generated using the commands

```
>> b = boundaries(f);
>> b = b{1};
>> bim = bound2im(b, 344, 270);
```

where the dimensions shown are the dimensions of `f`. Figure 11.16(b) shows image `bim`. The boundary shown has 1090 points. Next, we computed the Fourier descriptors,

```
>> z = frdescp(b);
```

and obtained the inverse using approximately 50% of the possible 1090 descriptors:

```
>> z546 = ifrdescp(z, 546);
>> z546im = bound2im(z546, 344, 270);
```

Image `z546im` [Fig. 11.17(a)] shows close correspondence with the original boundary in Fig. 11.16(b). Some subtle details, like a 1-pixel bay in the bottom-facing cusp in the original boundary, were lost, but, for all practical purposes, the two boundaries are identical. Figures 11.17(b) through (f) show the results obtained using 110, 56, 28, 14, and 8 descriptors, which are approximately 10%, 5%, 2.5%, 1.25% and 0.7%, of the possible 1090 descriptors. The result obtained using 110 descriptors [Fig. 11.17(c)] shows slight further smoothing of the boundary, but, again, the general shape is quite close to the original. Figure 11.17(e) shows that even the result with 14 descriptors, a mere 1.25% of the total, retained the principal features of the boundary. Figure 11.17(f) shows distortion that is unacceptable because the main feature of the boundary (the four long protrusions) was lost. Further reduction to 4 and 2 descriptors would result in an ellipse and, finally, a circle.

Some of the boundaries in Fig. 11.17 have one-pixel gaps due to round off in pixel values. These small gaps, common with Fourier descriptors, can be repaired with function `bwmorph` using the `'bridge'` option. ■

As mentioned earlier, descriptors should be as insensitive as possible to translation, rotation, and scale changes. In cases where results depend on the order in which points are processed, an additional constraint is that descriptors should be insensitive to starting point. Fourier descriptors are not directly insensitive to these geometric changes, but the changes in these parameters can be related to simple transformations on the descriptors (see Gonzalez and Woods [2002]).

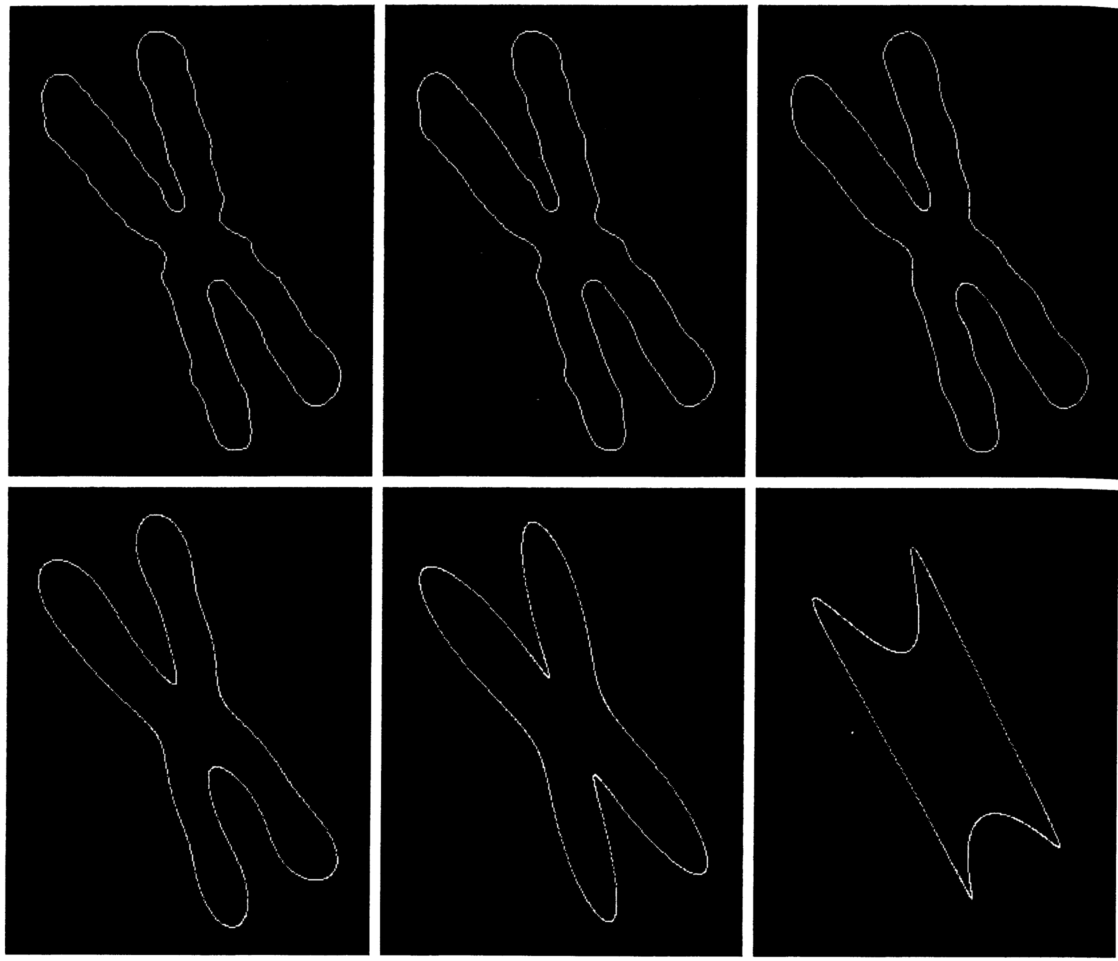
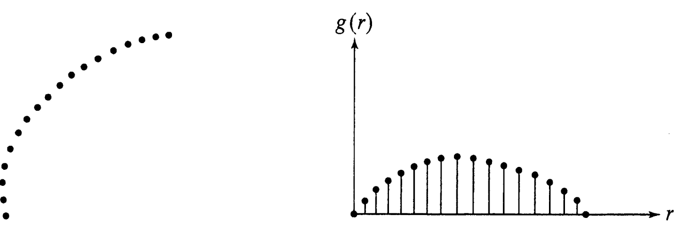


FIGURE 11.17 (a)–(f) Boundary reconstructed using 546, 110, 56, 28, 14, and 8 Fourier descriptors out of a possible 1090 descriptors.

11.3.4 Statistical Moments

The shape of 1-D boundary representations (e.g., boundary segments and signature waveforms) can be described quantitatively by using statistical moments, such as the mean, variance, and higher-order moments. Consider Fig. 11.18(a), which shows a boundary segment, and Fig. 11.18(b), which shows the segment represented as a 1-D function, $g(r)$, of an arbitrary variable r. This function was obtained by connecting the two end points of the segment to form a "major" axis and then using function `x2majoraxis` discussed in Section 11.3.2 to align the major axis with the x-axis.

