# CHAPTER 14

# EDGE DETECTION

## WHAT WILL WE LEARN?

- What is edge detection and why is it so important to computer vision?
- What are the main edge detection techniques and how well do they work?
- How can edge detection be performed in MATLAB?
- What is the Hough transform and how can it be used to postprocess the results of an edge detection algorithm?

## 14.1 FORMULATION OF THE PROBLEM

Edge detection is a fundamental image processing operation used in many computer vision solutions. The goal of edge detection algorithms is to find the most relevant edges in an image or scene. These edges should then be connected into meaningful lines and boundaries, resulting in a segmented image[1] containing two or more regions. Subsequent stages in a machine vision system will use the segmented results for tasks such as object counting, measuring, feature extraction, and classification.

The need for edge detection algorithms as part of a vision system also has its roots in biological vision: there is compelling evidence that the very early stages of the human visual system (HVS) contain edge-sensitive cells that respond strongly

[1] Image segmentation will be discussed in Chapter 15.

*Practical Image and Video Processing Using MATLAB®*. By Oge Marques.

(i.e., exhibit a higher firing rate) when presented with edges of certain intensity and orientation. Edge detection algorithms, therefore, attempt to emulate an ability present in the human visual system.

Such an ability is, according to many vision theorists, essential to all processing steps that take place afterward in the HVS. David Marr, in his very influential theory of vision [Mar82], speaks of *primal sketches*, which can be understood as the outcome of the very early steps in the transition from an image to a symbolic representation of its features, such as edges, lines, blobs, and terminations. In Marr's terminology, the result of edge detection algorithms compose the *raw* primal sketch; the outcome of additional processing, linking the resulting edges together, constitute the *full* primal sketch, which will be used by subsequent stages in the human visual processing system.

Edge detection is a hard image processing problem. Most edge detection solutions exhibit limited performance in the presence of images containing real-world scenes, that is, images that have not been carefully controlled in their illumination, size and position of objects, and contrast between objects and background. The impacts of shadows, occlusion among objects and parts of the scene, and noise—to mention just a few—on the results produced by an edge detection solution are often significant. Consequently, it is common to precede the edge detection stage with preprocessing operations such as noise reduction and illumination correction.

## 14.2 BASIC CONCEPTS

An *edge* can be defined as a boundary between two image regions having distinct characteristics according to some feature (e.g., gray level, color, or texture). In this chapter, we focus primarily on edges in grayscale 2D images, which are usually associated with a sharp variation of the intensity function across a portion of the image. Figure 14.1 illustrates this concept and shows the difference between an ideal edge (sharp and abrupt transition) and a ramp edge (gradual transition between dark and bright areas in the image).

Edge detection methods usually rely on calculations of the first or second derivative along the intensity profile. The first derivative has the desirable property of being directly proportional to the difference in intensity across the edge; consequently, the magnitude of the first derivative can be used to detect the presence of an edge at a certain point in the image. The sign of second derivative can be used to determine whether a pixel lies on the dark or on the bright side of an edge. Moreover, the zero crossing between its positive and negative peaks can be used to locate the center of thick edges.

Figure 14.2 illustrates these concepts. It shows an image with a ramp edge and the corresponding intensity profile, first and second derivatives, and zero crossing for any horizontal line in the image. It shows that the first derivative of the intensity function has a peak at the center of the luminance edge, whereas the second derivative—which is the slope of the first derivative function—has a zero crossing at the center

