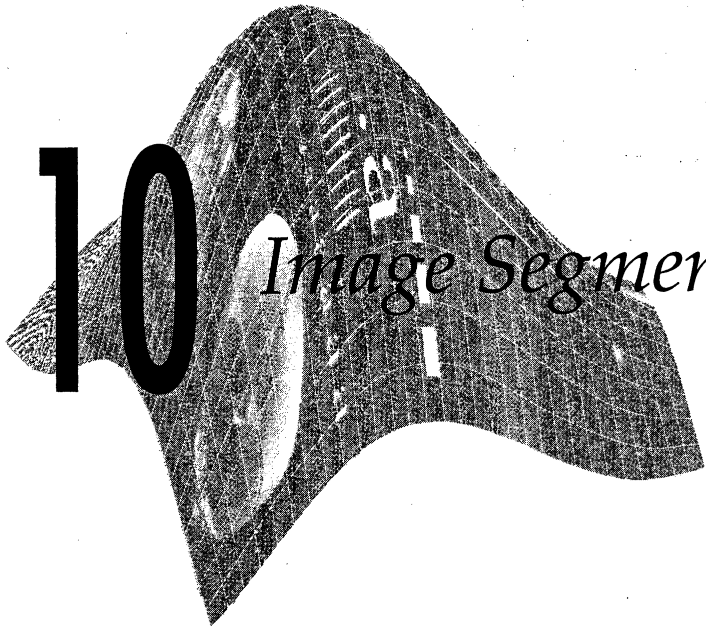
10 Image Segmentation



Preview

The material in the previous chapter began a transition from image processing methods whose inputs and outputs are images to methods in which the inputs are images, but the outputs are attributes extracted from those images. Segmentation is another major step in that direction.

Segmentation subdivides an image into its constituent regions or objects. The level to which the subdivision is carried depends on the problem being solved. That is, segmentation should stop when the objects of interest in an application have been isolated. For example, in the automated inspection of electronic assemblies, interest lies in analyzing images of the products with the objective of determining the presence or absence of specific anomalies, such as missing components or broken connection paths. There is no point in carrying segmentation past the level of detail required to identify those elements.

Segmentation of nontrivial images is one of the most difficult tasks in image processing. Segmentation accuracy determines the eventual success or failure of computerized analysis procedures. For this reason, considerable care should be taken to improve the probability of rugged segmentation. In some situations, such as industrial inspection applications, at least some measure of control over the environment is possible at times. In others, as in remote sensing, user control over image acquisition is limited principally to the choice of imaging sensors.

Segmentation algorithms for monochrome images generally are based on one of two basic properties of image intensity values: discontinuity and similarity. In the first category, the approach is to partition an image based on abrupt changes in intensity, such as edges in an image. The principal approaches in the second category are based on partitioning an image into regions that are similar according to a set of predefined criteria.

In this chapter we discuss a number of approaches in the two categories just mentioned as they apply to monochrome images (edge detection and segmentation of color images are discussed in Section 6.6). We begin the development with methods suitable for detecting intensity discontinuities such as points, lines, and edges. Edge detection in particular has been a staple of segmentation algorithms for many years. In addition to edge detection per se, we also discuss detecting linear edge segments using methods based on the *Hough transform*. The discussion of edge detection is followed by the introduction to thresholding techniques. Thresholding also is a fundamental approach to segmentation that enjoys a significant degree of popularity, especially in applications where speed is an important factor. The discussion on thresholding is followed by the development of region-oriented segmentation approaches. We conclude the chapter with a discussion of a morphological approach to segmentation called *watershed segmentation*. This approach is particularly attractive because it produces closed, well-defined regions, behaves in a global fashion, and provides a framework in which a priori knowledge about the images in a particular application can be utilized to improve segmentation results.

10.1 Point, Line, and Edge Detection

In this section we discuss techniques for detecting the three basic types of intensity discontinuities in a digital image: points, lines, and edges. The most common way to look for discontinuities is to run a mask through the image in the manner described in Sections 3.4 and 3.5. For a 3×3 mask this procedure involves computing the sum of products of the coefficients with the intensity levels contained in the region encompassed by the mask. That is, the response, R, of the mask at any point in the image is given by

$$\begin{aligned} R &= w_1 z_1 + w_2 z_2 + \cdots + w_9 z_9 \\ &= \sum_{i=1}^{9} w_i z_i \end{aligned}$$

where z_i is the intensity of the pixel associated with mask coefficient w_i. As before, the response of the mask is defined with respect to its center.

10.1.1 Point Detection

The detection of isolated points embedded in areas of constant or nearly constant intensity in an image is straightforward in principle. Using the mask shown in Fig. 10.1, we say that an isolated point has been detected at the location on which the mask is centered if

$$|R| \geq T$$

−1	−1	−1
−1	8	−1
−1	−1	−1

FIGURE 10.1 A mask for point detection.

where T is a nonnegative threshold. Point detection is implemented in MATLAB using function `imfilter`, with the mask in Fig. 10.1, or other similar mask. The important requirements are that the strongest response of a mask must be when the mask is centered on an isolated point, and that the response be 0 in areas of constant intensity.

If `T` is given, the following command implements the point-detection approach just discussed:

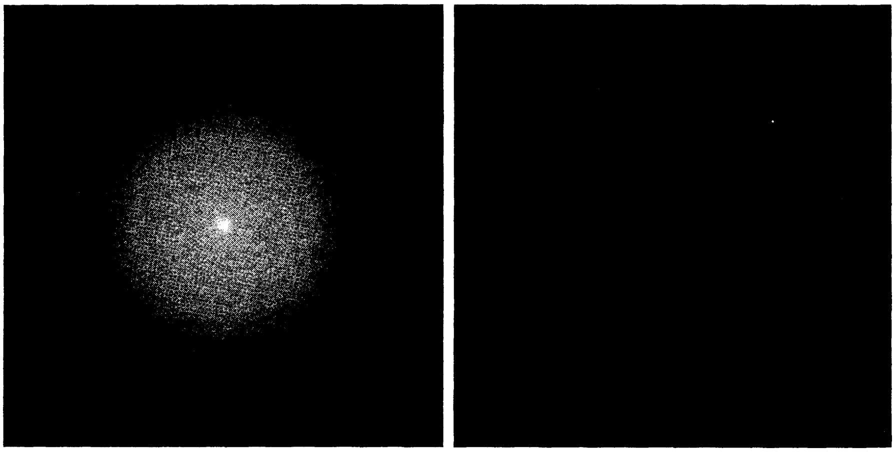
```
>> g = abs(imfilter(double(f), w)) >= T;
```

where `f` is the input image, `w` is an appropriate point-detection mask [e.g., the mask in Fig. 10.1], and `g` is the resulting image. Recall from the discussion in Section 3.4.1 that `imfilter` converts its output to the class of the input, so we use `double(f)` in the filtering operation to prevent premature truncation of values if the input is of class `uint8`, and because the `abs` operation does not accept integer data. The output image `g` is of class `logical`; its values are 0 and 1. If `T` is not given, its value often is chosen based on the filtered result, in which case the previous command string is broken down into three basic steps: (1) Compute the filtered image, `abs(imfilter(double(f), w))`, (2) find the value for `T` using the data from the filtered image, and (3) compare the filtered image against `T`. This approach is illustrated in the following example.

EXAMPLE 10.1: Point detection.

■ Figure 10.2(a) shows an image with a nearly invisible black point in the dark gray area of the northeast quadrant. Letting `f` denote this image, we find the location of the point as follows:

```
>> w = [-1 -1 -1; -1 8 -1; -1 -1 -1];
>> g = abs(imfilter(double(f), w));
>> T = max(g(:));
>> g = g >= T;
>> imshow(g)
```



a b

FIGURE 10.2 (a) Gray-scale image with a nearly invisible isolated black point in the dark gray area of the northeast quadrant. (b) Image showing the detected point. (The point was enlarged to make it easier to see.)

By selecting T to be the maximum value in the filtered image, g, and then finding all points in g such that g >= T, we identify the points that give the largest response. The assumption is that all these points are isolated points embedded in a constant or nearly constant background. Note that the test against T was conducted using the >= operator for consistency in notation. Since T was selected in this case to be the maximum value in g, clearly there can be no points in g with values greater than T. As Fig. 10.2(b) shows, there was a single isolated point that satisfied the condition g >= T with T set to max(g(:)). ■

Another approach to point detection is to find the points in all neighborhoods of size $m \times n$ for which the difference of the maximum and minimum pixels values exceeds a specified value of T. This approach can be implemented using function ordfilt2 introduced in Section 3.5.2:

```
>> g = imsubtract(ordfilt2(f, m*n, ones(m, n)), ...
                               ordfilt2(f, 1, ones(m, n)));
>> g = g >= T;
```

It is easily verified that choosing T = max(g(:)) yields the same result as in Fig. 10.2(b). The preceding formulation is more flexible than using the mask in Fig. 10.1. For example, if we wanted to compute the difference between the highest and the next highest pixel value in a neighborhood, we would replace the 1 on the far right of the preceding expression by m*n – 1. Other variations of this basic theme are formulated in a similar manner.

10.1.2 Line Detection

The next level of complexity is line detection. Consider the masks in Fig. 10.3. If the first mask were moved around an image, it would respond more strongly to lines (one pixel thick) oriented horizontally. With a constant background, the maximum response would result when the line passed through the middle row of the mask. Similarly, the second mask in Fig. 10.3 responds best to lines oriented at +45°; the third mask to vertical lines; and the fourth mask to lines in the −45° direction. Note that the preferred direction of each mask is weighted with a larger coefficient (i.e., 2) than other possible directions. The coefficients of each mask sum to zero, indicating a zero response from the mask in areas of constant intensity.

−1	−1	−1
2	2	2
−1	−1	−1

Horizontal

−1	−1	2
−1	2	−1
2	−1	−1

+45°

−1	2	−1
−1	2	−1
−1	2	−1

Vertical

2	−1	−1
−1	2	−1
−1	−1	2

−45°

FIGURE 10.3 Line detector masks.

Let R_1, R_2, R_3, and R_4 denote the responses of the masks in Fig. 10.3, from left to right, where the R's are given by the equation in the previous section. Suppose that the four masks are run individually through an image. If, at a certain point in the image, $|R_i| > |R_j|$, for all $j \neq i$, that point is said to be more likely associated with a line in the direction of mask i. For example, if at a point in the image, $|R_1| > |R_j|$ for $j = 2, 3, 4$, that particular point is said to be more likely associated with a horizontal line. Alternatively, we may be interested in detecting lines in a specified direction. In this case, we would use the mask associated with that direction and threshold its output, as in the equation in the previous section. In other words, if we are interested in detecting all the lines in an image in the direction defined by a given mask, we simply run the mask through the image and threshold the absolute value of the result. The points that are left are the strongest responses, which, for lines one pixel thick, correspond closest to the direction defined by the mask. The following example illustrates this procedure.

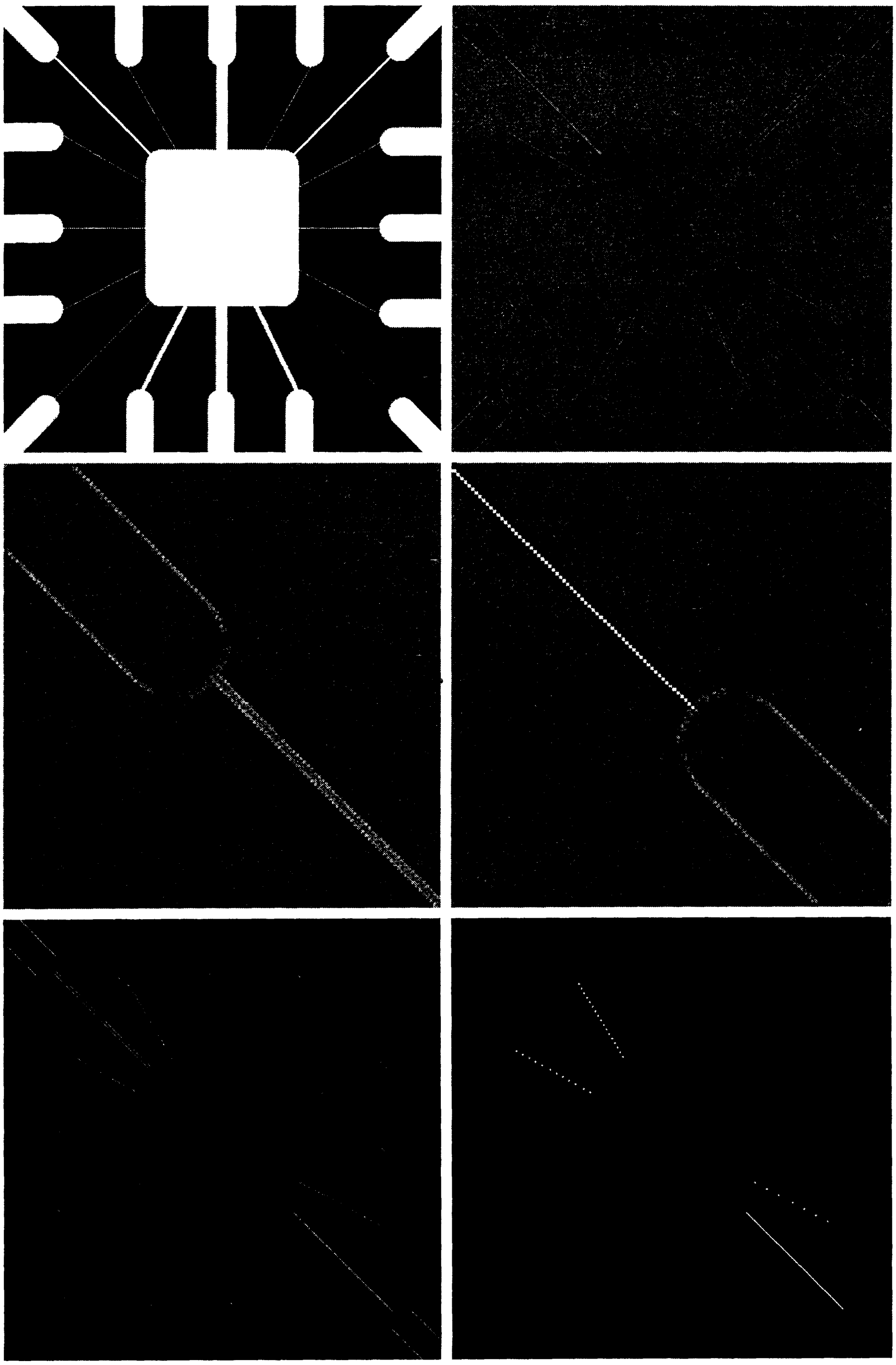
EXAMPLE 10.2: Detection of lines in a specified direction.

■ Figure 10.4(a) shows a digitized (binary) portion of a wire-bond mask for an electronic circuit. The image size is 486 × 486 pixels. Suppose that we are interested in finding all the lines that are one pixel thick, oriented at −45°. For this purpose, we use the last mask in Fig. 10.3. Figures 10.4(b) through (f) were generated using the following commands, where f is the image in Fig. 10.4(a):

```
>> w = [2 -1 -1 ; -1 2 -1; -1 -1 2];
>> g = imfilter(double(f), w);
>> imshow(g, [ ])  % Fig. 10.4(b)
>> gtop = g(1:120, 1:120);
>> gtop = pixeldup(gtop, 4);
>> figure, imshow(gtop, [ ]) % Fig. 10.4(c)
>> gbot = g(end-119:end, end-119:end);
>> gbot = pixeldup(gbot, 4);
>> figure, imshow(gbot, [ ]) % Fig. 10.4(d)
>> g = abs(g);
>> figure, imshow(g, [ ]) % Fig. 10.4(e)
>> T = max(g(:));
>> g = g >= T;
>> figure, imshow(g)  % Fig. 10.4(f)
```

The shades darker than the gray background in Fig. 10.4(b) correspond to negative values. There are two main segments oriented in the −45° direction, one at the top, left and one at the bottom, right [Figs. 10.4(c) and (d) show zoomed sections of these two areas]. Note how much brighter the straight line segment in Fig. 10.4(d) is than the segment in Fig. 10.4(c). The reason is that the component in the bottom, right of Fig. 10.4(a) is one pixel thick, while the one at the top, left is not. The mask response is stronger for the one-pixel-thick component.