One approach for describing the shape of $g(r)$ is to normalize it to unit area and treat it as a histogram. In other words, $g(r_i)$ is treated as the probability of value r_i occurring. In this case, r is considered a random variable and the moments are



a b

FIGURE 11.18 (a) Boundary segment. (b) Representation as a 1-D function.

$$\mu_n = \sum_{i=0}^{K-1} (r_i - m)^n g(r_i)$$

where

$$m = \sum_{i=0}^{K-1} r_i g(r_i)$$

In this notation, K is the number of points on the boundary, and $\mu_n(r)$ is directly related to the shape of $g(r)$. For example, the second moment $\mu_2(r)$ measures the spread of the curve about the mean value of r and the third moment, $\mu_3(r)$, measures its symmetry with reference to the mean. Statistical moments are computed with function `statmoments`, discussed in Section 5.2.4.

What we have accomplished is to reduce the description task to 1-D functions. Although moments are a popular approach, they are not the only descriptors that could be used for this purpose. For instance, another method involves computing the 1-D discrete Fourier transform, obtaining its spectrum, and using the first q components of the spectrum to describe $g(r)$. The advantage of moments over other techniques is that implementation of moments is straightforward, and moments also carry a "physical" interpretation of boundary shape. The insensitivity of this approach to rotation is clear from Fig. 11.18. Size normalization, if desired, can be achieved by scaling the range of values of g and r.

11.4 Regional Descriptors

In this section we discuss a number of IPT functions for region processing and introduce several additional functions for computing texture, moment invariants, and several other regional descriptors. Keep in mind that function `bwmorph` discussed in Section 9.3.4 is used frequently for the type of processing we outline in this section. Function `roipoly` (Section 5.2.4) also is used frequently in this context.

11.4.1 Function regionprops

Function `regionprops` is IPT's principal tool for computing region descriptors. This function has the syntax

```
D = regionprops(L, properties)
```

where L is a label matrix and D is a structure array of length max(L(:)). The fields of the structure denote different measurements for each region, as specified by properties. Argument properties can be a comma-separated list of strings, a cell array containing strings, the single string 'all', or the string 'basic'. Table 11.1 lists the set of valid property strings. If properties is the string 'all', then all the descriptors in Table 11.1 are computed. If properties is not specified or if it is the string 'basic', then the descriptors computed are 'Area', 'Centroid', and 'BoundingBox'. Keep in mind (as discussed in Section 2.1.1) that IPT uses x and y to indicate horizontal and vertical coordinates, respectively, with the origin being located in the top, left. Coordinates x and y increase to the right and downward from the origin, respectively. For the purposes of our discussion, on pixels are valued 1 while off pixels are valued 0.

EXAMPLE 11.9: Using function regionprops.

■ As a simple illustration, suppose that we want to obtain the area and the bounding box for each region in an image B. We write

```
>> B = bwlabel(B); % Convert B to a label matrix.
>> D = regionprops(B, 'area', 'boundingbox');
```

To extract the areas and number of regions we write

```
>> w = [D.Area];
>> NR = length(w);
```

where the elements of vector w are the areas of the regions and NR is the number of regions. Similarly, we can obtain a single matrix whose rows are the bounding boxes of each region using the statement

```
V = cat(1, D.BoundingBox);
```

This array is of dimension NR $\times$ 4. The cat operator is explained in Section 6.1.1. ■

11.4.2 Texture

An important approach for describing a region is to quantify its texture content. In this section we illustrate the use of two new functions for computing texture based on statistical and spectral measures.

Statistical Approaches

A frequently used approach for texture analysis is based on statistical properties of the intensity histogram. One class of such measures is based on statistical moments. As discussed in Section 5.2.4, the expression for the nth moment about the mean is given by

$$\mu_n = \sum_{i=0}^{L-1} (z_i - m)^n p(z_i)$$

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].

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.

where z_i is a random variable indicating intensity, $p(z)$ is the histogram of the intensity levels in a region, L is the number of possible intensity levels, and

$$m = \sum_{i=0}^{L-1} z_i p(z_i)$$

is the mean (average) intensity. These moments can be computed with function `statmoments` discussed in Section 5.2.4. Table 11.2 lists some common descriptors based on statistical moments and also on uniformity and entropy. Keep in mind that the second moment, $\mu_2(z)$, is the variance, σ^2.

Writing an M-function to compute the texture measures in Table 11.3 is straightforward. Function `statxture`, written for this purpose, is included in Appendix C. The syntax of this function is

TABLE 11.3 Texture measures for the regions shown in Fig. 11.19.

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

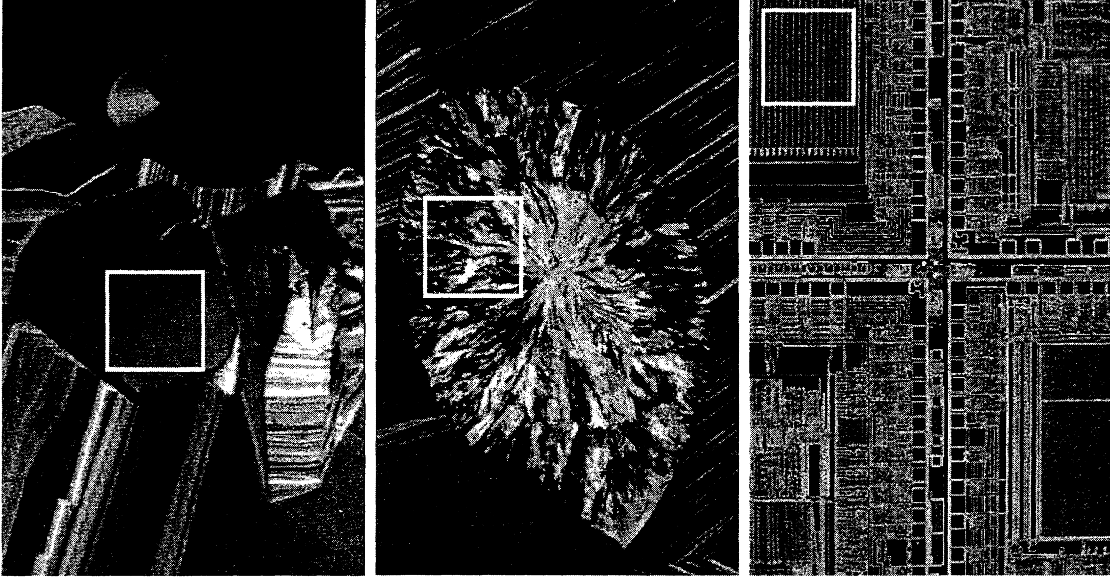
```
t = statxture(f, scale)
```

statxture

where `f` is an input image (or subimage) and `t` is a 6-element row vector whose components are the descriptors in Table 11.2, arranged in the same order. Parameter `scale` also is a 6-element row vector, whose components multiply the corresponding elements of `t` for scaling purposes. If omitted, `scale` defaults to all 1s.