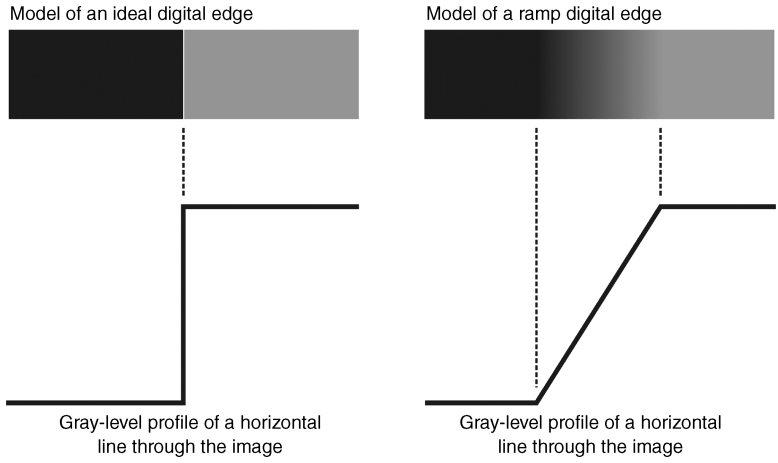



**FIGURE 14.1** Ideal and ramp edges: (a) ideal edge on a digital image and corresponding profile along a horizontal line; (b) ramp edge and corresponding profile.

of the luminance edge, with a positive value on one side and a negative value on the other.

The edges in Figures 14.1 and 14.2 were noise-free. When the input image is corrupted by noise, the first and second derivatives respond quite differently. Even modest noise levels—barely noticeable when you look at the original image or its profile—can render the second derivative results useless, whereas more pronounced noise levels will also impact the first derivative results to a point that they cannot be used for edge detection. Figure 14.3 illustrates this problem.

In summary, the process of edge detection consists of three main steps:

1. *Noise Reduction*: Due to the first and second derivative's great sensitivity to noise, the use of image smoothing techniques (see Chapters 10–12) before applying the edge detection operator is strongly recommended.
2. *Detection of Edge Points*: Here local operators that respond strongly to edges and weakly elsewhere (described later in this chapter) are applied to the image, resulting in an output image whose bright pixels are candidates to become edge points.
3. *Edge Localization*: Here the edge detection results are postprocessed, spurious pixels are removed, and broken edges are turned into meaningful lines and boundaries, using techniques such as the Hough transform (Section 14.6.1).

### In MATLAB

The IPT has a function for edge detection (`edge`), whose variants and options will be explored throughout this chapter and its tutorial.

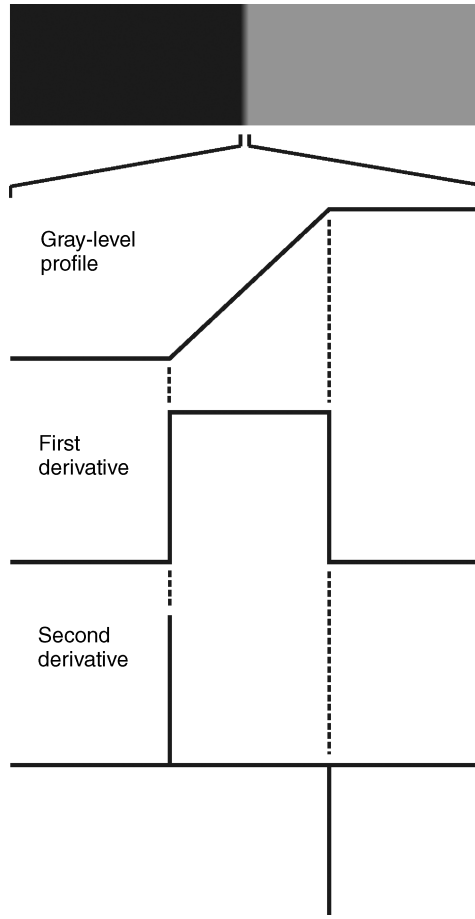



**FIGURE 14.2** Grayscale image containing two regions separated by a ramp edge: intensity profile and, first and second derivative results.

## 14.3 FIRST-ORDER DERIVATIVE EDGE DETECTION

The simplest edge detection methods work by estimating the gray-level gradient at a pixel, which can be approximated by the digital equivalent of the first-order derivative as follows:

$$g_x(x, y) \approx f(x + 1, y) - f(x - 1, y) \tag{14.1}$$

$$g_y(x, y) \approx f(x, y + 1) - f(x, y - 1) \tag{14.2}$$

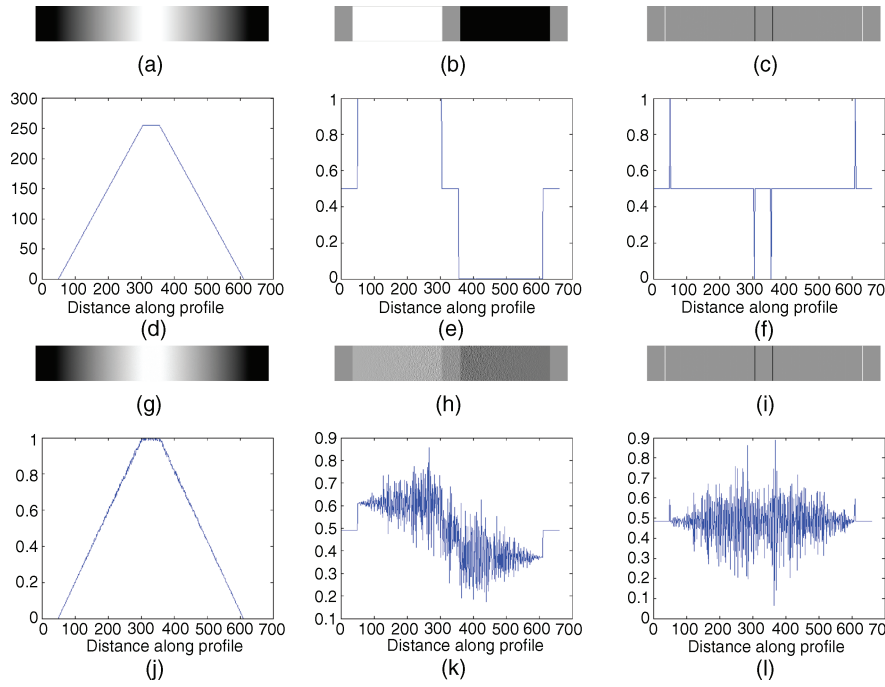



**FIGURE 14.3** First- and second-order edge detectors with and without noise: (a) original image; (b) first derivative; (c) second derivative; (d–f) horizontal profiles for images (a)–(c); (g–i) noisy versions of images (a)–(c); (j–l) horizontal profiles for images (g)–(i).

The $2 \times 2$ approximations of the first-order derivative above are also known as Roberts operators and can be represented using a $2 \times 2$ matrix notation as

$$g_x = \begin{bmatrix} 0 & -1 \\ 1 & 0 \end{bmatrix} \tag{14.3}$$

$$g_y = \begin{bmatrix} -1 & 0 \\ 0 & 1 \end{bmatrix} \tag{14.4}$$

These gradients are often computed within a $3 \times 3$ neighborhood using convolution:

$$g_x(x, y) = h_x * f(x, y) \tag{14.5}$$

$$g_y(x, y) = h_y * f(x, y) \tag{14.6}$$

where $h_x$ and $h_y$ are appropriate convolution masks (kernels).

The simplest pair of kernels, known as the Prewitt [Pre70] edge detector (operator), are as follows:

$$h_x = \begin{bmatrix} -1 & 0 & 1 \\ -1 & 0 & 1 \\ -1 & 0 & 1 \end{bmatrix} \tag{14.7}$$

$$h_y = \begin{bmatrix} -1 & -1 & -1 \\ 0 & 0 & 0 \\ 1 & 1 & 1 \end{bmatrix} \tag{14.8}$$

A similar pair of kernels, which gives more emphasis to on-axis pixels, is the Sobel edge detector, given by the following:

$$h_x = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} \tag{14.9}$$

$$h_y = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix} \tag{14.10}$$

As you may have noticed, despite their differences, the $3\times3$ masks presented so far share two properties:

- They have coefficients of opposite signs (across a row or column of coefficients equal to zero) in order to obtain a high response in image regions with variations in intensity (possibly due to the presence of an edge).
- The sum of the coefficients is equal to zero, which means that when applied to perfectly homogeneous regions in the image (i.e., a patch of the image with constant gray level), the result will be 0 (black pixel).

### In MATLAB

The IPT function `edge` has options for both Prewitt and Sobel operators. Edge detection using Prewitt and Sobel operators can also be achieved by using `imfilter` with the corresponding $3\times3$ masks (which can be created using `fspecial`).