Figure 10.4(e) shows the absolute value of Fig. 10.4(b). Since we are interested in the strongest response, we let T equal the maximum value in this image. Figure 10.4(f) shows in white the points whose values satisfied the



a b
c d
e f

FIGURE 10.4
(a) Image of a wire-bond mask. (b) Result of processing with the −45° detector in Fig. 10.3. (c) Zoomed view of the top, left region of (b). (d) Zoomed view of the bottom, right section of (b). (e) Absolute value of (b). (f) All points (in white) whose values satisfied the condition g >= T, where g is the image in (e). (The points in (f) were enlarged slightly to make them easier to see.)

condition `g >= T`, where `g` is the image in Fig. 10.4(e). The isolated points in this figure are points that also had similarly strong responses to the mask. In the original image, these points and their immediate neighbors are oriented in such a way that the mask produced a maximum response at those isolated locations. These isolated points can be detected using the mask in Fig. 10.1 and then deleted, or they could be deleted using morphological operators, as discussed in the last chapter. ■

10.1.3 Edge Detection Using Function `edge`

Although point and line detection certainly are important in any discussion on image segmentation, edge detection is by far the most common approach for detecting meaningful discontinuities in intensity values. Such discontinuities are detected by using first- and second-order derivatives. The first-order derivative of choice in image processing is the gradient, defined in Section 6.6.1. We repeat the pertinent equations here for convenience. The *gradient* of a 2-D function, $f(x, y)$, is defined as the *vector*

$$\nabla \mathbf{f} = \begin{bmatrix} G_x \\ G_y \end{bmatrix} = \begin{bmatrix} \dfrac{\partial f}{\partial x} \\ \dfrac{\partial f}{\partial y} \end{bmatrix}$$

The magnitude of this vector is

$$\begin{aligned} \nabla f &= \text{mag}(\nabla \mathbf{f}) = [G_x^2 + G_y^2]^{1/2} \\ &= [(\partial f/\partial x)^2 + (\partial f/\partial y)^2]^{1/2} \end{aligned}$$

To simplify computation, this quantity is approximated sometimes by omitting the square-root operation,

$$\nabla f \approx G_x^2 + G_y^2$$

or by using absolute values,

$$\nabla f \approx |G_x| + |G_y|$$

These approximations still behave as derivatives; that is, they are zero in areas of constant intensity and their values are proportional to the degree of intensity change in areas whose pixel values are variable. It is common practice to refer to the magnitude of the gradient or its approximations simply as "the gradient."

A fundamental property of the gradient vector is that it points in the direction of the maximum rate of change of f at coordinates (x, y). The angle at which this maximum rate of change occurs is

$$\alpha(x, y) = \tan^{-1}\left(\frac{G_y}{G_x}\right)$$

One of the key issues is how to estimate the derivatives G_x and G_y digitally. The various approaches used by function `edge` are discussed later in this section.

Second-order derivatives in image processing are generally computed using the Laplacian introduced in Section 3.5.1. That is, the Laplacian of a 2-D function $f(x, y)$ is formed from second-order derivatives, as follows:

$$\nabla^2 f(x, y) = \frac{\partial^2 f(x, y)}{\partial x^2} + \frac{\partial^2 f(x, y)}{\partial y^2}$$

The Laplacian is seldom used by itself for edge detection because, as a second-order derivative, it is unacceptably sensitive to noise, its magnitude produces double edges, and it is unable to detect edge direction. However, as discussed later in this section, the Laplacian can be a powerful complement when used in combination with other edge-detection techniques. For example, although its double edges make it unsuitably for edge detection directly, this property can be used for edge *location*.

With the preceding discussion as background, the basic idea behind edge detection is to find places in an image where the intensity changes rapidly, using one of two general criteria:

1. Find places where the first derivative of the intensity is greater in magnitude than a specified threshold.
2. Find places where the second derivative of the intensity has a zero crossing.

IPT's function `edge` provides several derivative estimators based on the criteria just discussed. For some of these estimators, it is possible to specify whether the edge detector is sensitive to horizontal or vertical edges or to both. The general syntax for this function is

```
[g, t] = edge(f, 'method', parameters)
```

where `f` is the input image, `method` is one of the approaches listed in Table 10.1, and `parameters` are additional parameters explained in the following discussion. In the output, `g` is a logical array with 1s at the locations where edge points were detected in `f` and 0s elsewhere. Parameter `t` is optional; it gives the threshold used by `edge` to determine which gradient values are strong enough to be called edge points.

Sobel Edge Detector

The *Sobel* edge detector uses the masks in Fig. 10.5(b) to approximate digitally the first derivatives G_x and G_y. In other words, the gradient at the center point in a neighborhood is computed as follows by the Sobel detector:

$$\begin{aligned} g &= [G_x^2 + G_y^2]^{1/2} \\ &= \{[(z_7 + 2z_8 + z_9) - (z_1 + 2z_2 + z_3)]^2 \\ &\quad + [(z_3 + 2z_6 + z_9) - (z_1 + 2z_4 + z_7)]^2\}^{1/2} \end{aligned}$$

TABLE 10.1
Edge detectors available in function `edge`.

Edge Detector	Basic Properties
Sobel	Finds edges using the Sobel approximation to the derivatives shown in Fig. 10.5(b).
Prewitt	Finds edges using the Prewitt approximation to the derivatives shown in Fig. 10.5(c).
Roberts	Finds edges using the Roberts approximation to the derivatives shown in Fig. 10.5(d).
Laplacian of a Gaussian (LoG)	Finds edges by looking for zero crossings after filtering $f(x, y)$ with a Gaussian filter.
Zero crossings	Finds edges by looking for zero crossings after filtering $f(x, y)$ with a user-specified filter.
Canny	Finds edges by looking for local maxima of the gradient of $f(x, y)$. The gradient is calculated using the derivative of a Gaussian filter. The method uses two thresholds to detect strong and weak edges, and includes the weak edges in the output only if they are connected to strong edges. Therefore, this method is more likely to detect true weak edges.

Then, we say that a pixel at location (x, y) is an edge pixel if $g \geq T$ at that location, where T is a specified threshold.

From the discussion in Section 3.5.1, we know that Sobel edge detection can be implemented by filtering an image, `f`, (using `imfilter`) with the left mask in Fig. 10.5(b), filtering `f` again with the other mask, squaring the pixels values of each filtered image, adding the two results, and computing their square root. Similar comments apply to the second and third entries in Table 10.1. Function `edge` simply packages the preceding operations into one function call and adds other features, such as accepting a threshold value or determining a threshold automatically. In addition, `edge` contains edge detection techniques that are not implementable directly with `imfilter`.

The general calling syntax for the Sobel detector is

```
[g, t] = edge(f, 'sobel', T, dir)
```

where `f` is the input image, `T` is a specified threshold, and `dir` specifies the preferred direction of the edges detected: `'horizontal'`, `'vertical'`, or `'both'` (the default). As noted earlier, `g` is a `logical` image containing 1s at locations where edges were detected and 0s elsewhere. Parameter `t` in the output is optional. It is the threshold value used by `edge`. If `T` is specified, then `t = T`. Otherwise, if `T` is not specified (or is empty, `[ ]`), `edge` sets `t` equal to a threshold it determines automatically and then uses for edge detection. One of the principal reason for including `t` in the output argument is to get an initial value for the threshold. Function `edge` uses the Sobel detector as a default if the syntax `g = edge(f)`, or `[g, t] = edge(f)`, is used.

z_1	z_2	z_3
z_4	z_5	z_6
z_7	z_8	z_9

Image neighborhood

−1	−2	−1
0	0	0
1	2	1

$G_x = (z_7 + 2z_8 + z_9) - (z_1 + 2z_2 + z_3)$

−1	0	1
−2	0	2
−1	0	1

$G_y = (z_3 + 2z_6 + z_9) - (z_1 + 2z_4 + z_7)$

Sobel

−1	−1	−1
0	0	0
1	1	1

$G_x = (z_7 + z_8 + z_9) - (z_1 + z_2 + z_3)$

−1	0	1
−1	0	1
−1	0	1

$G_y = (z_3 + z_6 + z_9) - (z_1 + z_4 + z_7)$

Prewitt

−1	0
0	1

$G_x = z_9 - z_5$

0	−1
1	0

$G_y = z_8 - z_6$

Roberts

a
b
c
d

FIGURE 10.5 Some edge detector masks and the first-order derivatives they implement.

Prewitt Edge Detector

The Prewitt edge detector uses the masks in Fig. 10.5(c) to approximate digitally the first derivatives G_x and G_y. Its general calling syntax is

```
[g, t] = edge(f, 'prewitt', T, dir)
```

The parameters of this function are identical to the Sobel parameters. The Prewitt detector is slightly simpler to implement computationally than the Sobel detector, but it tends to produce somewhat noisier results. (It can be shown that the coefficient with value 2 in the Sobel detector provides smoothing.)

Roberts Edge Detector

The Roberts edge detector uses the masks in Fig. 10.5(d) to approximate digitally the first derivatives G_x and G_y. Its general calling syntax is

```
[g, t] = edge(f, 'roberts', T, dir)
```

The parameters of this function are identical to the Sobel parameters. The Roberts detector is one of the oldest edge detectors in digital image processing, and as Fig. 10.5(d) shows, it is also the simplest. This detector is used considerably less than the others in Fig. 10.5 due in part to its limited functionality (e.g., it is not symmetric and cannot be generalized to detect edges that are multiples of 45°). However, it still is used frequently in hardware implementations where simplicity and speed are dominant factors.

Laplacian of a Gaussian (LoG) Detector

Consider the Gaussian function

$$h(r) = -e^{-\frac{r^2}{2\sigma^2}}$$

where $r^2 = x^2 + y^2$ and σ is the standard deviation. This is a smoothing function which, if convolved with an image, will blur it. The degree of blurring is determined by the value of σ. The Laplacian of this function (the second derivative with respect to r) is

$$\nabla^2 h(r) = -\left[\frac{r^2 - \sigma^2}{\sigma^4}\right] e^{-\frac{r^2}{2\sigma^2}}$$

For obvious reasons, this function is called the Laplacian of a Gaussian (LoG). Because the second derivative is a linear operation, convolving (filtering) an image with $\nabla^2 h(r)$ is the same as convolving the image with the smoothing function first and then computing the Laplacian of the result. This is the key concept underlying the LoG detector. We convolve the image with $\nabla^2 h(r)$, knowing that it has two effects: It smoothes the image (thus reducing noise), and it computes the Laplacian, which yields a double-edge image. Locating edges then consists of finding the zero crossings between the double edges.

The general calling syntax for the LoG detector is

```
[g, t] = edge(f, 'log', T, sigma)
```

where `sigma` is the standard deviation and the other parameters are as explained previously. The default value for `sigma` is 2. As before, `edge` ignores any edges that are not stronger than `T`. If `T` is not provided, or it is empty, `[ ]`, `edge` chooses the value automatically. Setting `T` to 0 produces edges that are closed contours, a familiar characteristic of the LoG method.

Zero-Crossings Detector

This detector is based on the same concept as the LoG method, but the convolution is carried out using a specified filter function, `H`. The calling syntax is

```
[g, t] = edge(f, 'zerocross', T, H)
```

The other parameters are as explained for the LoG detector.

Canny Edge Detector

The Canny detector (Canny [1986]) is the most powerful edge detector provided by function edge. The method can be summarized as follows:

1. The image is smoothed using a Gaussian filter with a specified standard deviation, σ, to reduce noise.
2. The local gradient, $g(x, y) = [G_x^2 + G_y^2]^{1/2}$, and edge direction, $\alpha(x, y) = \tan^{-1}(G_y/G_x)$, are computed at each point. Any of the first three techniques in Table 10.1 can be used to compute G_x and G_y. An edge point is defined to be a point whose strength is locally maximum in the direction of the gradient.
3. The edge points determined in (2) give rise to ridges in the gradient magnitude image. The algorithm then tracks along the top of these ridges and sets to zero all pixels that are not actually on the ridge top so as to give a thin line in the output, a process known as *nonmaximal suppression*. The ridge pixels are then thresholded using two thresholds, $T1$ and $T2$, with $T1 < T2$. Ridge pixels with values greater than $T2$ are said to be "strong" edge pixels. Ridge pixels with values between $T1$ and $T2$ are said to be "weak" edge pixels.
4. Finally, the algorithm performs edge linking by incorporating the weak pixels that are 8-connected to the strong pixels.

The syntax for the Canny edge detector is

```
[g, t] = edge(f, 'canny', T, sigma)
```

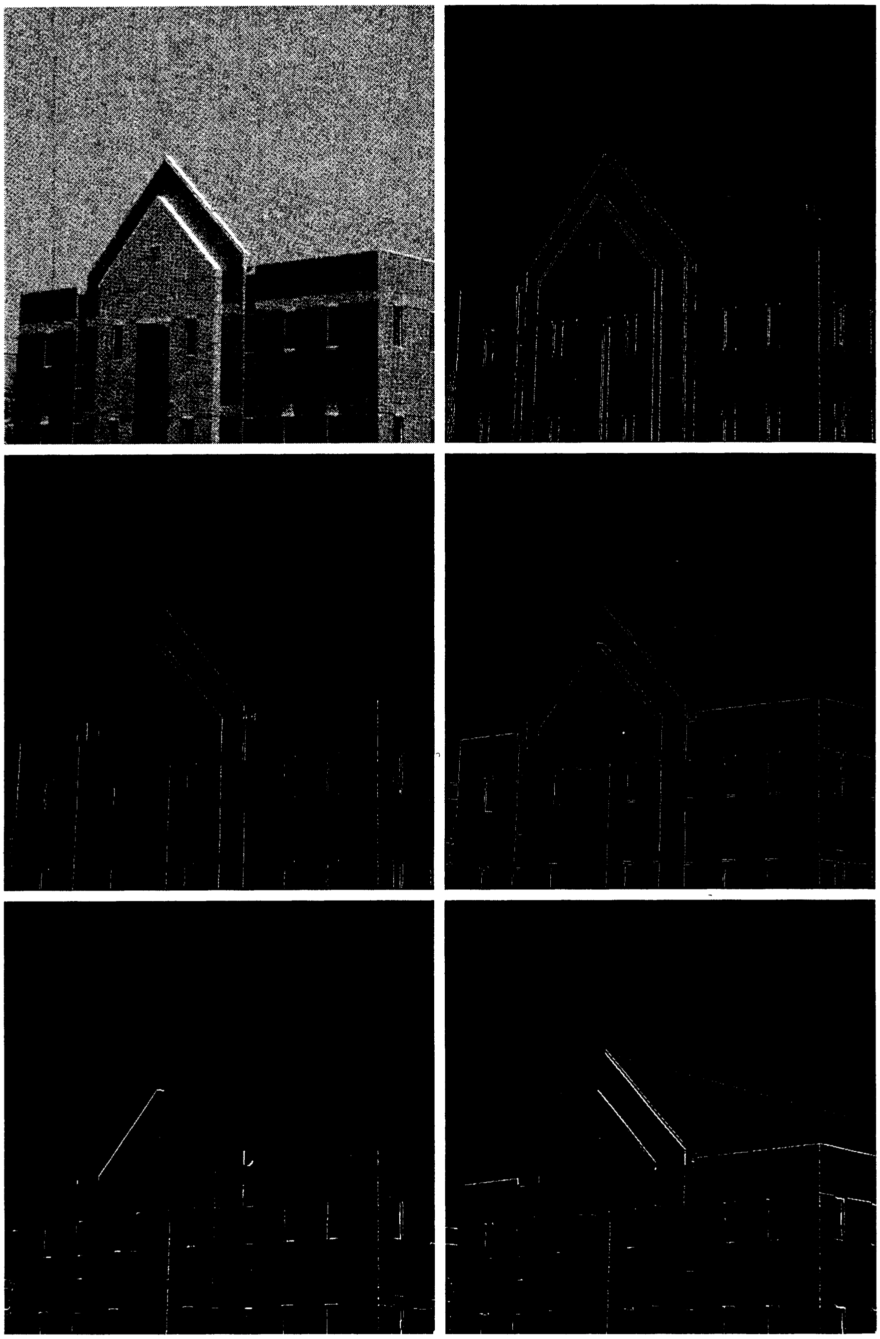
where T is a vector, T = [T1, T2], containing the two thresholds explained in step 3 of the preceding procedure, and sigma is the standard deviation of the smoothing filter. If t is included in the output argument, it is a two-element vector containing the two threshold values used by the algorithm. The rest of the syntax is as explained for the other methods, including the automatic computation of thresholds if T is not supplied. The default value for sigma is 1.