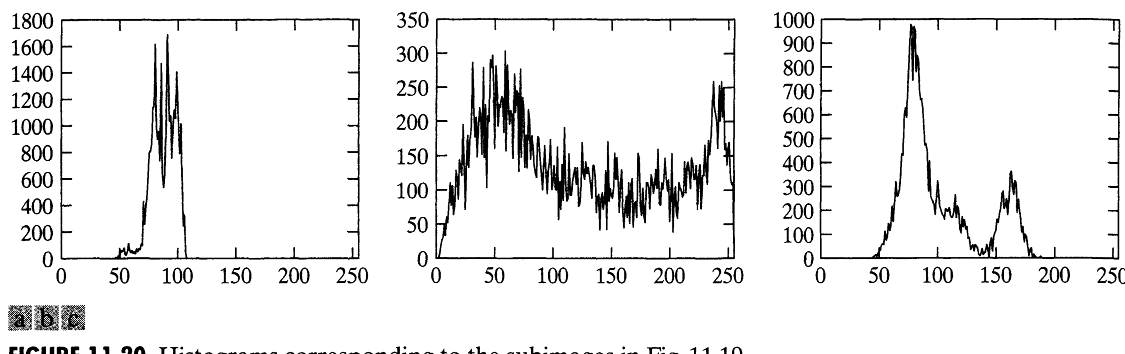
EXAMPLE 11.10: Statistical texture measures.

■ The three regions outlined by the white boxes in Fig. 11.19 represent, from left to right, examples of smooth, coarse and periodic texture. The histograms of these regions, obtained using function `imhist`, are shown in Fig. 11.20. The entries in Table 11.3 were obtained by applying function `statxture` to each of the subimages in Fig. 11.19. These results are in general agreement with the texture content of the respective subimages. For example, the entropy of the coarse region [Fig. 11.19(b)] is higher than the others because the values of the pixels in that region are more random than the values in the other



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.)



a b c

FIGURE 11.20 Histograms corresponding to the subimages in Fig. 11.19.

regions. This also is true for the contrast and for the average intensity in this case. On the other hand, this region is the least smooth and the least uniform, as revealed by the values of R and the uniformity measure. The histogram of the coarse region also shows the greatest lack of symmetry with respect to the location of the mean value, as is evident in Fig. 11.20(b), and also by the largest value of the third moment shown in Table 11.3. ■

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. These global texture patterns, easily distinguishable as concentrations of high-energy bursts in the spectrum, generally are quite difficult to detect with spatial methods because of the local nature of these techniques. Thus spectral texture is useful for discriminating between periodic and nonperiodic texture patterns, and, further, for quantifying differences between periodic patterns.

Interpretation of spectrum features is simplified by expressing the spectrum in polar coordinates to yield a function $S(r, \theta)$, where S is the spectrum function and r and θ are the variables in this coordinate system. For each direction θ, $S(r, \theta)$ may be considered a 1-D function, $S_\theta(r)$. Similarly, for each frequency r, $S_r(\theta)$ is a 1-D function. Analyzing $S_\theta(r)$ for a fixed value of θ yields the behavior of the spectrum (such as the presence of peaks) along a radial direction from the origin, whereas analyzing $S_r(\theta)$ for a fixed value of r yields the behavior along a circle centered on the origin.

A global description is obtained by integrating (summing for discrete variables) these functions:

$$S(r) = \sum_{\theta=0}^{\pi} S_\theta(r)$$

and

$$S(\theta) = \sum_{r=1}^{R_0} S_r(\theta)$$

where R_0 is the radius of a circle centered at the origin.

The results of these two equations constitute a pair of values $[S(r), S(\theta)]$ for *each* pair of coordinates (r, θ). By varying these coordinates we can generate two 1-D functions, $S(r)$ and $S(\theta)$, that constitute a spectral-energy description of texture for an entire image or region under consideration. Furthermore, descriptors of these functions themselves can be computed in order to characterize their behavior quantitatively. Descriptors typically used for this purpose are the location of the highest value, the mean and variance of both the amplitude and axial variations, and the distance between the mean and the highest value of the function.

Function `specxture` (see Appendix C for a listing) can be used to compute the two preceding texture measures. The syntax is

```
[srad, sang, S] = specxture(f)
```

`specxture`

where `srad` is $S(r)$, `sang` is $S(\theta)$, and S is the spectrum image (displayed using the log, as explained in Chapter 4).

EXAMPLE 11.11: Computing spectral texture.

■ Figure 11.21(a) shows an image with randomly distributed objects and Fig. 11.22(b) shows an image containing the same objects, but arranged periodically. The corresponding Fourier spectra, computed using function `specxture`, are shown in Figs. 11.21(c) and (d). The periodic bursts of energy extending quadrilaterally in two dimensions in the Fourier spectra are due to the periodic texture of the coarse background material on which the matches rest. The other components of the spectra in Fig. 11.21(c) are clearly caused by the random orientation of the strong edges in Fig. 11.21(a). By contrast, the main energy in Fig. 11.21(d) not associated with the background is along the horizontal axis, corresponding to the strong vertical edges in Fig. 11.21(b).

Figures 11.22(a) and (b) are plots of $S(r)$ and $S(\theta)$ for the random matches, and similarly in (c) and (d) for the ordered matches, all computed with function `specxture`. The plots were obtained with the commands `plot(srad)` and `plot(sang)`. The axes in Figs. 11.22(a) and (c) were scaled using

```
>> axis([horzmin horzmax vertmin vertmax])
```

discussed in Section 3.3.1, with the maximum and minimum values obtained from Fig. 11.22(a).

The plot of $S(r)$ corresponding to the randomly-arranged matches shows no strong periodic components (i.e., there are no peaks in the spectrum besides the peak at the origin, which is the DC component). On the other hand, the plot of $S(r)$ corresponding to the ordered matches shows a strong peak near $r = 15$ and a smaller one near $r = 25$. Similarly, the random nature of the energy bursts in Fig. 11.21(c) is quite apparent in the plot of $S(\theta)$ in Fig. 11.22(b). By contrast, the plot in Fig. 11.22(d) shows strong energy components in the region near the origin and at 90° and 180°. This is consistent with the energy distribution in Fig. 11.21(d). ■

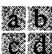


FIGURE 11.21 (a) and (b) Images of unordered and ordered objects. (c) and (d) Corresponding spectra.