### ■ EXAMPLE 14.1

Figure 14.4 shows an example of using `imfilter` to apply the Prewitt edge detector to a test image. Due to the fact that the Prewitt kernels have both positive and negative coefficients, the resulting array contains negative and positive values. Since negative

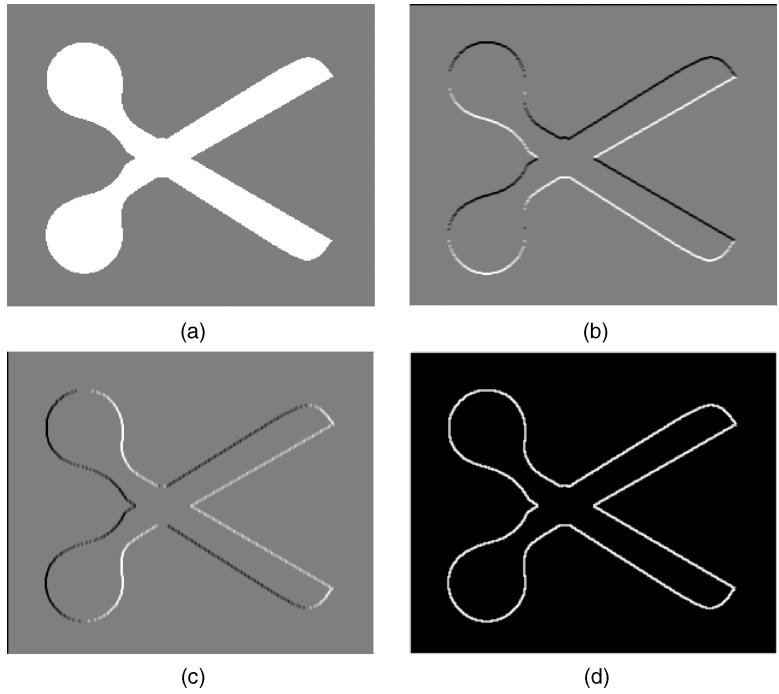



**FIGURE 14.4** Edge detection example: (a) original image; (b) result of Prewitt horizontal kernel; (c) result of Prewitt vertical kernel; (d) combination of (b) and (c).

values are usually truncated when displaying an image in MATLAB, the partial results (Figure 14.4b and c) have been mapped to a modified gray-level range (where the highest negative value becomes black, the highest positive value is displayed as white, and all zero values are shown with a midlevel gray). This display strategy also gives us insight into which side of the edge corresponds to a dark pixel and which side corresponds to a bright one. Moreover, it is worth observing that both detectors can find edges in this test image, despite the fact that it does not contain any purely vertical or horizontal edges of significant length.

The combined final result (Figure 14.4d) was obtained by computing the magnitude of the gradient, originally defined as

$$g = \sqrt{g_x^2 + g_y^2} \tag{14.11}$$

which can be approximated by

$$g = |g_x| + |g_y| \tag{14.12}$$

## ■ EXAMPLE 14.2

Figure 14.5 shows examples of edge detection using `imfilter` to apply the Sobel operator on a grayscale image. All results are displayed in their negative (using `imcomplement`) for better viewing on paper.

The idea of using horizontal and vertical masks used by the Prewitt and Sobel operators can be extended to include all eight compass directions: north, northeast, east, southeast, south, southwest, west, and northwest. The Kirsch [Kir71] (Figure 14.6) and the Robinson [Rob77] (Figure 14.7) kernels are two examples of *compass masks*. You will have a chance to design and apply the Kirsch and the Robinson masks to grayscale images in Tutorial 14.1.

Edge detection results can be thresholded to reduce the number of false positives (i.e., pixels that appear in the output image although they do not correspond to actual edges). Figure 14.8 shows an example of using `edge` to implement the Sobel operator with different threshold levels. The results range from unacceptable because of too many spurious pixels (part (a)) to unacceptable because of too few edge pixels (part (d)). Part (c) shows the result using the best threshold value, as determined by the