EXAMPLE 10.3: Edge extraction with the Sobel detector.

■ We can extract and display the vertical edges in the image, f, of Fig. 10.6(a) using the commands

```
>> [gv, t] = edge(f, 'sobel', 'vertical');
>> imshow(gv)
>> t

t =

   0.0516
```

As Fig. 10.6(b) shows, the predominant edges in the result are vertical (the inclined edges have vertical and horizontal components, so they are detected as well). We can clean up the weaker edges somewhat by specifying a higher threshold value. For example, Fig. 10.6(c) was generated using the command



a b
c d
e f

FIGURE 10.6 (a) Original image. (b) Result of function `edge` using a vertical Sobel mask with the threshold determined automatically. (c) Result using a specified threshold. (d) Result of determining both vertical and horizontal edges with a specified threshold. (e) Result of computing edges at 45° with `imfilter` using a specified mask and a specified threshold. (f) Result of computing edges at −45° with `imfilter` using a specified mask and a specified threshold.

```
>> gv = edge(f, 'sobel', 0.15, 'vertical');
```

Using the same value of T in the command

```
>> gboth = edge(f, 'sobel', 0.15);
```

resulted in Fig. 10.6(d), which shows predominantly vertical and horizontal edges.

Function edge does not compute Sobel edges at ±45°. To compute such edges we need to specify the mask and use imfilter. For example, Fig. 10.6(e) was generated using the commands

```
>> w45 = [-2 -1 0; -1 0 1; 0 1 2]

w45 =

    -2    -1     0
    -1     0     1
     0     1     2

>> g45 = imfilter(double(f), w45, 'replicate');
>> T = 0.3*max(abs(g45(:)));
>> g45 = g45 >= T;
>> figure, imshow(g45);
```

The value of T *was chosen experimentally to show results comparable with Figs. 10(c) and 10(d).*

The strongest edge in Fig. 10.6(e) is the edge oriented at 45°. Similarly, using the mask wm45 = [0 1 2; -1 0 1; -2 -1 0] with the same sequence of commands resulted in the strong edges oriented at −45° shown in Fig. 10.6(f).

Using the 'prewitt' and 'roberts' options in function edge follows the same general procedure just illustrated for the Sobel edge detector. ■

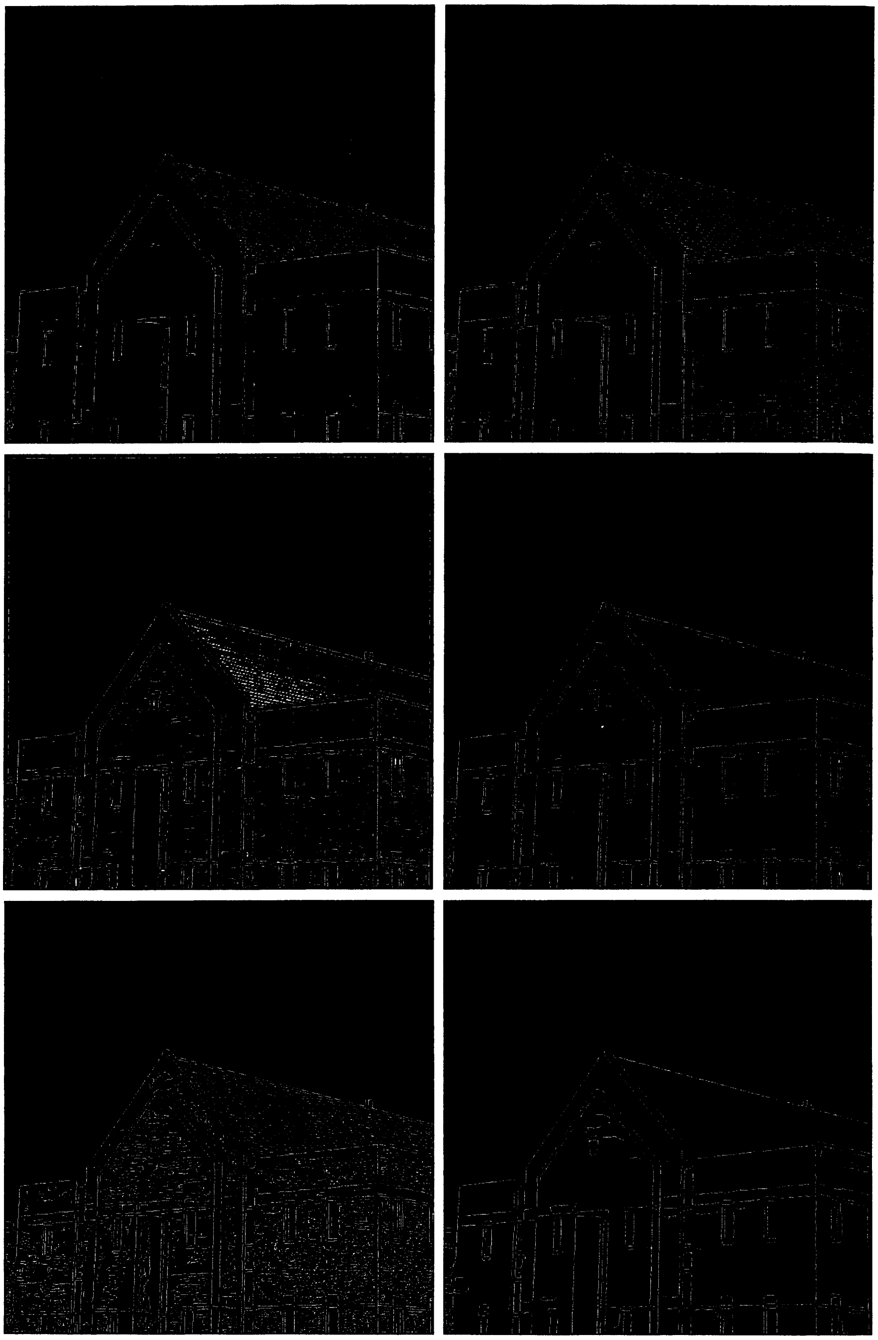
EXAMPLE 10.4: Comparison of the Sobel, LoG, and Canny edge detectors.

■ In this example we compare the relative performance of the Sobel, LoG, and Canny edge detectors. The objective is to produce a clean edge "map" by extracting the principal edge features of the building image, f, in Fig. 10.6(a), while reducing "irrelevant" detail, such as the fine texture in the brick walls and tile roof. The principal edges of interest in this discussion are the building corners, the windows, the light-brick structure framing the entrance, the entrance itself, the roofline, and the concrete band surrounding the building about two-thirds of the distance above ground level.

The left column in Fig. 10.7 shows the edge images obtained using the default syntax for the 'sobel', 'log', and 'canny' options:

```
>> [g_sobel_default, ts] = edge(f, 'sobel'); % Fig. 10.7(a)
>> [g_log_default, tlog] = edge(f, 'log');   % Fig. 10.7(c)
>> [g_canny_default, tc] = edge(f, 'canny'); % Fig. 10.7(e)
```

The values of the thresholds in the output argument resulting from the preceding computations were ts = 0.074, tlog = 0.0025, and tc = [0.019, 0.047]. The defaults values of sigma for the 'log' and 'canny' options were 2.0 and 1.0, respectively. With the exception of the Sobel image, the default results were far from the objective of producing clean edge maps.



a b
c d
e f

FIGURE 10.7 Left column: Default results for the Sobel, LoG, and Canny edge detectors. Right column: Results obtained interactively to bring out the principal features in the original image of Fig. 10.6(a) while reducing irrelevant, fine detail. The Canny edge detector produced the best results by far.

Starting with the default values, the parameters in each option were varied interactively with the objective of bringing out the principal features mentioned earlier, while reducing irrelevant detail as much as possible. The results in the right column of Fig. 10.7 were obtained with the following commands:

```
>> g_sobel_best = edge(f, 'sobel', 0.05);            % Fig. 10.7(b)
>> g_log_best  = edge(f, 'log', 0.003, 2.25);        % Fig. 10.7(d)
>> g_canny_best = edge(f, 'canny', [0.04 0.10], 1.5); % Fig. 10.7(f)
```

As Fig. 10.7(b) shows, the Sobel result actually deviated even further from the objective when we tried to bring out the concrete band and left edge of the entrance way. The LoG result in Fig. 10.7(d) is somewhat better than the Sobel result and much better than the LoG default, but it still could not bring out the left edge of the main entrance nor the concrete band around the building. The Canny result [Fig. 10.7(f)] is superior by far to the other two results. Note in particular how the left edge of the entrance was clearly detected, as were the concrete band and other details such as the complete roof ventilation grill above the main entrance. In addition to detecting the desired features, the Canny detector also produced the cleanest edge map. ■

10.2 Line Detection Using the Hough Transform

Ideally, the methods discussed in the previous section should yield pixels lying only on edges. In practice, the resulting pixels seldom characterize an edge completely because of noise, breaks in the edge from nonuniform illumination, and other effects that introduce spurious intensity discontinuities. Thus edge-detection algorithms typically are followed by linking procedures to assemble edge pixels into meaningful edges. One approach that can be used to find and link line segments in an image is the *Hough transform* (Hough [1962]).

Given a set of points in an image (typically a binary image), suppose that we want to find subsets of these points that lie on straight lines. One possible solution is to first find all lines determined by every pair of points and then find all subsets of points that are close to particular lines. The problem with this procedure is that it involves finding $n(n-1)/2 \sim n^2$ lines and then performing $n(n(n-1))/2 \sim n^3$ comparisons of every point to all lines. This approach is computationally prohibitive in all but the most trivial applications.

With the Hough transform, on the other hand, we consider a point (x_i, y_i) and all the lines that pass through it. Infinitely many lines pass through (x_i, y_i), all of which satisfy the slope-intercept equation $y_i = ax_i + b$ for some values of a and b. Writing this equation as $b = -x_i a + y_i$ and considering the *ab*-plane (also called *parameter space*) yields the equation of a *single* line for a fixed pair (x_i, y_i). Furthermore, a second point (x_j, y_j) also has a line in parameter space associated with it, and this line intersects the line associated with (x_i, y_i) at (a', b'), where a' is the slope and b' the intercept of the line containing both (x_i, y_i) and (x_j, y_j) in the *xy*-plane. In fact, all points contained on this line have lines in parameter space that intersect at (a', b'). Figure 10.8 illustrates these concepts.

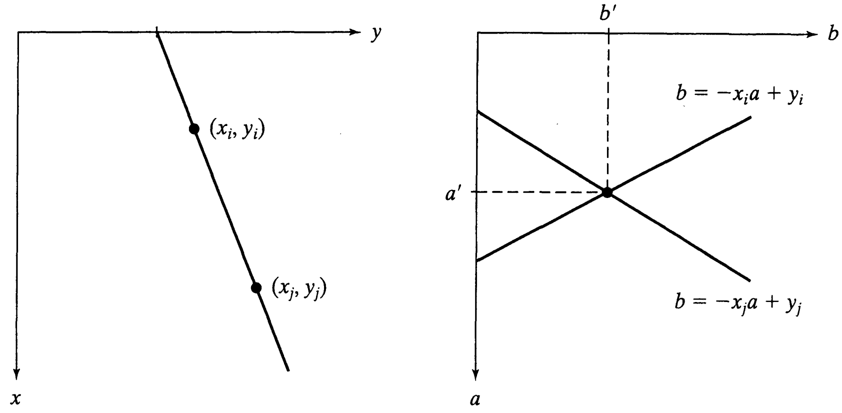


FIGURE 10.8 (a) xy-plane. (b) Parameter space.

In principle, the parameter-space lines corresponding to all image points (x_i, y_i) could be plotted, and then image lines could be identified by where large numbers of parameter-space lines intersect. A practical difficulty with this approach, however, is that a (the slope of the line) approaches infinity as the line approaches the vertical direction. One way around this difficulty is to use the normal representation of a line:

$$x \cos \theta + y \sin \theta = \rho$$

Figure 10.9(a) illustrates the geometric interpretation of the parameters ρ and θ. A horizontal line has $\theta = 0°$, with ρ being equal to the positive x-intercept. Similarly, a vertical line has $\theta = 90°$, with ρ being equal to the positive y-intercept, or $\theta = -90°$, with ρ being equal to the negative y intercept. Each sinusoidal curve in Figure 10.9(b) represents the family of lines that pass through a particular point (x_i, y_i). The intersection point (ρ', θ') corresponds to the line that passes through both (x_i, y_i) and (x_j, y_j).

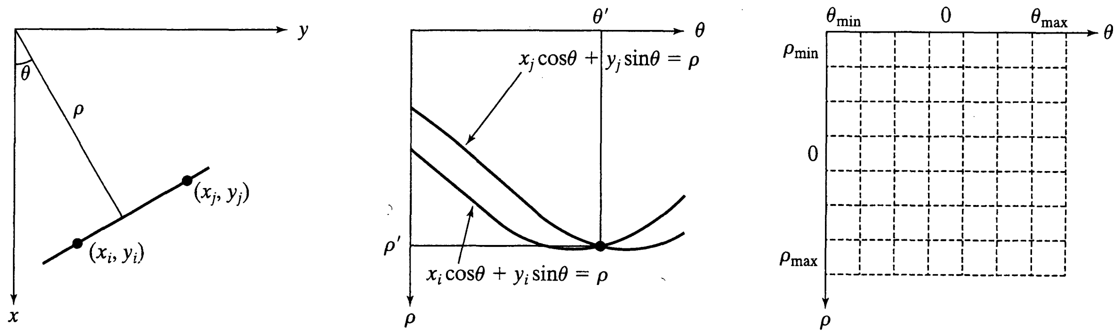


FIGURE 10.9 (a) (ρ, θ) parameterization of lines in the xy-plane. (b) Sinusoidal curves in the $\rho\theta$-plane; the point of intersection, (ρ', θ'), corresponds to the parameters of the line joining (x_i, y_i) and (x_j, y_j). (c) Division of the $\rho\theta$-plane into accumulator cells.