11.4.3 Moment Invariants

The 2-D *moment* of order $(p + q)$ of a digital image $f(x, y)$ is defined as

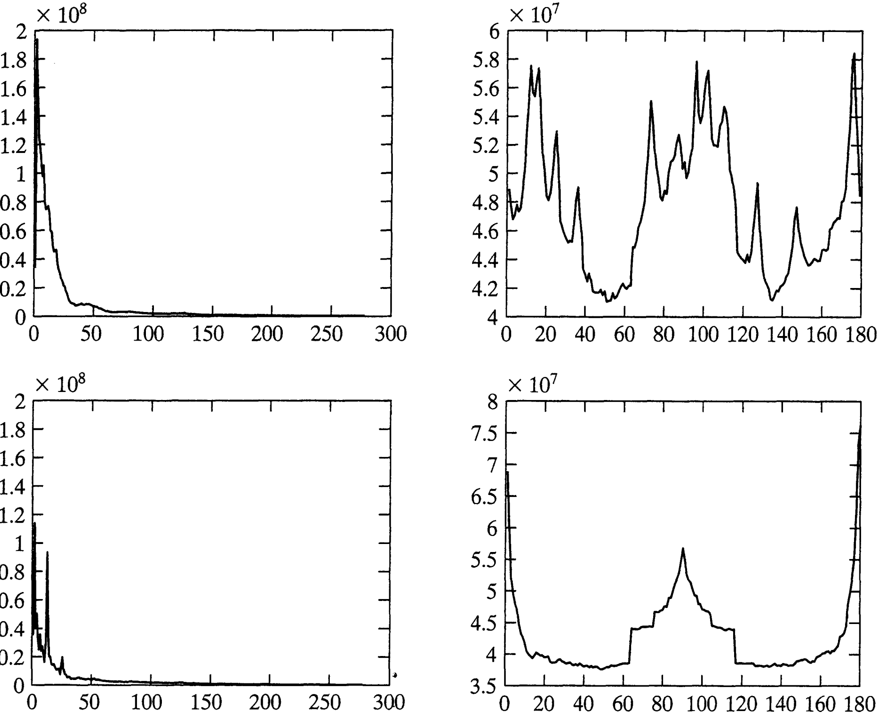
$$m_{pq} = \sum_x \sum_y x^p y^q f(x, y)$$

for $p, q = 0, 1, 2, \ldots$, where the summations are over the values of the spatial coordinates x and y spanning the image. The corresponding *central moment* is defined as

$$\mu_{pq} = \sum_x \sum_y (x - \bar{x})^p (y - \bar{y})^q f(x, y)$$

where

$$\bar{x} = \frac{m_{10}}{m_{00}} \quad \text{and} \quad \bar{y} = \frac{m_{01}}{m_{00}}$$



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.

The *normalized central moment* of order $(p + q)$ is defined as

$$\eta_{pq} = \frac{\mu_{pq}}{\mu_{00}^{\gamma}}$$

for $p, q = 0, 1, 2, \ldots$, where

$$\gamma = \frac{p + q}{2} + 1$$

for $p + q = 2, 3, \ldots$.

A set of seven 2-D *moment invariants* that are insensitive to translation, scale change, mirroring, and rotation can be derived from these equations. They are

$$\phi_1 = \eta_{20} + \eta_{02}$$

$$\phi_2 = (\eta_{20} - \eta_{02})^2 + 4\eta_{11}^2$$

$$\phi_3 = (\eta_{30} - 3\eta_{12})^2 + (3\eta_{21} - \eta_{03})^2$$

$$\phi_4 = (\eta_{30} + \eta_{12})^2 + (\eta_{21} + \eta_{03})^2$$

$$\begin{aligned}\phi_5 = {} & (\eta_{30} - 3\eta_{12})(\eta_{30} + \eta_{12})[(\eta_{30} + \eta_{12})^2 \\ & -3(\eta_{21} + \eta_{03})^2] + (3\eta_{21} - \eta_{03})(\eta_{21} + \eta_{03}) \\ & [3(\eta_{30} + \eta_{12})^2 - (\eta_{21} + \eta_{03})^2]\end{aligned}$$

$$\begin{aligned}\phi_6 = {} & (\eta_{20} - \eta_{02})[(\eta_{30} + \eta_{12})^2 - (\eta_{21} + \eta_{03})^2] \\ & + 4\eta_{11}(\eta_{30} + \eta_{12})(\eta_{21} + \eta_{03})\end{aligned}$$

$$\phi_7 = (3\eta_{21} - \eta_{03})(\eta_{30} + \eta_{12})[(\eta_{30} + \eta_{12})^2 - 3(\eta_{21} + \eta_{03})^2] + (3\eta_{12} - \eta_{30})(\eta_{21} + \eta_{03}) [3(\eta_{30} + \eta_{12})^2 - (\eta_{21} + \eta_{03})^2]$$

An M-function for computing the moment invariants, which we call `invmoments`, is a direct implementation of these seven equations. The syntax is as follows (see Appendix C for the code listing):

invmoments

```
phi = invmoments(f)
```

where `f` is the input image and `phi` is a seven-element row vector containing the moment invariants just defined.

EXAMPLE 11.12: Moment invariants.

■ The image in Fig. 11.23(a) was obtained from an original of size 400 × 400 pixels by using the command

```
>> fp = padarray(f, [84 84], 'both');
```

Zero padding was used to make all displayed images consistent with the image occupying the largest area (568 × 568 pixels) which, as discussed below, is the image rotated by 45°. The padding is for display purposes only, and was not used in any moment computations. The half-size and corresponding padded images were obtained using the commands

`B = fliplr(A)` *returns* `A` *with the columns flipped about the vertical axis, and* `B = flipud(A)` *returns* `A` *with the rows flipped about the horizontal axis.*

```
>> fhs = f(1:2:end, 1:2:end);
>> fhsp = padarray(fhs, [184 184], 'both');
```

The mirrored image was obtained using MATLAB function `fliplr`:

```
>> fm = fliplr(f);
>> fmp = padarray(fm, [84 84], 'both');
```

To rotate an image we use function `imrotate`:

```
g = imrotate(f, angle, method, 'crop')
```

which rotates `f` by `angle` degrees in the counterclockwise direction. Parameter `method` can be one of the following:

`'nearest'` uses nearest neighbor interpolation;
`'bilinear'` uses bilinear interpolation (typically a good choice); and
`'bicubic'` uses bicubic interpolation.