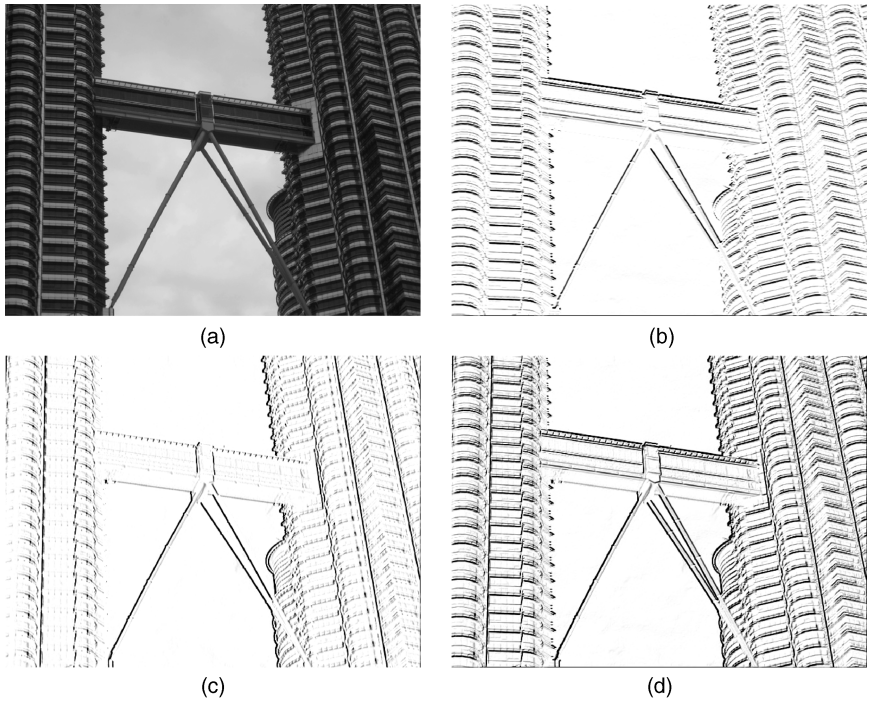



**FIGURE 14.5** Edge detection using Sobel operator: (a) original image; (b) result of Sobel horizontal kernel; (c) result of Sobel vertical kernel; (d) combination of (b) and (c).

$k_0 =$ | −3 | −3 | 5 |
|---|---|---|
| −3 | 0 | 5 |
| −3 | −3 | 5 |

$k_1 =$ | −3 | 5 | 5 |
|---|---|---|
| −3 | 0 | 5 |
| −3 | −3 | −3 |

$k_2 =$ | 5 | 5 | 5 |
|---|---|---|
| −3 | 0 | −3 |
| −3 | −3 | −3 |

$k_3 =$ | 5 | 5 | −3 |
|---|---|---|
| 5 | 0 | −3 |
| −3 | −3 | −3 |

$k_4 =$ | 5 | −3 | −3 |
|---|---|---|
| 5 | 0 | −3 |
| 5 | −3 | −3 |

$k_5 =$ | −3 | −3 | −3 |
|---|---|---|
| 5 | 0 | −3 |
| 5 | 5 | −3 |

$k_6 =$ | −3 | −3 | −3 |
|---|---|---|
| −3 | 0 | −3 |
| 5 | 5 | 5 |

$k_7 =$ | −3 | −3 | −3 |
|---|---|---|
| −3 | 0 | 5 |
| −3 | 5 | 5 |

**FIGURE 14.6** Kirsch compass masks.

`edge` function using the syntax: `[BW,thresh] = edge(I,'sobel');`, where `I` is the input image.

## 14.4 SECOND-ORDER DERIVATIVE EDGE DETECTION

The Laplacian operator (originally introduced in Section 10.4.1) is a straightforward digital approximation of the second-order derivative of the intensity. Although it has the potential for being employed as an isotropic (i.e., omnidirectional) edge detector, it is rarely used in isolation because of two limitations (commented earlier in this chapter):

- It generates "double edges," that is, positive and negative values for each edge.
- It is extremely sensitive to noise.