The computational attractiveness of the Hough transform arises from subdividing the $\rho\theta$ parameter space into so-called *accumulator cells*, as illustrated in Figure 10.9(c), where $(\rho_{\min}, \rho_{\max})$ and $(\theta_{\min}, \theta_{\max})$ are the expected ranges of the parameter values. Usually, the maximum range of values is $-90° \leq \theta \leq 90°$ and $-D \leq \rho \leq D$, where D is the distance between corners in the image. The cell at coordinates (i, j), with accumulator value $A(i, j)$, corresponds to the square associated with parameter space coordinates (ρ_i, θ_j). Initially, these cells are set to zero. Then, for every nonbackground point (x_k, y_k) in the image plane, we let θ equal each of the allowed subdivision values on the θ axis and solve for the corresponding ρ using the equation $\rho = x_k \cos\theta + y_k \sin\theta$. The resulting ρ-values are then rounded off to the nearest allowed cell value along the ρ-axis. The corresponding accumulator cell is then incremented. At the end of this procedure, a value of Q in $A(i, j)$, means that Q points in the xy-plane lie on the line $x\cos\theta_j + y\sin\theta_j = \rho_i$. The number of subdivisions in the $\rho\theta$-plane determines the accuracy of the colinearity of these points.

A function for computing the Hough transform is given next. This function makes use of *sparse* matrices, which are matrices that contain a small number of nonzero elements. This characteristic provides advantages in both matrix storage space and computation time. Given a matrix A, we convert it to sparse matrix format by using function sparse, which has the basic syntax

```
S = sparse(A)
```

sparse

For example,

```
>> A = [ 0   0   0   5
         0   2   0   0
         1   3   0   0
         0   0   4   0 ];
>> S = sparse(A)

S =

    (3,1)    1
    (2,2)    2
    (3,2)    3
    (4,3)    4
    (1,4)    5
```

This output lists the nonzero elements of S, together with their row and column indices. The elements are sorted by columns.

A syntax used more frequently with function sparse consists of five arguments:

```
S = sparse(r, c, s, m, n)
```

Here, `r` and `c` are vectors of row and column indices, respectively, of the nonzero elements of the matrix we wish to convert to sparse format. Parameter `s` is a vector containing the values that correspond to the index pairs (`r, c`), and `m` and `n` are the row and column dimensions for the resulting matrix. For instance, the matrix `S` in the previous example can be generated directly using the command

```
>> S = sparse([3 2 3 4 1], [1 2 2 3 4], [1 2 3 4 5], 4, 4);
```

There are a number of other syntax forms for function `sparse`, as detailed in the help page for this function.

Given a sparse matrix `S` generated by any of its applicable syntax forms, we can obtain the full matrix back by using function `full`, whose syntax is

full

```
A = full(S)
```

To explore Hough transform-based line detection in MATLAB, we first write a function, `hough.m`, that computes the Hough transform:

hough

```
function [h, theta, rho] = hough(f, dtheta, drho)
%HOUGH Hough transform.
%   [H, THETA, RHO] = HOUGH(F, DTHETA, DRHO) computes the Hough
%   transform of the image F. DTHETA specifies the spacing (in
%   degrees) of the Hough transform bins along the theta axis. DRHO
%   specifies the spacing of the Hough transform bins along the rho
%   axis. H is the Hough transform matrix. It is NRHO-by-NTHETA,
%   where NRHO = 2*ceil(norm(size(F))/DRHO) - 1, and NTHETA =
%   2*ceil(90/DTHETA). Note that if 90/DTHETA is not an integer, the
%   actual angle spacing will be 90 / ceil(90/DTHETA).
%
%   THETA is an NTHETA-element vector containing the angle (in
%   degrees) corresponding to each column of H. RHO is an
%   NRHO-element vector containing the value of rho corresponding to
%   each row of H.
%
%   [H, THETA, RHO] = HOUGH(F) computes the Hough transform using
%   DTHETA = 1 and DRHO = 1.

if nargin < 3
   drho = 1;
end
if nargin < 2
   dtheta = 1;
end

f = double(f);
[M,N] = size(f);
theta = linspace(-90, 0, ceil(90/dtheta) + 1);
theta = [theta -fliplr(theta(2:end - 1))];
ntheta = length(theta);
```

```
D = sqrt((M - 1)^2 + (N - 1)^2);
q = ceil(D/drho);
nrho = 2*q - 1;
rho = linspace(-q*drho, q*drho, nrho);

[x, y, val] = find(f);
x = x - 1;  y = y - 1;

% Initialize output.
h = zeros(nrho, length(theta));

% To avoid excessive memory usage, process 1000 nonzero pixel
% values at a time.
for k = 1:ceil(length(val)/1000)
   first = (k - 1)*1000 + 1;
   last = min(first+999, length(x));

   x_matrix     = repmat(x(first:last), 1, ntheta);
   y_matrix     = repmat(y(first:last), 1, ntheta);
   val_matrix   = repmat(val(first:last), 1, ntheta);
   theta_matrix = repmat(theta, size(x_matrix, 1), 1)*pi/180;

   rho_matrix = x_matrix.*cos(theta_matrix) + ...
       y_matrix.*sin(theta_matrix);
slope = (nrho - 1)/(rho(end) - rho(1));
rho_bin_index = round(slope*(rho_matrix - rho(1)) + 1);

theta_bin_index = repmat(1:ntheta, size(x_matrix, 1), 1);

% Take advantage of the fact that the SPARSE function, which
% constructs a sparse matrix, accumulates values when input
% indices are repeated. That's the behavior we want for the
% Hough transform. We want the output to be a full (nonsparse)
% matrix, however, so we call function FULL on the output of
% SPARSE.
h = h + full(sparse(rho_bin_index(:), theta_bin_index(:), ...
                    val_matrix(:), nrho, ntheta));
end
```

EXAMPLE 10.5: Illustration of the Hough transform.

■ In this example we illustrate the use of function hough on a simple binary image. First we construct an image containing isolated foreground pixels in several locations.

```
>> f = zeros(101, 101);
>> f(1, 1)     = 1;  f(101, 1) = 1; f(1, 101) = 1;
>> f(101, 101) = 1;  f(51, 51) = 1;
```

Figure 10.10(a) shows our test image. Next we compute and display the Hough transform.

```
>> H = hough(f);
>> imshow(H, [ ])
```

Figure 10.10(b) shows the result, displayed with imshow in the familiar way. However, it often is more useful to visualize Hough transforms in a larger plot,

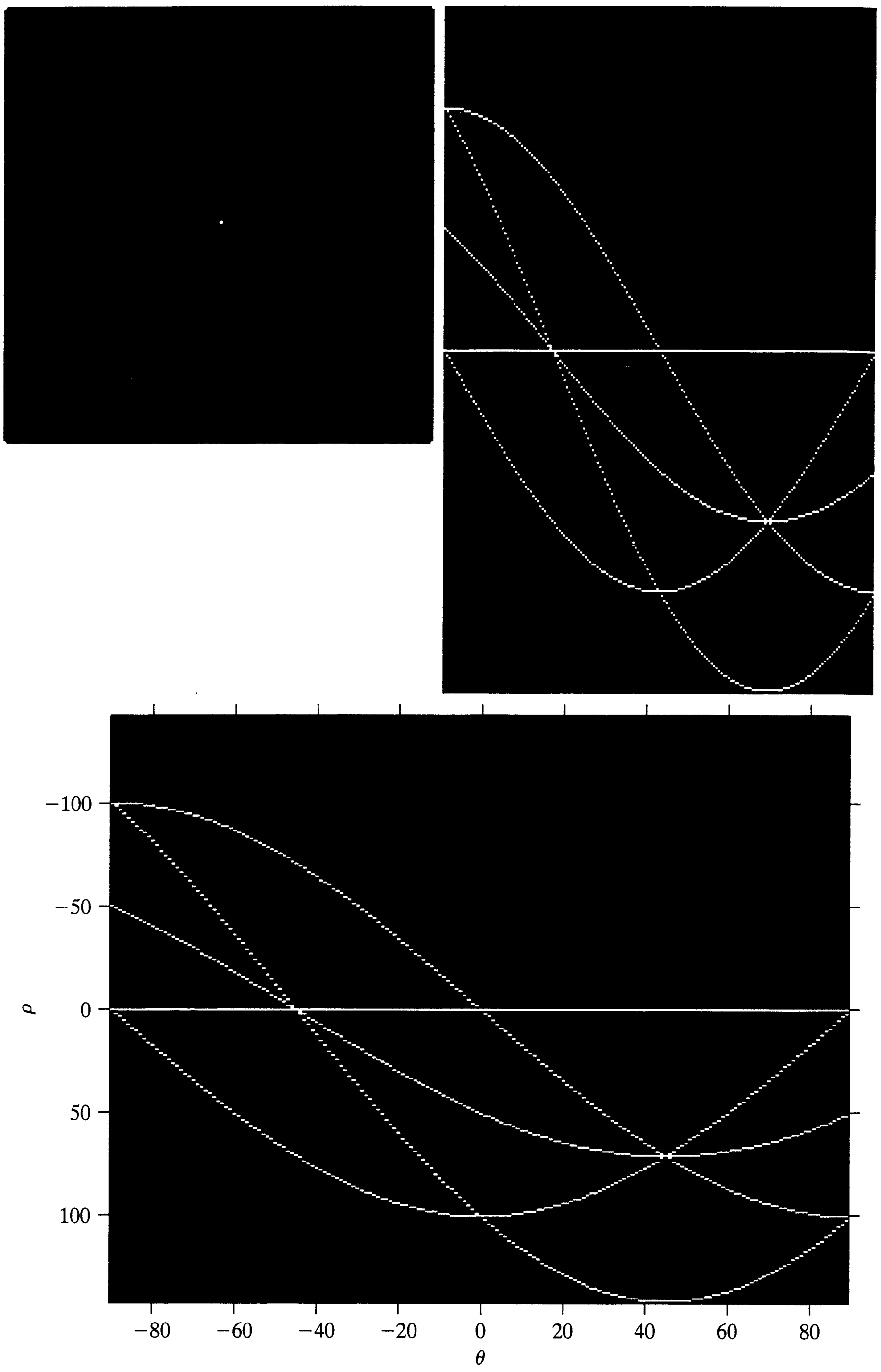


FIGURE 10.10 (a) Binary image with five dots (four of the dots are in the corners). (b) Hough transform displayed using `imshow`. (c) Alternative Hough transform display with axis labeling. (The dots in (a) were enlarged to make them easier to see.)

with labeled axes. In the next code fragment we call hough with three output arguments; the second two output arguments contain the θ and ρ values corresponding to each column and row, respectively, of the Hough transform matrix. These vectors, theta and rho, can then be passed as additional input arguments to imshow to control the horizontal and vertical axis labeling. We also pass the 'notruesize' option to imshow. The axis function is used to turn on axis labeling and to make the display fill the rectangular shape of the figure. Finally the xlabel and ylabel functions (see Section 3.3.1) are used to label the axes using a LaTeX-style notation for Greek letters.

```
>> [H, theta, rho] = hough(f);
>> imshow(theta, rho, H, [ ], 'notruesize')
>> axis on, axis normal
>> xlabel('\theta'), ylabel('\rho')
```

Figure 10.10(c) shows the labeled result. The intersections of three sinusoidal curves at $\pm 45°$ indicate that there are two sets of three collinear points in f. The intersections of two sinusoidal curves at $(\theta, \rho) = (-90, 0)$, $(-90, -100)$, $(0, 0)$, and $(0, 100)$ indicate that there are four sets of collinear points that lie along vertical and horizontal lines. ■

10.2.1 Hough Transform Peak Detection

The first step in using the Hough transform for line detection and linking is peak detection. Finding a meaningful set of distinct peaks in a Hough transform can be challenging. Because of the quantization in space of the digital image, the quantization in parameter space of the Hough transform, as well as the fact that edges in typical images are not perfectly straight, Hough transform peaks tend to lie in more than one Hough transform cell. One strategy to overcome this problem is the following:

1. Find the Hough transform cell containing the highest value and record its location.
2. Suppress (set to zero) Hough transform cells in the immediate neighborhood of the maximum found in step 1.
3. Repeat until the desired number of peaks has been found, or until a specified threshold has been reached.

Function houghpeaks implements this strategy.

houghpeaks

```
function [r, c, hnew] = houghpeaks(h, numpeaks, threshold, nhood)
%HOUGHPEAKS Detect peaks in Hough transform.
%   [R, C, HNEW] = HOUGHPEAKS(H, NUMPEAKS, THRESHOLD, NHOOD) detects
%   peaks in the Hough transform matrix H. NUMPEAKS specifies the
%   maximum number of peak locations to look for. Values of H below
%   THRESHOLD will not be considered to be peaks. NHOOD is a
%   two-element vector specifying the size of the suppression
%   neighborhood. This is the neighborhood around each peak that is
%   set to zero after the peak is identified. The elements of NHOOD
%   must be positive, odd integers. R and C are the row and column
%   coordinates of the identified peaks. HNEW is the Hough transform
%   with peak neighborhood suppressed.
%
%   If NHOOD is omitted, it defaults to the smallest odd values >=
%   size(H)/50. If THRESHOLD is omitted, it defaults to
%   0.5*max(H(:)). If NUMPEAKS is omitted, it defaults to 1.

if nargin < 4
   nhood = size(h)/50;
   % Make sure the neighborhood size is odd.
   nhood = max(2*ceil(nhood/2) + 1, 1);
end
if nargin < 3
   threshold = 0.5 * max(h(:));
end
if nargin < 2
   numpeaks = 1;
end

done = false;
hnew = h; r = []; c = [];
while ~done
   [p, q] = find(hnew == max(hnew(:)));
   p = p(1); q = q(1);
   if hnew(p, q) >= threshold
      r(end + 1) = p; c(end + 1) = q;

      % Suppress this maximum and its close neighbors.
      p1 = p - (nhood(1) - 1)/2; p2 = p + (nhood(1) - 1)/2;
      q1 = q - (nhood(2) - 1)/2; q2 = q + (nhood(2) - 1)/2;
      [pp, qq] = ndgrid(p1:p2, q1:q2);
      pp = pp(:); qq = qq(:);

      % Throw away neighbor coordinates that are out of bounds in
      % the rho direction.
      badrho = find((pp < 1) | (pp > size(h, 1)));
      pp(badrho) = []; qq(badrho) = [];

      % For coordinates that are out of bounds in the theta
      % direction, we want to consider that H is antisymmetric
      % along the rho axis for theta = +/- 90 degrees.
      theta_too_low = find(qq < 1);
      qq(theta_too_low) = size(h, 2) + qq(theta_too_low);
      pp(theta_too_low) = size(h, 1) - pp(theta_too_low) + 1;
      theta_too_high = find(qq > size(h, 2));
      qq(theta_too_high) = qq(theta_too_high) - size(h, 2);
      pp(theta_too_high) = size(h, 1) - pp(theta_too_high) + 1;

      % Convert to linear indices to zero out all the values.
      hnew(sub2ind(size(hnew), pp, qq)) = 0;
      done = length(r) == numpeaks;
   else
      done = true;
   end
end
```

Function houghpeaks is illustrated in Example 10.6.