The image size is increased automatically by padding to fit the rotation. If `'crop'` is included in the argument, the central part of the rotated image is cropped to the same size as the original. The default is to specify `angle` only, in which case `'nearest'` interpolation is used and no cropping takes place.

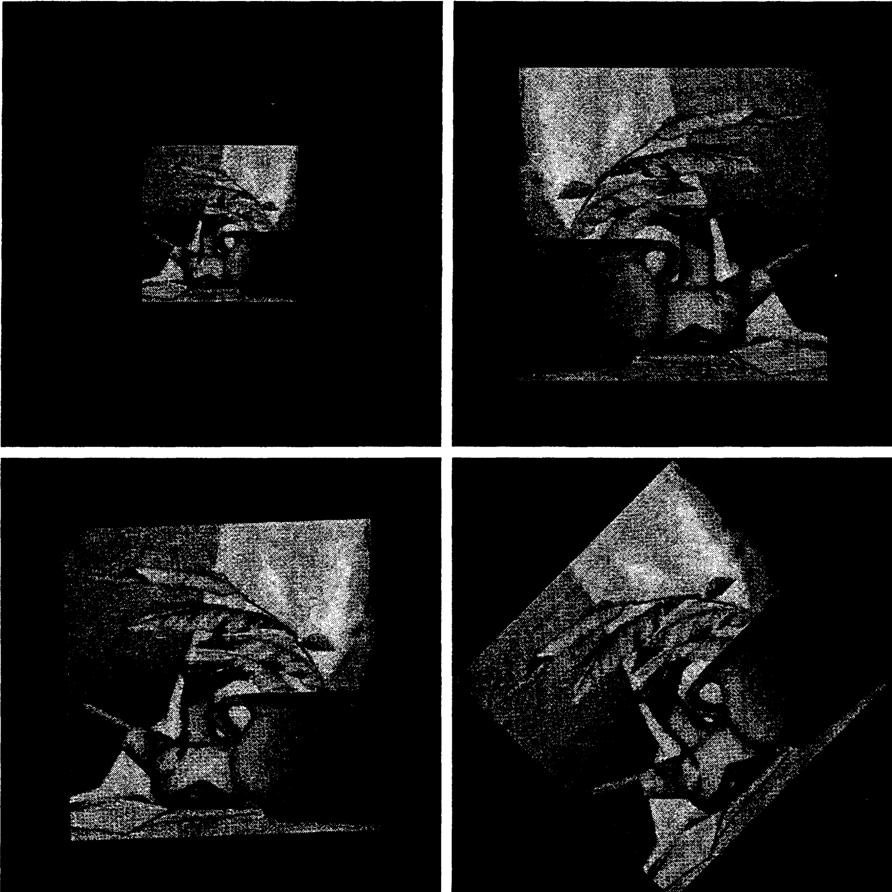


FIGURE 11.23 (a) Original, padded image. (b) Half size image. (c) Mirrored image. (d) Image rotated by 2°. (e) Image rotated 45°. The zero padding in (a) through (d) was done to make the images consistent in size with (e) for viewing purposes only.

TABLE 11.4 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.

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

The rotated images for our example were generated as follows:

```
>> fr2 = imrotate(f, 2, 'bilinear');
>> fr2p = padarray(fr2, [76 76], 'both');
>> fr45 = imrotate(f, 45, 'bilinear');
```

Note that no padding was required in the last image because it is the largest image in the set. The 0s in both rotated images were generated by IPT in the process of rotation.

The seven moment invariants of the five images just discussed were generated using the commands

```
>> 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)));
```

Note that the absolute value of the `log` was used instead of the moment invariant values themselves. Use of the `log` reduces dynamic range, and the absolute value avoids having to deal with the complex numbers that result when computing the `log` of negative moment invariants. Because interest generally lies on the invariance of the moments, and not on their sign, use of the absolute value is common practice.

The seven moments of the original, half-size, mirrored, and rotated images are summarized in Table 11.4. Note how close the numbers are, indicating a high degree of invariance to the changes just mentioned. Results like these are the reason why moment invariants have been a basic staple in image description for more than four decades. ■

11.5 Using Principal Components for Description

Suppose that we have n registered images, "stacked" in the arrangement shown in Fig. 11.24. There are n pixels for any given pair of coordinates (i, j), one pixel at that location for each image. These pixels may be arranged in the form of a column vector

$$\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}$$

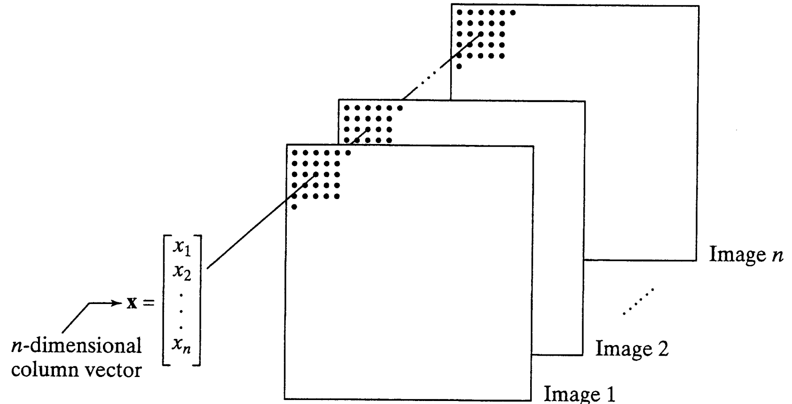


FIGURE 11.24 Forming a vector from corresponding pixels in a stack of images of the same size.

If the images are of size $M \times N$, there will be total of MN such n-dimensional vectors comprising all pixels in the n images.

The mean vector, $\mathbf{m_x}$, of a vector population can be approximated by the sample average:

$$\mathbf{m_x} = \frac{1}{K}\sum_{k=1}^{K}\mathbf{x}_k$$

with $K = MN$. Similarly, the $n \times n$ covariance matrix, $\mathbf{C_x}$, of the population can be approximated by

$$\mathbf{C_x} = \frac{1}{K-1}\sum_{k=1}^{K}(\mathbf{x}_k - \mathbf{m_x})(\mathbf{x}_k - \mathbf{m_x})^T$$

where $K - 1$ instead of K is used to obtain an unbiased estimate of $\mathbf{C_x}$ from the samples. Because $\mathbf{C_x}$ is real and symmetric, finding a set of n orthonormal eigenvectors always is possible.