10.2.2 Hough Transform Line Detection and Linking

Once a set of candidate peaks has been identified in the Hough transform, it remains to be determined if there are line segments associated with those peaks, as well as where they start and end. For each peak, the first step is to find the location of all nonzero pixels in the image that contributed to that peak. For this purpose, we write function houghpixels.

houghpixels

```
function [r, c] = houghpixels(f, theta, rho, rbin, cbin)
%HOUGHPIXELS Compute image pixels belonging to Hough transform bin.
%   [R, C] = HOUGHPIXELS(F, THETA, RHO, RBIN, CBIN) computes the
%   row-column indices (R, C) for nonzero pixels in image F that map
%   to a particular Hough transform bin, (RBIN, CBIN). RBIN and CBIN
%   are scalars indicating the row-column bin location in the Hough
%   transform matrix returned by function HOUGH. THETA and RHO are
%   the second and third output arguments from the HOUGH function.

[x, y, val] = find(f);
x = x - 1; y = y - 1;

theta_c = theta(cbin) * pi / 180;
rho_xy = x*cos(theta_c) + y*sin(theta_c);
nrho = length(rho);
slope = (nrho - 1)/(rho(end) - rho(1));
rho_bin_index = round(slope*(rho_xy - rho(1)) + 1);

idx = find(rho_bin_index == rbin);

r = x(idx) + 1; c = y(idx) + 1;
```

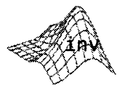
The pixels associated with the locations found using houghpixels must be grouped into line segments. Function houghlines uses the following strategy:

1. Rotate the pixel locations by $90° - \theta$ so that they lie approximately along a vertical line.
2. Sort the pixel locations by their rotated x-values.
3. Use function diff to locate gaps. Ignore small gaps; this has the effect of merging adjacent line segments that are separated by a small space.
4. Return information about line segments that are longer than some minimum length threshold.

houghlines

```
function lines = houghlines(f,theta,rho,rr,cc,fillgap,minlength)
%HOUGHLINES Extract line segments based on the Hough transform.
%   LINES = HOUGHLINES(F, THETA, RHO, RR, CC, FILLGAP, MINLENGTH)
%   extracts line segments in the image F associated with particular
%   bins in a Hough transform. THETA and RHO are vectors returned by
%   function HOUGH. Vectors RR and CC specify the rows and columns
%   of the Hough transform bins to use in searching for line
%   segments. If HOUGHLINES finds two line segments associated with
%   the same Hough transform bin that are separated by less than
%   FILLGAP pixels, HOUGHLINES merges them into a single line
%   segment. FILLGAP defaults to 20 if omitted. Merged line
%   segments less than MINLENGTH pixels long are discarded.
%   MINLENGTH defaults to 40 if omitted.
%
%   LINES is a structure array whose length equals the number of
%   merged line segments found. Each element of the structure array
%   has these fields:
%
%      point1   End-point of the line segment; two-element vector
%      point2   End-point of the line segment; two-element vector
%      length   Distance between point1 and point2
%      theta    Angle (in degrees) of the Hough transform bin
%      rho      Rho-axis position of the Hough transform bin

if nargin < 6
   fillgap = 20;
end
if nargin < 7
   minlength = 40;
end

numlines = 0; lines = struct;
for k = 1:length(rr)
   rbin = rr(k); cbin = cc(k);

   % Get all pixels associated with Hough transform cell.
   [r, c] = houghpixels(f, theta, rho, rbin, cbin);
   if isempty(r)
      continue
   end

   % Rotate the pixel locations about (1,1) so that they lie
   % approximately along a vertical line.
   omega = (90 - theta(cbin)) * pi / 180;
   T = [cos(omega) sin(omega); -sin(omega) cos(omega)];
   xy = [r - 1 c - 1] * T;
   x = sort(xy(:,1));

   % Find the gaps larger than the threshold.
   diff_x = [diff(x); Inf];
   idx = [0; find(diff_x > fillgap)];
   for p = 1:length(idx) - 1
      x1 = x(idx(p) + 1); x2 = x(idx(p + 1));
      linelength = x2 - x1;
      if linelength >= minlength
         point1 = [x1 rho(rbin)]; point2 = [x2 rho(rbin)];
```

```
        % Rotate the end-point locations back to the original
        % angle.
        Tinv = inv(T);
        point1 = point1 * Tinv; point2 = point2 * Tinv;

        numlines = numlines + 1;
        lines(numlines).point1 = point1 + 1;
        lines(numlines).point2 = point2 + 1;
        lines(numlines).length = linelength;
        lines(numlines).theta = theta(cbin);
        lines(numlines).rho = rho(rbin);
      end
  end
end
```



B = inv(A) *computes the inverse of square matrix* A.

EXAMPLE 10.6: Using the Hough transform for line detection and linking.

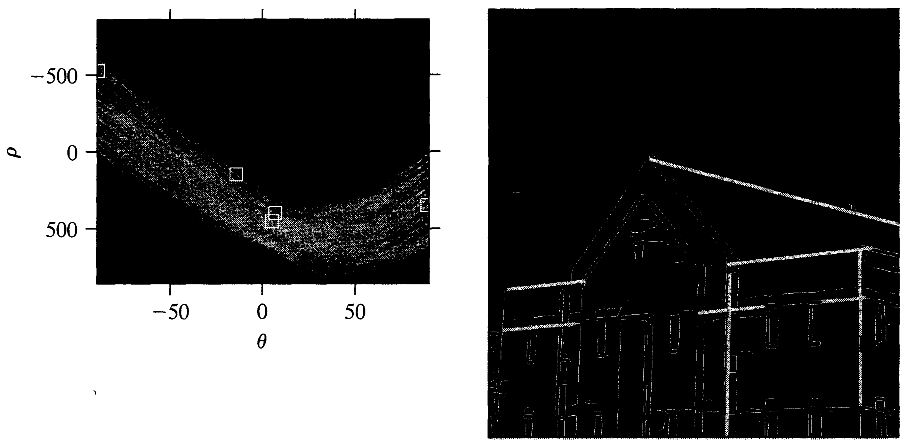
■ In this example we use functions hough, houghpeaks, and houghlines to find a set of line segments in the binary image, f, in Fig. 10.7(f). First, we compute and display the Hough transform, using a finer angular spacing than the default ($\Delta\theta = 0.5$ instead of 1.0).

```
>> [H, theta, rho] = hough(f, 0.5);
>> imshow(theta, rho, H, [ ], 'notruesize'), axis on, axis normal
>> xlabel('\theta'), ylabel('\rho')
```

Next we use function houghpeaks to find five Hough transform peaks that are likely to be significant.

```
>> [r, c] = houghpeaks(H, 5);
>> hold on
>> plot(theta(c), rho(r), 'linestyle', 'none', ...
      'marker', 's', 'color', 'w')
```

Figure 10.11(a) shows the Hough transform with the peak locations superimposed. Finally, we use function houghlines to find and link line segments, and



a b

FIGURE 10.11 (a) Hough transform with five peak locations selected. (b) Line segments corresponding to the Hough transform peaks.

we superimpose the line segments on the original binary image using imshow, hold on, and plot:

```
>> lines = houghlines(f, theta, rho, r, c)
>> figure, imshow(f), hold on
>> for k = 1:length(lines)
xy = [lines(k).point1 ; lines(k).point2];
plot(xy(:,2), xy(:,1), 'LineWidth', 4, 'Color', [.6 .6 .6]);
end
```

Figure 10.11(b) shows the resulting image with the detected segments superimposed as thick, gray lines. ■

10.3 Thresholding

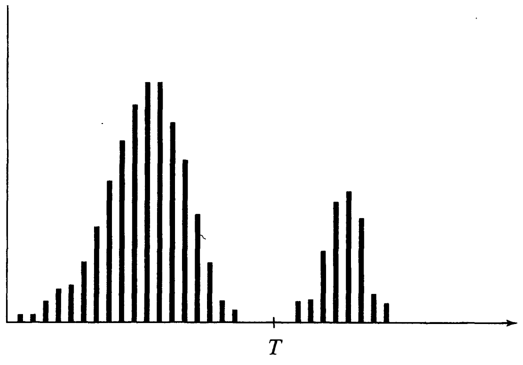
Because of its intuitive properties and simplicity of implementation, image thresholding enjoys a central position in applications of image segmentation. Simple thresholding was first introduced in Section 2.7.2, and we have used it in various discussions in the preceding chapters. In this section, we discuss ways of choosing the threshold value automatically, and we consider a method for varying the threshold according to the properties of local image neighborhoods.

Suppose that the intensity histogram shown in Fig. 10.12 corresponds to an image, $f(x, y)$, composed of light objects on a dark background, in such a way that object and background pixels have intensity levels grouped into two dominant modes. One obvious way to extract the objects from the background is to select a threshold T that separates these modes. Then any point (x, y) for which $f(x, y) \geq T$ is called an *object point*; otherwise, the point is called a *background point*. In other words, the thresholded image $g(x, y)$ is defined as

$$g(x, y) = \begin{cases} 1 & \text{if } f(x, y) \geq T \\ 0 & \text{if } f(x, y) < T \end{cases}$$

Pixels labeled 1 correspond to objects, whereas pixels labeled 0 correspond to the background. When T is a constant, this approach is called *global thresholding*.

FIGURE 10.12 Selecting a threshold by visually analyzing a bimodal histogram.



Methods for choosing a global threshold are discussed in Section 10.3.1. In Section 10.3.2 we discuss allowing the threshold to vary, which is called *local thresholding*.

10.3.1 Global Thresholding

One way to choose a threshold is by visual inspection of the image histogram. The histogram in Figure 10.12 clearly has two distinct modes; as a result, it is easy to choose a threshold T that separates them. Another method of choosing T is by trial and error, picking different thresholds until one is found that produces a good result as judged by the observer. This is particularly effective in an interactive environment, such as one that allows the user to change the threshold using a *widget* (graphical control) such as a slider and see the result immediately.

For choosing a threshold automatically, Gonzalez and Woods [2002] describe the following iterative procedure:

1. Select an initial estimate for T. (A suggested initial estimate is the midpoint between the minimum and maximum intensity values in the image.)
2. Segment the image using T. This will produce two groups of pixels: G_1, consisting of all pixels with intensity values $\geq T$, and G_2, consisting of pixels with values $< T$.
3. Compute the average intensity values μ_1 and μ_2 for the pixels in regions G_1 and G_2.
4. Compute a new threshold value:

$$T = \frac{1}{2}(\mu_1 + \mu_2)$$

5. Repeat steps 2 through 4 until the difference in T in successive iterations is smaller than a predefined parameter T_o.

We show how to implement this procedure in MATLAB in Example 10.7.

The toolbox provides a function called `graythresh` that computes a threshold using Otsu's method (Otsu [1979]). To examine the formulation of this histogram-based method, we start by treating the normalized histogram as a discrete probability density function, as in

$$p_r(r_q) = \frac{n_q}{n} \qquad q = 0, 1, 2, \ldots, L - 1$$

where n is the total number of pixels in the image, n_q is the number of pixels that have intensity level r_q, and L is the total number of possible intensity levels in the image. Now suppose that a threshold k is chosen such that C_0 is the set of pixels with levels $[0, 1, \ldots, k - 1]$ and C_1 is the set of pixels with levels $[k, k + 1, \ldots, L - 1]$. Otsu's method chooses the threshold value k that maximizes the *between-class variance* σ_B^2, which is defined as

$$\sigma_B^2 = \omega_0(\mu_0 - \mu_T)^2 + \omega_1(\mu_1 - \mu_T)^2$$

where

$$\omega_0 = \sum_{q=0}^{k-1} p_q(r_q)$$
$$\omega_1 = \sum_{q=k}^{L-1} p_q(r_q)$$
$$\mu_0 = \sum_{q=0}^{k-1} qp_q(r_q)/\omega_0$$
$$\mu_1 = \sum_{q=k}^{L-1} qp_q(r_q)/\omega_1$$
$$\mu_T = \sum_{q=0}^{L-1} qp_q(r_q)$$

Function `graythresh` takes an image, computes its histogram, and then finds the threshold value that maximizes σ_B^2. The threshold is returned as a normalized value between 0.0 and 1.0. The calling syntax for `graythresh` is

graythresh

```
T = graythresh(f)
```

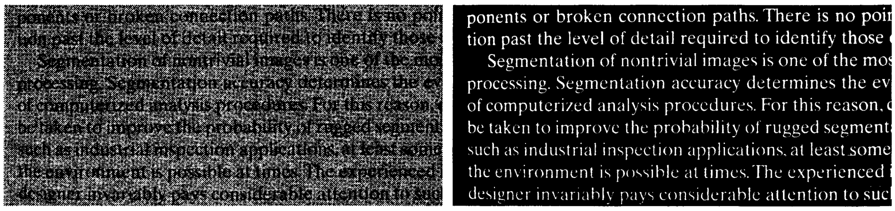
where `f` is the input image and `T` is the resulting threshold. To segment the image we use `T` in function `im2bw` introduced in Section 2.7.2. Because the threshold is normalized to the range [0, 1], it must be scaled to the proper range before it is used. For example, if `f` is of class `uint8`, we multiply `T` by 255 before using it.

EXAMPLE 10.7: Computing global thresholds.

■ In this example we illustrate the iterative procedure described previously as well as Otsu's method on the gray-scale image, `f`, of scanned text, shown in Fig. 10.13(a). The iterative method can be implemented as follows:

```
>> T = 0.5*(double(min(f(:))) + double(max(f(:))));
>> done = false;
>> while ~done
    g = f >= T;
    Tnext = 0.5*(mean(f(g)) + mean(f(~g)));
    done = abs(T - Tnext) < 0.5;
    T = Tnext;
end
```

For this particular image, the while loop executes four times and terminates


ponents or broken connection paths. There is no poi
tion past the level of detail required to identify those
Segmentation of nontrivial images is one of the mo
processing. Segmentation accuracy determines the ev
of computerized analysis procedures. For this reason,
be taken to improve the probability of rugged segment
such as industrial inspection applications, at least some
the environment is possible at times. The experienced
designer invariably pays considerable attention to suc

a b

FIGURE 10.13 (a) Scanned text. (b) Thresholded text obtained using function `graythresh`.

with T equal to 101.47.

Next we compute a threshold using function graythresh:

```
>> T2 = graythresh(f)
T2 =
    0.3961

>> T2 * 255
ans =
    101
```

Thresholding using these two values produces images that are almost indistinguishable from each other. Figure 10.13(b) shows the image thresholded using T2. ▪

10.3.2 Local Thresholding

Global thresholding methods can fail when the background illumination is uneven, as was illustrated in Figs. 9.26(a) and (b). A common practice in such situations is to preprocess the image to compensate for the illumination problems and then apply a global threshold to the preprocessed image. The improved thresholding result shown in Fig. 9.26(e) was computed by applying a morphological top-hat operator and then using graythresh on the result. We can show that this process is equivalent to thresholding $f(x, y)$ with a locally varying threshold function $T(x, y)$:

$$g(x, y) = \begin{cases} 1 & \text{if } f(x, y) \geq T(x, y) \\ 0 & \text{if } f(x, y) < T(x, y) \end{cases}$$

where

$$T(x, y) = f_o(x, y) + T_o$$

The image $f_o(x, y)$ is the morphological opening of f, and the constant T_o is the result of function graythresh applied to f_o.

10.4 Region-Based Segmentation

The objective of segmentation is to partition an image into regions. In Sections 10.1 and 10.2 we approached this problem by finding boundaries between regions based on discontinuities in intensity levels, whereas in Section 10.3 segmentation was accomplished via thresholds based on the distribution of pixel properties, such as intensity values. In this section we discuss segmentation techniques that are based on finding the regions directly.

10.4.1 Basic Formulation

Let R represent the entire image region. We may view segmentation as a process that partitions R into n subregions, $R_1, R_2, \ldots, R_n$, such that

(a) $\bigcup_{i=1}^{n} R_i = R$.

(b) R_i is a connected region, $i = 1, 2, \ldots, n$.

(c) $R_i \cap R_j = \emptyset$ for all i and $j, i \neq j$.
(d) $P(R_i) = \text{TRUE}$ for $i = 1, 2, \ldots, n$.
(e) $P(R_i \cup R_j) = \text{FALSE}$ for any adjacent regions R_i and R_j.

In the context of the discussion in Section 9.4, two disjoint regions, R_i and R_j, are said to be adjacent if their union forms a connected component.

Here, $P(R_i)$ is a logical predicate defined over the points in set R_i and $\emptyset$ is the null set.

Condition (a) indicates that the segmentation must be complete; that is, every pixel must be in a region. The second condition requires that points in a region be connected in some predefined sense (e.g., 4- or 8-connected). Condition (c) indicates that the regions must be disjoint. Condition (d) deals with the properties that must be satisfied by the pixels in a segmented region—for example $P(R_i) = \text{TRUE}$ if all pixels in R_i have the same gray level. Finally, condition (e) indicates that adjacent regions R_i and R_j are different in the sense of predicate P.

10.4.2 Region Growing

As its name implies, *region growing* is a procedure that groups pixels or subregions into larger regions based on predefined criteria for growth. The basic approach is to start with a set of "seed" points and from these grow regions by appending to each seed those neighboring pixels that have predefined properties similar to the seed (such as specific ranges of gray level or color).

Selecting a set of one or more starting points often can be based on the nature of the problem, as shown later in Example 10.8. When a priori information is not available, one procedure is to compute at every pixel the same set of properties that ultimately will be used to assign pixels to regions during the growing process. If the result of these computations shows clusters of values, the pixels whose properties place them near the centroid of these clusters can be used as seeds.

The selection of similarity criteria depends not only on the problem under consideration, but also on the type of image data available. For example, the analysis of land-use satellite imagery depends heavily on the use of color. This problem would be significantly more difficult, or even impossible, to handle without the inherent information available in color images. When the images are monochrome, region analysis must be carried out with a set of descriptors based on intensity levels (such as moments or texture) and spatial properties. We discuss descriptors useful for region characterization in Chapter 11.

Descriptors alone can yield misleading results if connectivity (adjacency) information is not used in the region-growing process. For example, visualize a random arrangement of pixels with only three distinct intensity values. Grouping pixels with the same intensity level to form a "region" without paying attention to connectivity would yield a segmentation result that is meaningless in the context of this discussion.

Another problem in region growing is the formulation of a stopping rule. Basically, growing a region should stop when no more pixels satisfy the criteria for inclusion in that region. Criteria such as intensity values, texture, and color, are local in nature and do not take into account the "history" of region growth. Additional criteria that increase the power of a region-growing algorithm utilize the concept of size, likeness between a candidate pixel and the pixels grown so far (such as a comparison of the intensity of a candidate and the average intensity of the grown region), and the shape of the region being grown. The use of these types of descriptors is based on the assumption that a model of expected results is at least partially available.

To illustrate the principles of how region segmentation can be handled in MATLAB, we develop next an M-function, called regiongrow, to do basic region growing. The syntax for this function is

```
[g, NR, SI, TI] = regiongrow(f, S, T)
```

where f is an image to be segmented and parameter S can be an array (the same size as f) or a scalar. If S is an array, it must contain 1s at all the coordinates where seed points are located and 0s elsewhere. Such an array can be determined by inspection, or by an external seed-finding function. If S is a scalar, it defines an intensity value such that all the points in f with that value become seed points. Similarly, T can be an array (the same size as f) or a scalar. If T is an array, it contains a threshold value for each location in f. If T is scalar, it defines a global threshold. The threshold value(s) is (are) used to test if a pixel in the image is sufficiently similar to the seed or seeds to which it is 8-connected.

For example, if S = a and T = b, and we are comparing intensities, then a pixel is said to be similar to a (in the sense of passing the threshold test) if the absolute value of the difference between its intensity and a is less than or equal to b. If, in addition, the pixel in question is 8-connected to one or more seed values, then the pixel is considered a member of one or more regions. Similar comments hold if S and T are arrays, the basic difference being that comparisons are done with the appropriate locations defined in S and corresponding values of T.

In the output, g is the segmented image, with the members of each region being labeled with an integer value. Parameter NR is the number of different regions. Parameter SI is an image containing the seed points, and parameter TI is an image containing the pixels that passed the threshold test before they were processed for connectivity. Both SI and TI are of the same size as f.

The code for function regiongrow is as follows. Note the use of Chapter 9 function bwmorph to reduce to 1 the number of connected seed points in each region in S (when S is an array) and function imreconstruct to find pixels connected to each seed.

regiongrow

```
function [g, NR, SI, TI] = regiongrow(f, S, T)
%REGIONGROW Perform segmentation by region growing.
%   [G, NR, SI, TI] = REGIONGROW(F, SR, T). S can be an array (the
%   same size as F) with a 1 at the coordinates of every seed point
%   and 0s elsewhere. S can also be a single seed value. Similarly,
%   T can be an array (the same size as F) containing a threshold
%   value for each pixel in F. T can also be a scalar, in which
%   case it becomes a global threshold.
%
```

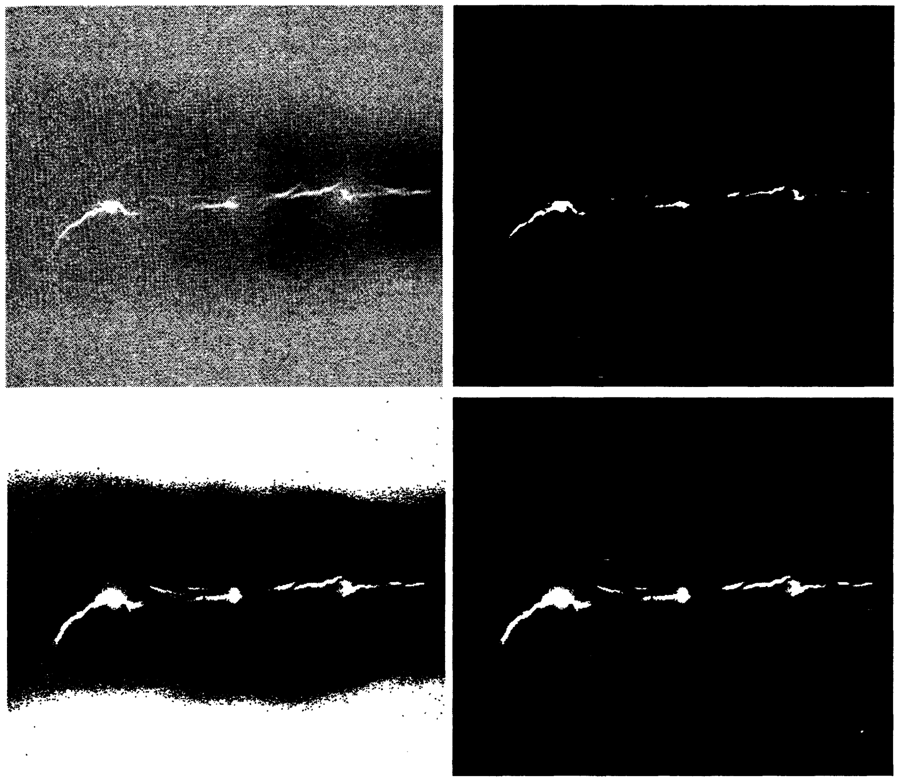
```
%  On the output, G is the result of region growing, with each
%  region labeled by a different integer, NR is the number of
%  regions, SI is the final seed image used by the algorithm, and TI
%  is the image consisting of the pixels in F that satisfied the
%  threshold test.

f = double(f);
% If S is a scalar, obtain the seed image.
if numel(S) == 1
   SI = f == S;
   S1 = S;
else
   % S is an array. Eliminate duplicate, connected seed locations
   % to reduce the number of loop executions in the following
   % sections of code.
   SI = bwmorph(S, 'shrink', Inf);
   J = find(SI);
   S1 = f(J); % Array of seed values.
end

TI = false(size(f));
for K = 1:length(S1)
   seedvalue = S1(K);
   S = abs(f - seedvalue) <= T;
   TI = TI | S;
end

% Use function imreconstruct with SI as the marker image to
% obtain the regions corresponding to each seed in S. Function
% bwlabel assigns a different integer to each connected region.
[g, NR] = bwlabel(imreconstruct(SI, TI));
```

`true` *is equivalent to* `logical(1)`, *and* `false` *is equivalent to* `logical(0)`.

EXAMPLE 10.8: Application of region growing to weld porosity detection.

■ Figure 10.14(a) shows an X-ray image of a weld (the horizontal dark region) containing several cracks and porosities (the bright, white streaks running horizontally through the middle of the image). We wish to use function `regiongrow` to segment the regions corresponding to weld failures. These segmented regions could be used for inspection, for inclusion in a database of historical studies, for controlling an automated welding system, and for other numerous applications.

The first order of business is to determine the initial seed points. In this application, it is known that some pixels in areas of defective welds tend to have the maximum allowable digital value (255 in this case). Based in this information, we let `S = 255`. The next step is to choose a threshold or threshold array. In this particular example we used `T = 65`. This number was based on analysis of the histogram in Fig. 10.15 and represents the difference between 255 and the location of the first major valley to the left, which is representative of the highest intensity value in the dark weld region. As noted earlier, a pixel has to



a b
c d

FIGURE 10.14 (a) Image showing defective welds. (b) Seed points. (c) Binary image showing all the pixels (in white) that passed the threshold test. (d) Result after all the pixels in (c) were analyzed for 8-connectivity to the seed points. (Original image courtesy of X-TEK Systems, Ltd.)

be 8-connected to at least one pixel in a region to be included in that region. If a pixel is found to be connected to more than one region, the regions are automatically merged by `regiongrow`.

Figure 10.14(b) shows the seed points (image `SI`). They are numerous in this case because the seeds were specified simply as all points in the image with a value of 255. Figure 10.14(c) is image `TI`. It shows all the points that passed the threshold test; that is, the points with intensity z_i, such that $|z_i - S| \leq T$. Figure 10.14(d) shows the result of extracting all the pixels in Figure 10.14(c) that were connected to the seed points. This is the segmented image, `g`. It is evident by comparing this image with the original that the region growing procedure did indeed segment the defective welds with a reasonable degree of accuracy.

Finally, we note by looking at the histogram in Fig. 10.15 that it would not have been possible to obtain the same or equivalent solution by any of the thresholding methods discussed in Section 10.3. The use of connectivity was a fundamental requirement in this case. ■

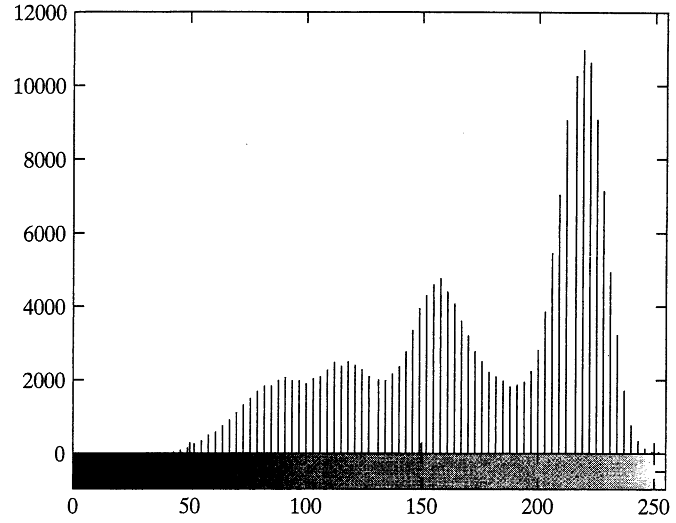


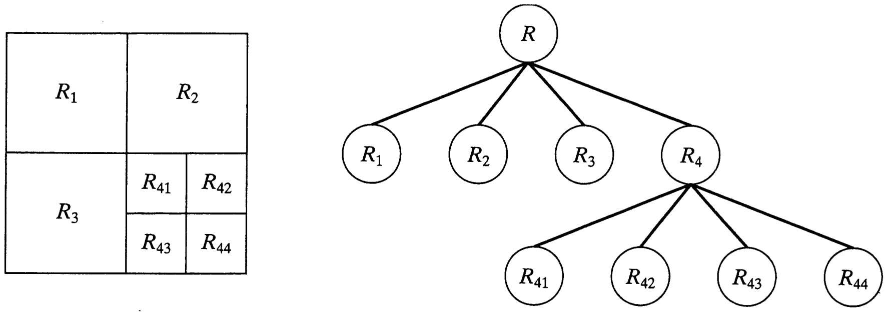
FIGURE 10.15 Histogram of Fig. 10.14(a).

10.4.3 Region Splitting and Merging

The procedure just discussed grows regions from a set of seed points. An alternative is to subdivide an image initially into a set of arbitrary, disjointed regions and then merge and/or split the regions in an attempt to satisfy the conditions stated in Section 10.4.1. The basics of splitting and merging are discussed next.

Let R represent the entire image region and select a predicate P. One approach for segmenting R is to subdivide it successively into smaller and smaller quadrant regions so that, for any region R_i, $P(R_i) = \text{TRUE}$. We start with the entire region. If $P(R) = \text{FALSE}$, we divide the image into quadrants. If P is FALSE for any quadrant, we subdivide that quadrant into subquadrants, and so on. This particular splitting technique has a convenient representation in the form of a so-called *quadtree*; that is, a tree in which each node has exactly four descendants, as illustrated in Fig. 10.16 (the subimages corresponding to the nodes of a quadtree sometimes are called *quadregions* or *quadimages*). Note that the root of the tree corresponds to the entire image and that each node corresponds to the subdivision of a node into four descendant nodes. In this case, only R_4 was subdivided further.

If only splitting is used, the final partition normally contains adjacent regions with identical properties. This drawback can be remedied by allowing merging, as



a b

FIGURE 10.16 (a) Partitioned image. (b) Corresponding quadtree.

well as splitting. Satisfying the constraints of Section 10.4.1 requires merging only adjacent regions whose combined pixels satisfy the predicate P. That is, two adjacent regions R_j and R_k are merged only if $P(R_j \cup R_k) = \text{TRUE}$.

The preceding discussion may be summarized by the following procedure in which, at any step,

1. Split into four disjoint quadrants any region R_i for which $P(R_i) = \text{FALSE}$.
2. When no further splitting is possible, merge any adjacent regions R_j and R_k for which $P(R_j \cup R_k) = \text{TRUE}$.
3. Stop when no further merging is possible.

Numerous variations of the preceding basic theme are possible. For example, a significant simplification results if we allow merging of any two adjacent regions R_i and R_j if each one satisfies the predicate individually. This results in a much simpler (and faster) algorithm because testing of the predicate is limited to individual quadregions. As Example 10.9 shows, this simplification is still capable of yielding good segmentation results in practice. Using this approach in step 2 of the procedure, all quadregions that satisfy the predicate are filled with 1s and their connectivity can be easily examined using, for example, function `imreconstruct`. This function, in effect, accomplishes the desired merging of adjacent quadregions. The quadregions that do not satisfy the predicate are filled with 0s to create a segmented image.

The function in IPT for implementing quadtree decomposition is `qtdecomp`. The syntax of interest in this section is

```
S = qtdecomp(f, @split_test, parameters)
```

qtdecomp

where `f` is the input image and `S` is a sparse matrix containing the quadtree structure. If `S(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 `S(k, m)`. Function `split_test` (see function `splitmerge` below for an example) is used to determine whether a region is to be split or not, and `parameters` are any additional parameters (separated by commas) required by `split_test`. The mechanics of this are similar to those discussed in Section 3.4.2 for function `coltfilt`.

Other forms of `qtdecomp` *are discussed in Section 11.2.2.*

To get the actual quadregion pixel values in a quadtree decomposition we use function `qtgetblk`, with syntax

```
[vals, r, c] = qtgetblk(f, S, m)
```

qtgetblk

where `vals` is an array containing the values of the blocks of size `m` × `m` in the quadtree decomposition of `f`, and `S` 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.

We illustrate the use of function `qtdecomp` by writing a basic split-and-merge M-function that uses the simplification discussed earlier, in which two regions are merged if each satisfies the predicate individually. The function, which we call `splitmerge`, has the following calling syntax:

```
g = splitmerge(f, mindim, @predicate)
```

where f is the input image and g is the output image in which each connected region is labeled with a different integer. Parameter mindim defines the size of the smallest block allowed in the decomposition; this parameter has to be a positive integer power of 2.