The *principal components transform* (also called the *Hotelling transform*) is given by

$$\mathbf{y} = \mathbf{A}(\mathbf{x} - \mathbf{m_x})$$

It is not difficult to show that the elements of vector $\mathbf{y}$ are uncorrelated. Thus, the covariance matrix $\mathbf{C_y}$ is diagonal. The rows of matrix $\mathbf{A}$ are the normalized eigenvectors of $\mathbf{C_x}$. Because $\mathbf{C_x}$ is real and symmetric, these vectors form an orthonormal set, and it follows that the elements along the main diagonal of $\mathbf{C_y}$ are the eigenvalues of $\mathbf{C_x}$. The main diagonal element in the ith row of $\mathbf{C_y}$ is the variance of vector element y_i.

Because the rows of $\mathbf{A}$ are orthonormal, its inverse equals its transpose. Thus, we can recover the $\mathbf{x}$'s by performing the inverse transformation

$$\mathbf{x} = \mathbf{A}^T\mathbf{y} + \mathbf{m_x}$$

The importance of the principal components transform becomes evident when only q eigenvectors are used, in which case $\mathbf{A}$ becomes a $q \times n$ matrix, $\mathbf{A}_q$. Now the reconstruction is an approximation:

$$\hat{\mathbf{x}} = \mathbf{A}_q^T\mathbf{y} + \mathbf{m_x}$$

The mean square error between the exact and approximate reconstruction of the $\mathbf{x}$'s is given by the expression

$$e_{ms} = \sum_{j=1}^{n} \lambda_j - \sum_{j=1}^{q} \lambda_j = \sum_{j=q+1}^{n} \lambda_j$$

The first line of this equation indicates that the error is zero if $q = n$ (that is, if all the eigenvectors are used in the inverse transformation). This equation also shows that the error can be minimized by selecting for $\mathbf{A}_q$ the q eigenvectors associated with the largest eigenvalues. Thus, the principal components transform is optimal in the sense that it minimizes the mean square error between the vectors $\mathbf{x}$ and their approximations $\hat{\mathbf{x}}$. The transform owes its name to using the eigenvectors corresponding to the largest (principal) eigenvalues of the covariance matrix. The example given later in this section further clarifies this concept.

A set of n registered images (each of size $M \times N$) is converted to a stack of the form shown in Fig. 11.24 by using the command:

```
>> S = cat(3, f1, f2,..., fn);
```

This image stack array, which is of size $M \times N \times n$, is converted to an array whose *rows* are n-dimensional vectors by using function `imstack2vectors` (see Appendix C for the code), which has the syntax

imstack2vectors

```
[X, R] = imstack2vectors(S, MASK)
```

where `S` is the image stack and `X` is the array of vectors extracted from `S` using the approach shown in Fig. 11.24. Input `MASK` is an $M \times N$ logical or numeric image with nonzero elements in the locations where elements of `S` are to be used in forming `X` and 0s in locations to be ignored. For example, if we wanted to use only vectors in the right, upper quadrant of the images in the stack, then `MASK` would contain 1s in that quadrant and 0s elsewhere. If `MASK` is not included in the argument, then all image locations are used in forming `X`. Finally, parameter `R` is an array whose rows are the 2-D coordinates corresponding to the location of the vectors used to form `X`. We show how to use `MASK` in Example 12.2. In the present discussion we use the default.

The following M-function, `covmatrix`, computes the mean vector and covariance matrix of the vectors in `X`.

covmatrix

```
function [C, m] = covmatrix(X)
%COVMATRIX Computes the covariance matrix of a vector population.
%   [C, M] = COVMATRIX(X) computes the covariance matrix C and the
%   mean vector M of a vector population organized as the rows of
%   matrix X. C is of size N-by-N and M is of size N-by-1, where N is
%   the dimension of the vectors (the number of columns of X).

[K, n] = size(X);
X = double(X);
```

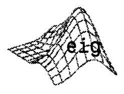
```
if n == 1 % Handle special case.
   C = 0;
   m = X;
else
   % Compute an unbiased estimate of m.
   m = sum(X, 1)/K;
   % Subtract the mean from each row of X.
   X = X - m(ones(K, 1), :);
   % Compute an unbiased estimate of C. Note that the product is
   % X'*X because the vectors are rows of X.
   C = (X'*X)/(K - 1);
   m = m'; % Convert to a column vector.
end
```

The following function implements the concepts developed in this section. Note the use of structures to simplify the output arguments.

princomp

```
function P = princomp(X, q)
%PRINCOMP Obtain principal-component vectors and related quantities.
%   P = PRINCOMP(X, Q) Computes the principal-component vectors of
%   the vector population contained in the rows of X, a matrix of
%   size K-by-n where K is the number of vectors and n is their
%   dimensionality. Q, with values in the range [0, n], is the number
%   of eigenvectors used in constructing the principal-components
%   transformation matrix. P is a structure with the following
%   fields:
%
%     P.Y      K-by-Q matrix whose columns are the principal-
%              component vectors.
%     P.A      Q-by-n principal components transformation matrix
%              whose rows are the Q eigenvectors of Cx corresponding
%              to the Q largest eigenvalues.
%     P.X      K-by-n matrix whose rows are the vectors reconstructed
%              from the principal-component vectors. P.X and P.Y are
%              identical if Q = n.
%    P.ems     The mean square error incurred in using only the Q
%              eigenvectors corresponding to the largest
%              eigenvalues. P.ems is 0 if Q = n.
%    P.Cx      The n-by-n covariance matrix of the population in X.
%    P.mx      The n-by-1 mean vector of the population in X.
%    P.Cy      The Q-by-Q covariance matrix of the population in
%              Y. The main diagonal contains the eigenvalues (in
%              descending order) corresponding to the Q eigenvectors.

[K, n] = size(X);
X = double(X);

% Obtain the mean vector and covariance matrix of the vectors in X.
[P.Cx, P.mx] = covmatrix(X);
P.mx = P.mx'; % Convert mean vector to a row vector.

% Obtain the eigenvectors and corresponding eigenvalues of Cx. The
% eigenvectors are the columns of n-by-n matrix V. D is an n-by-n
```

```
% diagonal matrix whose elements along the main diagonal are the
% eigenvalues corresponding to the eigenvectors in V, so that X*V =
% D*V.
[V, D] = eig(P.Cx);

% Sort the eigenvalues in decreasing order. Rearrange the
% eigenvectors to match.
d = diag(D);
[d, idx] = sort(d);
d = flipud(d);
idx = flipud(idx);
D = diag(d);
V = V(:, idx);

% Now form the q rows of A from first q columns of V.
P.A = V(:, 1:q)';

% Compute the principal component vectors.
Mx = repmat(P.mx, K, 1); % M-by-n matrix. Each row = P.mx.
P.Y = P.A*(X - Mx)'; % q-by-K matrix.

% Obtain the reconstructed vectors.
P.X = (P.A'*P.Y)' + Mx;

% Convert P.Y to K-by-q array and P.mx to n-by-1 vector.
P.Y = P.Y';
P.mx = P.mx';

% The mean square error is given by the sum of all the
% eigenvalues minus the sum of the q largest eigenvalues.
d = diag(D);
P.ems = sum(d(q + 1:end));

% Covariance matrix of the Y's:
P.Cy = P.A*P.Cx*P.A';
```



`[V, D] = eig(A)` *returns the eigenvectors of* `A` *as the columns of matrix* `V`*, and the corresponding eigenvalues along the main diagonal of diagonal matrix* `D`.