Function predicate is a user-defined function that must be included in the MATLAB path. Its syntax is

```
flag = predicate(region)
```

This function must be written so that it returns true (a logical 1) if the pixels in region satisfy the predicate defined by the code in the function; otherwise, the value of flag must be false (a logical 0). Example 10.9 illustrates the use of this function.

Function splitmerge has a simple structure. First, the image is partitioned using function qtdecomp. Function split_test uses predicate to determine if a region should be split or not. Because when a region is split into four it is not known which (if any) of the resulting four regions will pass the predicate test individually, it is necessary to examine the regions after the fact to see which regions in the partitioned image pass the test. Function predicate is used for this purpose also. Any quadregion that passes the test is filled with 1s. Any that does not is filled with 0s. A marker array is created by selecting one element of each région that is filled with 1s. This array is used in conjunction with the partitioned image to determine region connectivity (adjacency); function imreconstruct is used for this purpose.

The code for function splitmerge follows. The simple predicate function shown in the comments section of the code is used in Example 10.9. Note that the size of the input image is brought up to a square whose dimensions are the minimum integer power of 2 that encompasses the image. This is a requirement of function qtdecomp to guarantee that splits down to size 1 are possible.

splitmerge

```
function g = splitmerge(f, mindim, fun)
%SPLITMERGE Segment an image using a split-and-merge algorithm.
%   G = SPLITMERGE(F, MINDIM, @PREDICATE) segments image F by using a
%   split-and-merge approach based on quadtree decomposition. MINDIM
%   (a positive integer power of 2) specifies the minimum dimension
%   of the quadtree regions (subimages) allowed. If necessary, the
%   program pads the input image with zeros to the nearest square
%   size that is an integer power of 2. This guarantees that the
%   algorithm used in the quadtree decomposition will be able to
%   split the image down to blocks of size 1-by-1. The result is
%   cropped back to the original size of the input image. In the
%   output, G, each connected region is labeled with a different
%   integer.
%
%   Note that in the function call we use @PREDICATE for the value of
%   fun. PREDICATE is a function in the MATLAB path, provided by the
%   user. Its syntax is
%
```

```
%      FLAG = PREDICATE(REGION) which must return TRUE if the pixels
%      in REGION satisfy the predicate defined by the code in the
%      function; otherwise, the value of FLAG must be FALSE.
%
%   The following simple example of function PREDICATE is used in
%   Example 10.9 of the book. It sets FLAG to TRUE if the
%   intensities of the pixels in REGION have a standard deviation
%   that exceeds 10, and their mean intensity is between 0 and 125.
%   Otherwise FLAG is set to false.
%
%      function flag = predicate(region)
%      sd = std2(region);
%      m = mean2(region);
%      flag = (sd > 10) & (m > 0) & (m < 125);

% Pad image with zeros to guarantee that function qtdecomp will
% split regions down to size 1-by-1.
Q = 2^nextpow2(max(size(f)));
[M, N] = size(f);
f = padarray(f, [Q - M, Q - N], 'post');

% Perform splitting first.
S = qtdecomp(f, @split_test, mindim, fun);

% Now merge by looking at each quadregion and setting all its
% elements to 1 if the block satisfies the predicate.

% Get the size of the largest block. Use full because S is sparse.
Lmax = full(max(S(:)));
% Set the output image initially to all zeros. The MARKER array is
% used later to establish connectivity.
g = zeros(size(f));
MARKER = zeros(size(f));
% Begin the merging stage.
for K = 1:Lmax
   [vals, r, c] = qtgetblk(f, S, K);
   if ~isempty(vals)
      % Check the predicate for each of the regions
      % of size K-by-K with coordinates given by vectors
      % r and c.
      for I = 1:length(r)
         xlow = r(I); ylow = c(I);
         xhigh = xlow + K - 1; yhigh = ylow + K - 1;
         region = f(xlow:xhigh, ylow:yhigh);
         flag = feval(fun, region);
         if flag
            g(xlow:xhigh, ylow:yhigh) = 1;
            MARKER(xlow, ylow) = 1;
         end
      end
   end
end
```

feval

`feval(fun, param)` *evaluates function* `fun` *with parameter* `param`. *See the help page for* `feval` *for other syntax forms applicable to this function.*

```
% Finally, obtain each connected region and label it with a
% different integer value using function bwlabel.
g = bwlabel(imreconstruct(MARKER, g));

% Crop and exit
g = g(1:M, 1:N);

%-------------------------------------------------------------------%
function v = split_test(B, mindim, fun)
% THIS FUNCTION IS PART OF FUNCTION SPLIT-MERGE. IT DETERMINES
% WHETHER QUADREGIONS ARE SPLIT. The function returns in v
% logical 1s (TRUE) for the blocks that should be split and
% logical 0s (FALSE) for those that should not.

% Quadregion B, passed by qtdecomp, is the current decomposition of
% the image into k blocks of size m-by-m.

% k is the number of regions in B at this point in the procedure.
k = size(B, 3);

% Perform the split test on each block. If the predicate function
% (fun) returns TRUE, the region is split, so we set the appropriate
% element of v to TRUE. Else, the appropriate element of v is set to
% FALSE.
v(1:k) = false;
for I = 1:k
   quadregion = B(:, :, I);
   if size(quadregion, 1) <= mindim
      v(I) = false;
      continue
   end
   flag = feval(fun, quadregion);
   if flag
      v(I) = true;
   end
end
```

EXAMPLE 10.9: Image segmentation using region splitting and merging.

■ Figure 10.17(a) shows an X-ray band image of the Cygnus Loop. The image is of size 256 × 256 pixels. The objective of this example is to segment out of the image the "ring" of less dense matter surrounding the dense center. The region of interest has some obvious characteristics that should help in its segmentation. First, we note that the data has a random nature to it, indicating that its standard deviation should be greater than the standard deviation of the background (which is 0) and of the large central region. Similarly, the mean value (average intensity) of a region containing data from the outer ring should be greater than the mean of the background (which is 0) and less than the mean of the large, lighter central region. Thus, we should be able to segment the region of interest by using these two parameters. In fact, the predicate function shown as an example in the documentation of function `splitmerge` contains this knowledge about the problem. The parameters were determined by computing the mean and standard deviation of various regions in Fig. 10.17(a).

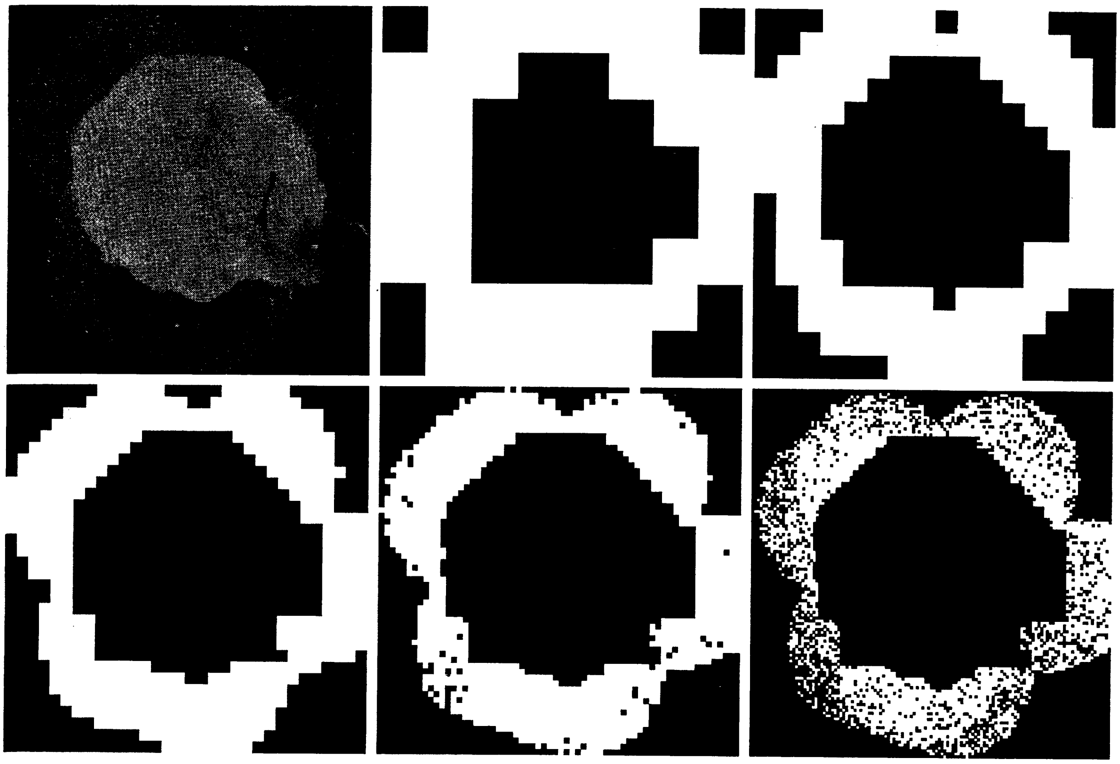


FIGURE 10.17 Image segmentation by a split-and-merge procedure. (a) Original image. (b) through (f) results of segmentation using function `splitmerge` with values of `mindim` equal to 32, 16, 8, 4, and 2, respectively. (Original image courtesy of NASA.)

Figures 10.17(b) through (f) show the results of segmenting Fig. 10.17(a) using function `splitmerge` with `mindim` values of 32, 16, 8, 4, and 2, respectively. All images show segmentation results with levels of detail that are inversely proportional to the value of `mindim`.

All results in Fig. 10.17 are reasonable segmentations. If one of these images were to be used as a mask to extract the region of interest out of the original image, then the result in Fig. 10.17(d) would be the best choice because it is the solid region with the most detail. An important aspect of the method just illustrated is its ability to "capture" in function `predicate` information about a problem domain that can help in segmentation. ■

10.5 Segmentation Using the Watershed Transform

In geography, a *watershed* is the ridge that divides areas drained by different river systems. A *catchment basin* is the geographical area draining into a river or reservoir. The *watershed transform* applies these ideas to gray-scale image processing in a way that can be used to solve a variety of image segmentation problems.

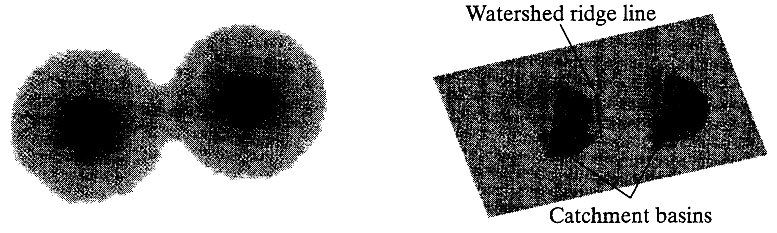


FIGURE 10.18 (a) Gray-scale image of dark blobs. (b) Image viewed as a surface, with labeled watershed ridge line and catchment basins.

Understanding the watershed transform requires that we think of a gray-scale image as a topological surface, where the values of $f(x, y)$ are interpreted as heights. We can, for example, visualize the simple image in Fig. 10.18(a) as the three-dimensional surface in Fig. 10.18(b). If we imagine rain falling on this surface, it is clear that water would collect in the two areas labeled as catchment basins. Rain falling exactly on the labeled watershed ridge line would be equally likely to collect in either of the two catchment basins. The watershed transform finds the catchment basins and ridge lines in a gray-scale image. In terms of solving image segmentation problems, the key concept is to change the starting image into another image whose catchment basins are the objects or regions we want to identify.

Methods for computing the watershed transform are discussed in detail in Gonzalez and Woods [2002] and in Soille [2003]. In particular, the algorithm used in IPT is adapted from Vincent and Soille [1991].

10.5.1 Watershed Segmentation Using the Distance Transform

A tool commonly used in conjunction with the watershed transform for segmentation is the *distance transform*. The distance transform of a binary image is a relatively simple concept: It is the distance from every pixel to the nearest nonzero-valued pixel. Figure 10.19 illustrates the distance transform. Figure 10.19(a) shows a small binary image matrix. Figure 10.19(b) shows the corresponding distance transform. Note that 1-valued pixels have a distance transform value of 0. The distance transform can be computed using IPT function `bwdist`, whose calling syntax is

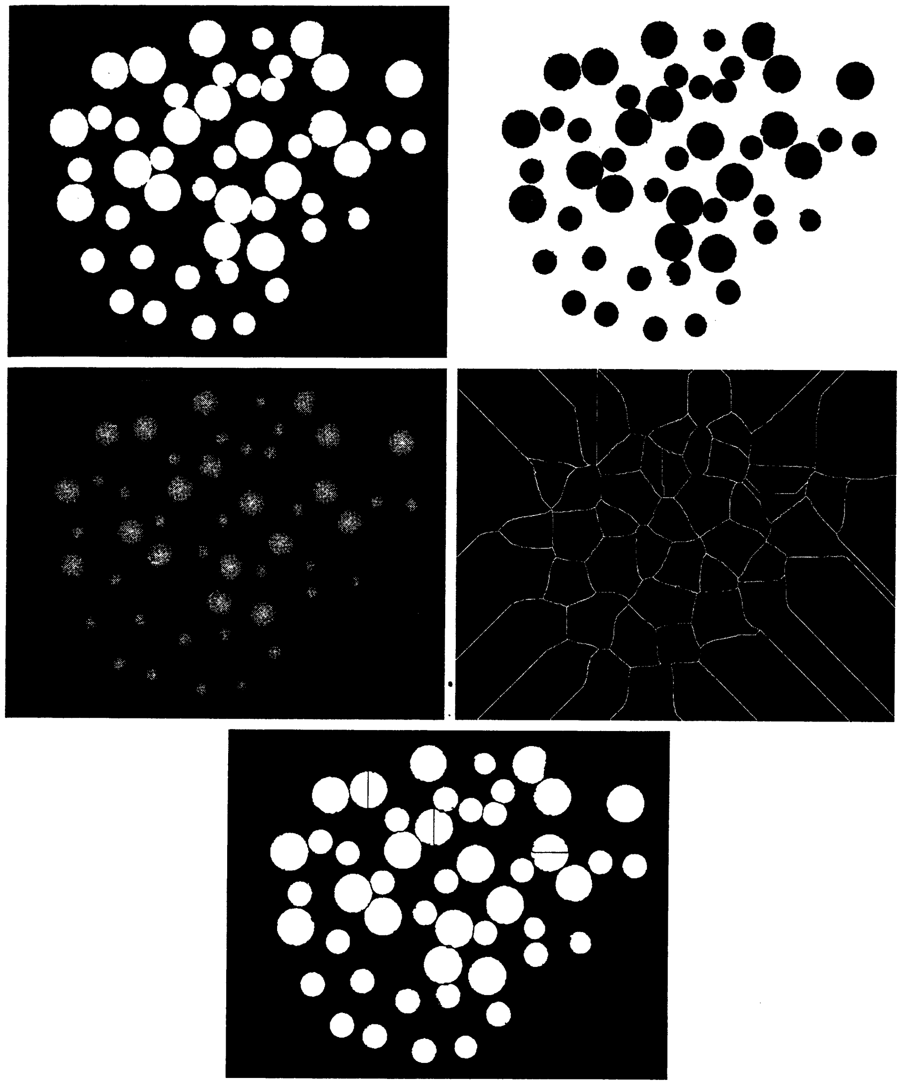
bwdist

```
D = bwdist(f)
```

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

0.00	0.00	1.00	2.00	3.00
0.00	0.00	1.00	2.00	3.00
1.00	1.00	1.41	2.00	2.24
1.41	1.00	1.00	1.00	1.41
1.00	0.00	0.00	0.00	1.00

FIGURE 10.19 (a) Small binary image. (b) Distance transform.



a b
c d
e

FIGURE 10.20
(a) Binary image.
(b) Complement of image in (a).
(c) Distance transform.
(d) Watershed ridge lines of the negative of the distance transform.
(e) Watershed ridge lines superimposed in black over original binary image. Some oversegmentation is evident.

EXAMPLE 10.10: Segmenting a binary image using the distance and watershed transforms.

■ In this example we show how the distance transform can be used with IPT's watershed transform to segment circular blobs, some of which are touching each other. Specifically, we want to segment the preprocessed dowel image, `f`, shown in Figure 9.29(b). First, we convert the image to binary using `im2bw` and `graythresh`, as described in Section 10.3.1.

```
>> g = im2bw(f, graythresh(f));
```

Figure 10.20(a) shows the result. The next steps are to complement the image, compute its distance transform, and then compute the watershed transform of the negative of the distance transform, using function `watershed`. The calling syntax for this function is

watershed

```
L = watershed(f)
```

where `L` is a label matrix, as defined and discussed in Section 9.4. Positive integers in `L` correspond to catchment basins, and zero values indicate watershed ridge pixels.

```
>> gc = ~g;
>> D = bwdist(gc);
>> L = watershed(-D);
>> w = L == 0;
```

Figures 10.20(b) and (c) show the complemented image and its distance transform. Since 0-valued pixels of `L` are watershed ridge pixels, the last line of the preceding code computes a binary image, `w`, that shows only these pixels. This watershed ridge image is shown in Fig. 10.20(d). Finally, a logical AND of the original binary image and the complement of `w` serves to complete the segmentation, as shown in Fig. 10.20(e).

```
>> g2 = g & ~w;
```

Note that some objects in Fig. 10.20(e) were split improperly. This is called *oversegmentation* and is a common problem with watershed-based segmentation methods. The next two sections discuss different techniques for overcoming this difficulty. ■

10.5.2 Watershed Segmentation Using Gradients

The gradient magnitude is used often to preprocess a gray-scale image prior to using the watershed transform for segmentation. The gradient magnitude image has high pixel values along object edges, and low pixel values everywhere else. Ideally, then, the watershed transform would result in watershed ridge lines along object edges. The next example illustrates this concept.

EXAMPLE 10.11: Segmenting a gray-scale image using gradients and the watershed transform.

■ Figure 10.21(a) shows an image, `f`, containing several dark blobs. We start by computing its gradient magnitude, using either the linear filtering methods described in Section 10.1, or using a morphological gradient as described in Section 9.6.1.

```
>> h = fspecial('sobel');
>> fd = double(f);
>> g = sqrt(imfilter(fd, h, 'replicate') .^ 2 + ...
           imfilter(fd, h', 'replicate') .^ 2);
```

Figure 10.21(b) shows the gradient magnitude image, `g`. Next we compute the watershed transform of the gradient and find the watershed ridge lines.

```
>> L = watershed(g);
>> wr = L == 0;
```

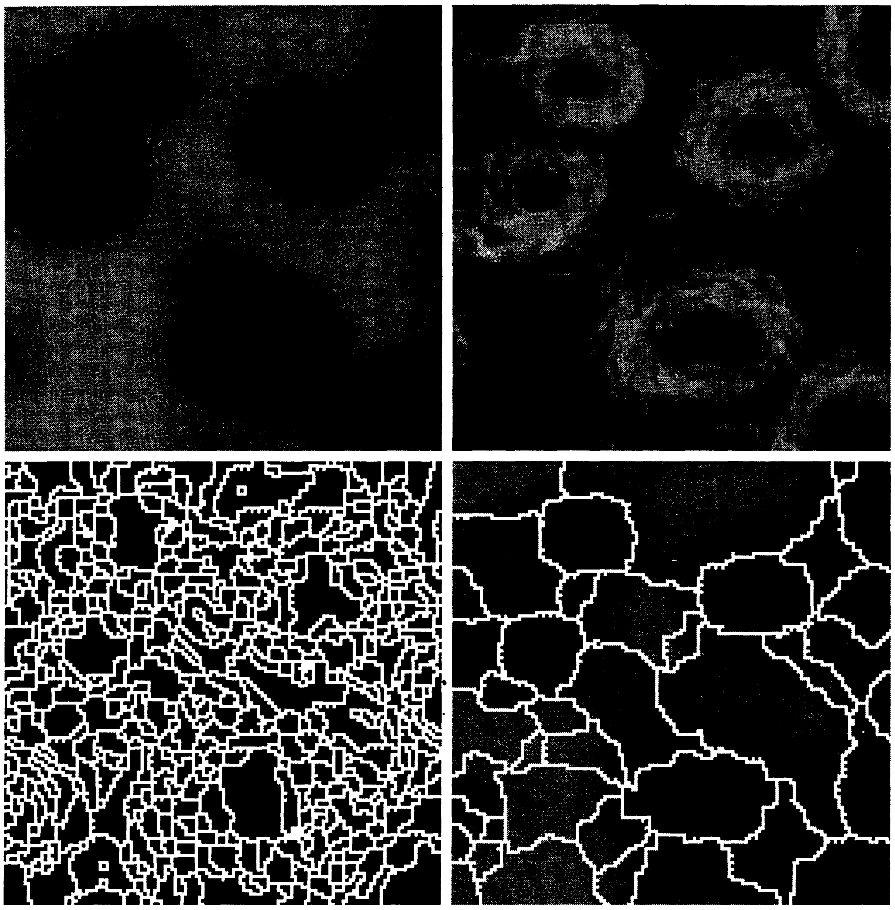


FIGURE 10.21 (a) Gray-scale image of small blobs. (b) Gradient magnitude image. (c) Watershed transform of (b), showing severe oversegmentation. (d) Watershed transform of the smoothed gradient image; some oversegmentation is still evident. (Original image courtesy of Dr. S. Beucher, CMM/Ecole de Mines de Paris.)

As Fig. 10.21(c) shows, this is not a good segmentation result; there are too many watershed ridge lines that do not correspond to the objects in which we are interested. This is another example of oversegmentation. One approach to this problem is to smooth the gradient image before computing its watershed transform. Here we use a close-opening, as described in Chapter 9.

```
>> g2 = imclose(imopen(g, ones(3,3)), ones(3,3));
>> L2 = watershed(g2);
>> wr2 = L2 == 0;
>> f2 = f;
>> f2(wr2) = 255;
```

The last two lines in the preceding code superimpose the watershed ridgelines in wr as white lines in the original image. Figure 10.21(d) shows the superimposed result. Although improvement over Fig. 10.21(c) was achieved, there are still some extraneous ridge lines, and it can be difficult to determine which catchment basins are actually associated with the objects of interest. The next section describes further refinements of watershed-based segmentation that deal with these difficulties. ■

10.5.3 Marker-Controlled Watershed Segmentation

Direct application of the watershed transform to a gradient image usually leads to oversegmentation due to noise and other local irregularities of the gradient. The resulting problems can be serious enough to render the result virtually useless. In the context of the present discussion, this means a large number of segmented regions. A practical solution to this problem is to limit the number of allowable regions by incorporating a preprocessing stage designed to bring additional knowledge into the segmentation procedure.

An approach used to control oversegmentation is based on the concept of markers. A *marker* is a connected component belonging to an image. We would like to have a set of *internal* markers, which are inside each of the objects of interest, as well as a set of *external* markers, which are contained within the background. These markers are then used to modify the gradient image using a procedure described in Example 10.12. Various methods have been used for computing internal and external markers, many of which involve the linear filtering, nonlinear filtering, and morphological processing described in previous chapters. Which method we choose for a particular application is highly dependent on the specific nature of the images associated with that application.

EXAMPLE 10.12: Illustration of marker-controlled watershed segmentation.

■ This example applies marker-controlled watershed segmentation to the electrophoresis gel image in Figure 10.22(a). We start by considering the results obtained from computing the watershed transform of the gradient image, without any other processing.

```
>> h = fspecial('sobel');
>> fd = double(f);
>> g = sqrt(imfilter(fd, h, 'replicate') .^ 2 + ...
            imfilter(fd, h', 'replicate') .^ 2);
>> L = watershed(g);
>> wr = L == 0;
```

We can see in Fig. 10.22(b) that the result is severely oversegmented, due in part to the large number of regional minima. IPT function `imregionalmin` computes the location of all regional minima in an image. Its calling syntax is

imregionalmin

```
rm = imregionalmin(f)
```

where `f` is a gray-scale image and `rm` is a binary image whose foreground pixels mark the locations of regional minima. We can use `imregionalmin` on the gradient image to see why the `watershed` function produces so many small catchment basins:

```
>> rm = imregionalmin(g);
```

Most of the regional minima locations shown in Fig. 10.22(c) are very shallow and represent detail that is irrelevant to our segmentation problem. To eliminate these extraneous minima we use IPT function `imextendedmin`,

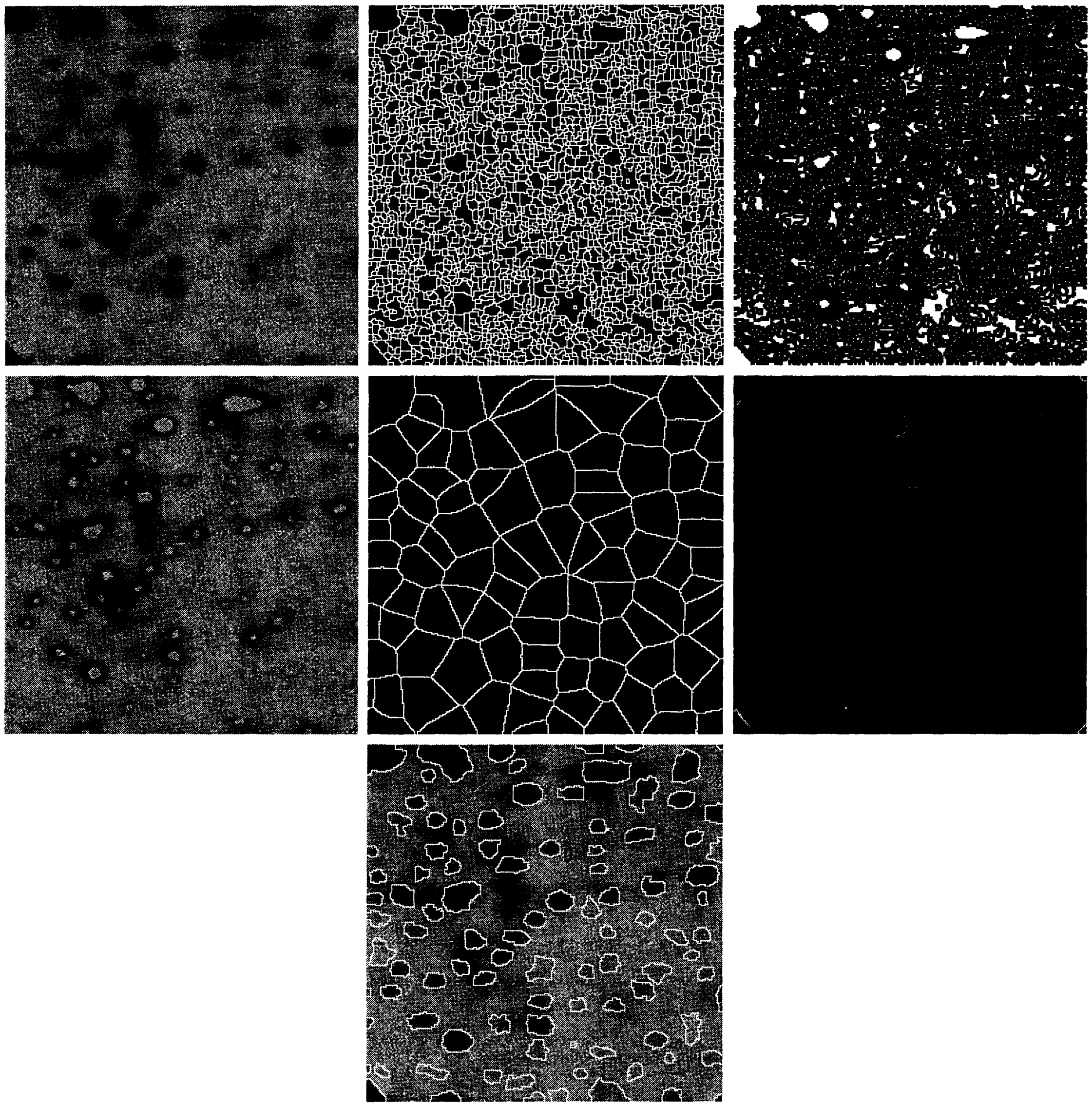


FIGURE 10.22 (a) Gel image. (b) Oversegmentation resulting from applying the watershed transform to the gradient magnitude image. (c) Regional minima of gradient magnitude. (d) Internal markers. (e) External markers. (f) Modified gradient magnitude. (g) Segmentation result. (Original image courtesy of Dr. S. Beucher, CMM/Ecole des Mines de Paris.)

which computes the set of "low spots" in the image that are deeper (by a certain height threshold) than their immediate surroundings. (See Soille [2003] for a detailed explanation of the *extended minima transform* and related operations.) The calling syntax for this function is

imextendedmin

```
im = imextendedmin(f, h)
```

where `f` is a gray-scale image, `h` is the height threshold, and `im` is a binary image whose foreground pixels mark the locations of the deep regional minima. Here we use function `imextendedmin` to obtain our set of internal markers:

```
>> im = imextendedmin(f, 2);
>> fim = f;
>> fim(im) = 175;
```

The last two lines superimpose the extended minima locations as gray blobs on the original image, as shown in Fig. 10.22(d). We see that the resulting blobs do a reasonably good job of "marking" the objects we want to segment.

Next we must find external markers, or pixels that we are confident belong to the background. The approach we follow here is to mark the background by finding pixels that are exactly midway between the internal markers. Surprisingly, we do this by solving another watershed problem; specifically, we compute the watershed transform of the distance transform of the internal marker image, `im`:

```
>> Lim = watershed(bwdist(im));
>> em = Lim == 0;
```

Figure 10.22(e) shows the resulting watershed ridge lines in the binary image `em`. Since these ridgelines are midway in between the dark blobs marked by `im`, they should serve well as our external markers.

Given both internal and external markers, we use them now to modify the gradient image using a procedure called *minima imposition*. The minima imposition technique (see Soille [2003] for details) modifies a gray-scale image so that regional minima occur only in marked locations. Other pixel values are "pushed up" as necessary to remove all other regional minima. IPT function `imimposemin` implements this technique. Its calling syntax is

imimposemin

```
mp = imimposemin(f, mask)
```

where `f` is a gray-scale image and `mask` is a binary image whose foreground pixels mark the desired locations of regional minima in the output image, `mp`. We modify the gradient image by imposing regional minima at the locations of both the internal and the external markers:

```
>> g2 = imimposemin(g, im | em);
```

Figure 10.22(f) shows the result. We are finally ready to compute the watershed transform of the marker-modified gradient image and look at the resulting watershed ridgelines:

```
>> L2 = watershed(g2);
>> f2 = f;
>> f2(L2 == 0) = 255;
```

The last two lines superimpose the watershed ridge lines on the original image. The result, a much-improved segmentation, is shown in Fig. 10.22(g). ■

Marker selection can range from the simple procedures just described to considerably more complex methods involving size, shape, location, relative distances, texture content, and so on (see Chapter 11 regarding descriptors). The point is that using markers brings a priori knowledge to bear on the segmentation problem. Humans often aid segmentation and higher-level tasks in everyday vision by using a priori knowledge, one of the most familiar being the use of context. Thus, the fact that segmentation by watersheds offers a framework that can make effective use of this type of knowledge is a significant advantage of this method.

Summary

Image segmentation is an essential preliminary step in most automatic pictorial pattern-recognition and scene analysis problems. As indicated by the range of examples presented in this chapter, the choice of one segmentation technique over another is dictated mostly by the particular characteristics of the problem being considered. The methods discussed in this chapter, although far from exhaustive, are representative of techniques used commonly in practice.