EXAMPLE 11.13: Principal components.

■ Figure 11.25 shows six satellite images of size 512 × 512, corresponding to six spectral bands: visible blue (450–520 nm), visible green (520–600 nm), visible red (630–690 nm), near infrared (760–900 nm), middle infrared (1550–1750 nm), and thermal infrared (10,400–12,500 nm). The objective of this example is to illustrate the use of function `princomp` for principal-components work. The first step is to organize the elements of the six images in a stack of size 512 × 512 × 6, as discussed earlier:

```
>> S = cat(3, f1, f2, f3, f4, f5, f6);
```

where the `f`'s correspond to the six multispectral images just discussed. Then we organize the stack into array `X`:

```
>> [X, R] = imstack2vectors(S);
```

Next, we obtain the six principal-component images by using `q = 6` in function `princomp`:

```
>> P = princomp(X, 6);
```

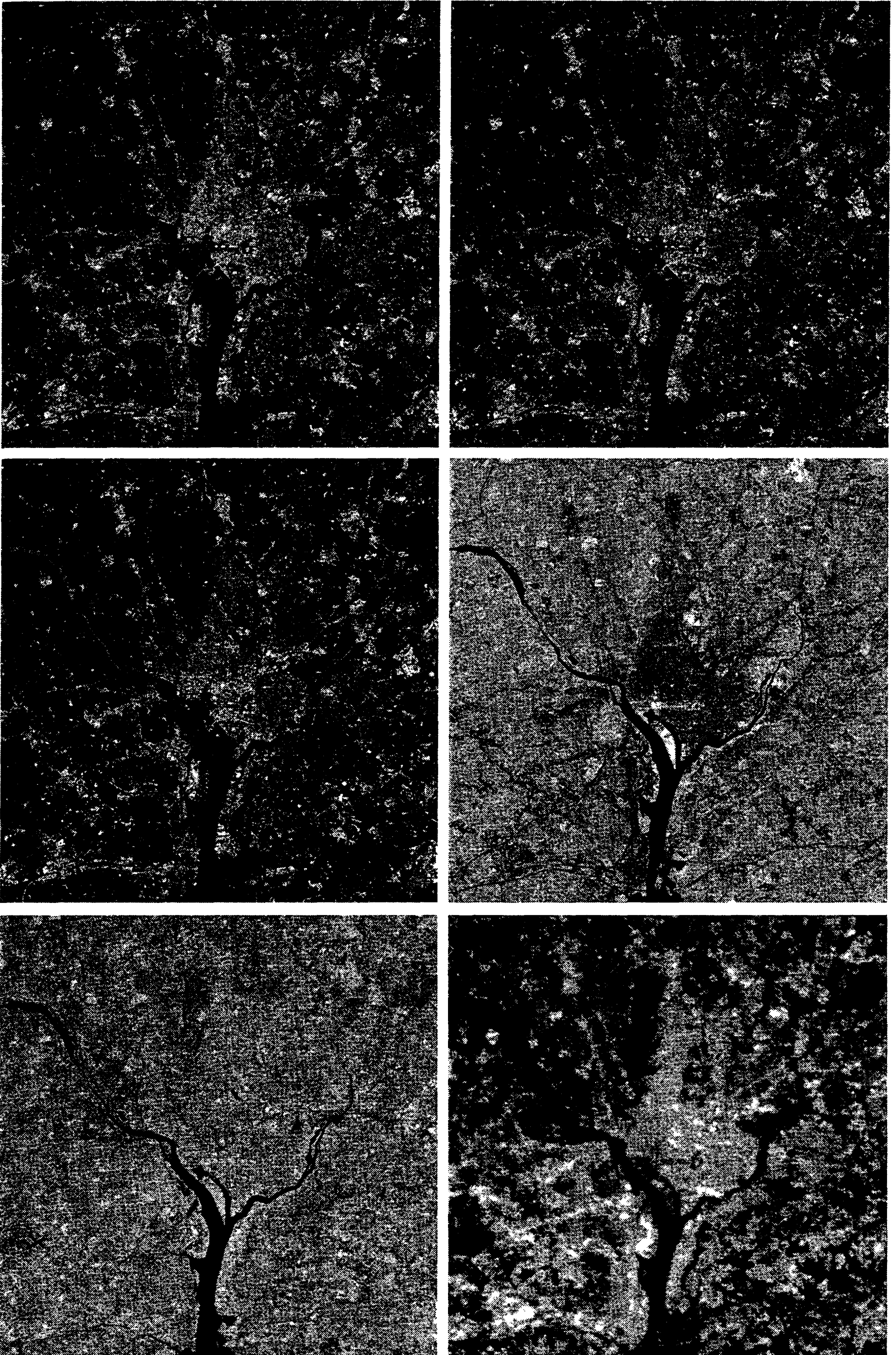


FIGURE 11.25 Six multispectral images in the (a) visible blue, (b) visible green, (c) visible red, (d) near infrared, (e) middle infrared, and (f) thermal infrared bands. (Images courtesy of NASA.)

The first component image is generated and displayed with the commands

```
>> g1 = P.Y(:, 1);
>> g1 = reshape(g1, 512, 512);
>> imshow(g1, [ ])
```

The other five images are obtained and displayed in the same manner. The eigenvalues are along the main diagonal of `P.Cy`, so we use

```
>> d = diag(P.Cy);
```

where `d` is a 6-dimensional column vector because we used `q = 6` in the function.

Figure 11.26 shows the six principal-component images just computed. The most obvious feature is that a significant portion of the contrast detail is contained in the first two images, and it decreases rapidly from there. The reason is easily explained by looking at the eigenvalues. As Table 11.5 shows, the first two eigenvalues are quite large in comparison with the others. Because the eigenvalues are the variances of the elements of the **y** vectors, and variance is a measure of contrast, it is not unexpected that the images corresponding to the dominant eigenvalues would exhibit significantly higher contrast.

Suppose that we use a smaller value of `q`, say `q = 2`. Then reconstruction is based only on two principal component images. Using

```
>> P = princomp(X, 2);
```

and statements of the form

```
>> h1 = P.X(:, 1);
>> h1 = reshape(h1, 512, 512);
```

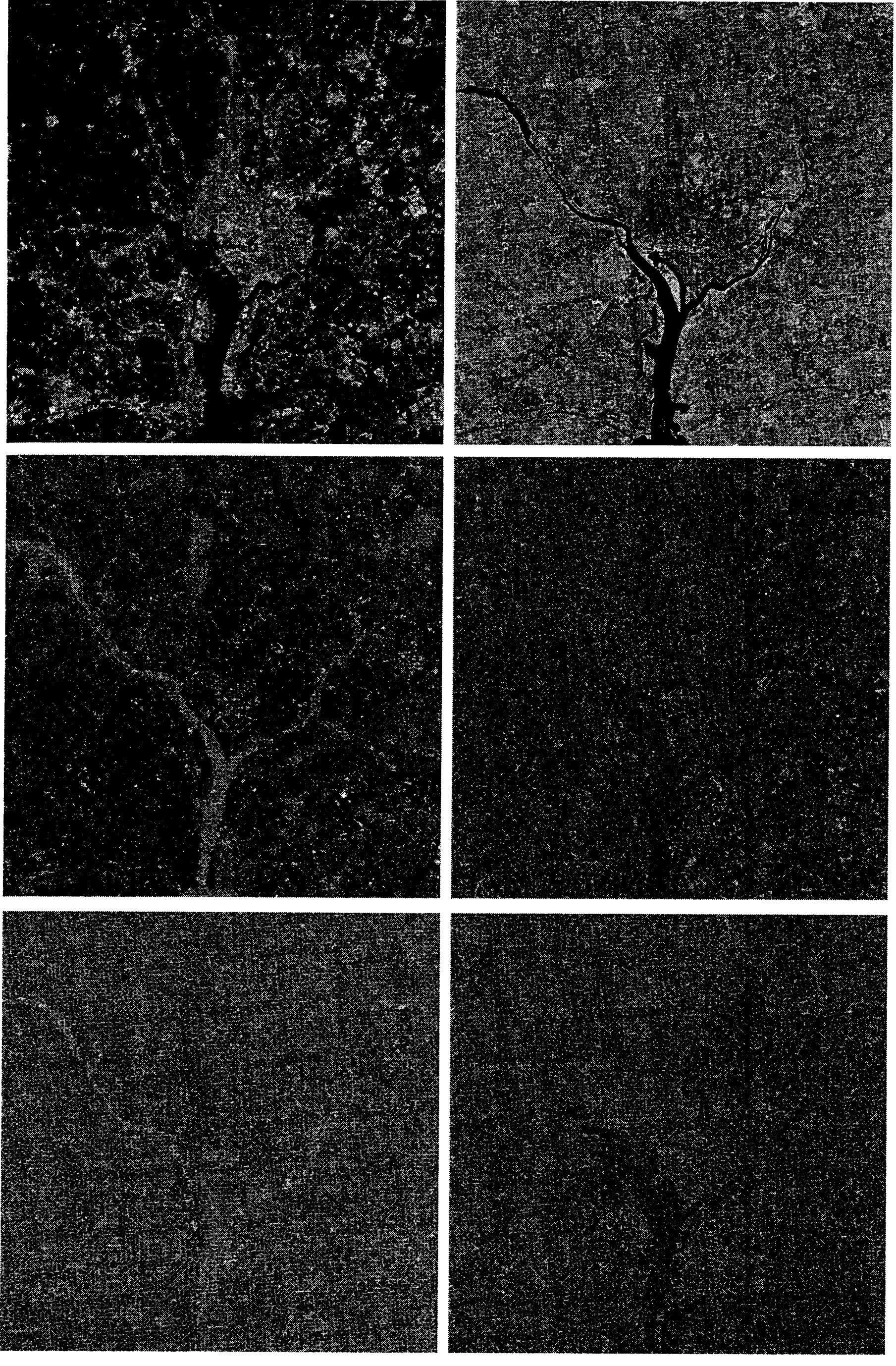
for each image resulted in the reconstructed images in Fig. 11.27. Visually, these images are quite close to the originals in Fig. 11.25. In fact, even the difference images show little degradation. For instance, to compare the original and reconstructed band 1 images, we write

```
>> D1 = double(f1) - double(h1);
>> D1 = gscale(D1);
>> imshow(D1)
```

Figure 11.28(a) shows the result. The low contrast in this image is an indication that little visual data was lost when only two principal component images were used to reconstruct the original image. Figure 11.28(b) shows the difference of the band 6 images. The difference here is more pronounced because the original band 6 image is actually blurry. But the two principal-component images used in the reconstruction are sharp, and they have the strongest influence on the reconstruction. The mean square error incurred in using only two principal component images is given by

```
P.ems

ans =

    1.7311e+003
```

which is the sum of the four smaller eigenvalues in Table 11.5. ■



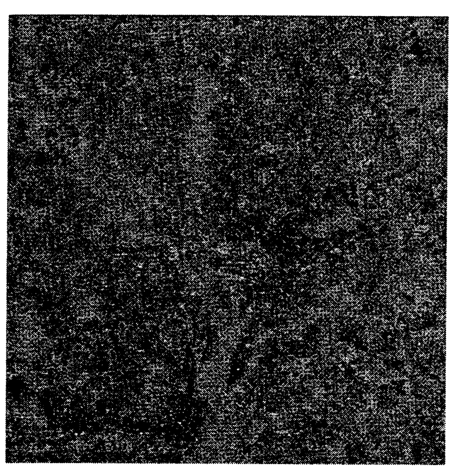
a	b
c	d
e	f

FIGURE 11.26 Principal-component images corresponding to the images in Fig. 11.25.

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`.

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.



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.

Before leaving this section we point out that function `princomp` can be used to align objects (regions or boundaries) with the eigenvectors of the objects. The coordinates of the objects are arranged as the columns of `X`, and we use `q = 2`. The transformed data, aligned with the eigenvectors, is contained in `P.Y`. This is a rugged alignment procedure that uses all coordinates to compute the transformation matrix and aligns the data in the direction of its principal spread.

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. For example, descriptors such as the ones just covered constitute the input to the object recognition algorithms developed in the next chapter. The M-functions developed in the preceding sections of this chapter are a significant extension to the power of standard IPT functions for image representation and description. It is undoubtedly clear by now that the choice of one type of descriptor over another is dictated to a large degree by the problem at hand. This is one of the principal reasons why the solution of image processing problems is aided significantly by having a flexible prototyping environment in which existing functions can be integrated with new code to gain flexibility and reduce development time. The material in this chapter is a good example of how to construct the basis for such an environment.