### In MATLAB

Edge detection using the Laplacian operator can be implemented using the `fspecial` function (to generate the Laplacian 3×3 convolution mask) and the

$r_0 =$ | −1 | 0 | 1 |
|---|---|---|
| −2 | 0 | 2 |
| −1 | 0 | 1 |

$r_1 =$ | 0 | 1 | 2 |
|---|---|---|
| −1 | 0 | 1 |
| −2 | 1 | 0 |

$r_2 =$ | 1 | 2 | 1 |
|---|---|---|
| 0 | 0 | 0 |
| −1 | −2 | −1 |

$r_3 =$ | 2 | 1 | 0 |
|---|---|---|
| 1 | 0 | −1 |
| 0 | −1 | −2 |

$r_4 =$ | 1 | 0 | −1 |
|---|---|---|
| 2 | 0 | −2 |
| 1 | 0 | −1 |

$r_5 =$ | 0 | −1 | −2 |
|---|---|---|
| 1 | 0 | −1 |
| 2 | 1 | 0 |

$r_6 =$ | −1 | −2 | −1 |
|---|---|---|
| 0 | 0 | 0 |
| 1 | 2 | 1 |

$r_7 =$ | −2 | −1 | 0 |
|---|---|---|
| −1 | 0 | 1 |
| 0 | 1 | 2 |

**FIGURE 14.7** Robinson compass masks.

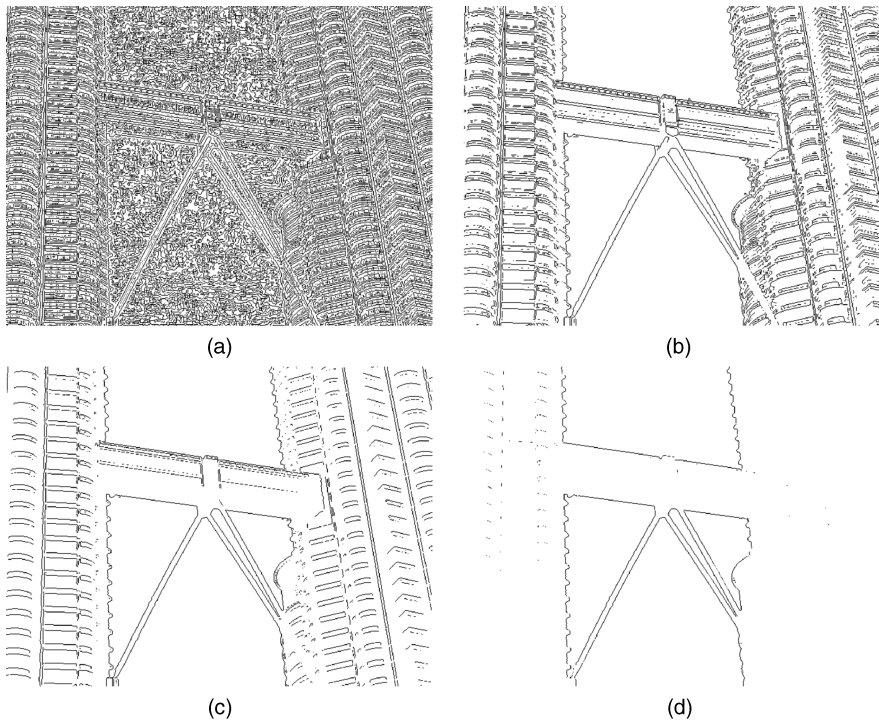



**FIGURE 14.8** Edge detection using Sobel operator and thresholding (the original image is the same as Figure 14.5a): (a) threshold of 0; (b) threshold of 0.05; (c) threshold of 0.1138 (the best value); (d) threshold of 0.2.

`zerocross` option in function `edge` as follows:

```
h = fspecial('laplacian',0);
J = edge(I,'zerocross',t,h);
```

where `t` is a user-provided sensitivity threshold.

## ■ EXAMPLE 14.3

Figure 14.9 shows the results of applying the zero-cross edge detector to an image and the impact of varying the thresholds. Part (a) shows a clean input image; part (b) shows the results of edge detection in (a) using default parameters, whereas part (c) shows the effects of reducing the threshold to 0. Clearly the result in (b) is much better than (c).

Part (d) is a noisy version of (a) (with zero-mean Gaussian noise with $\sigma = 0.0001$). Parts (e) and (f) are the edge detection results using the noisy image in part (d) as an input and the same options as in (b) and (c), respectively. In this case, although the amount of noise is hardly noticeable in the original image (d), both edge detection results are unacceptable.

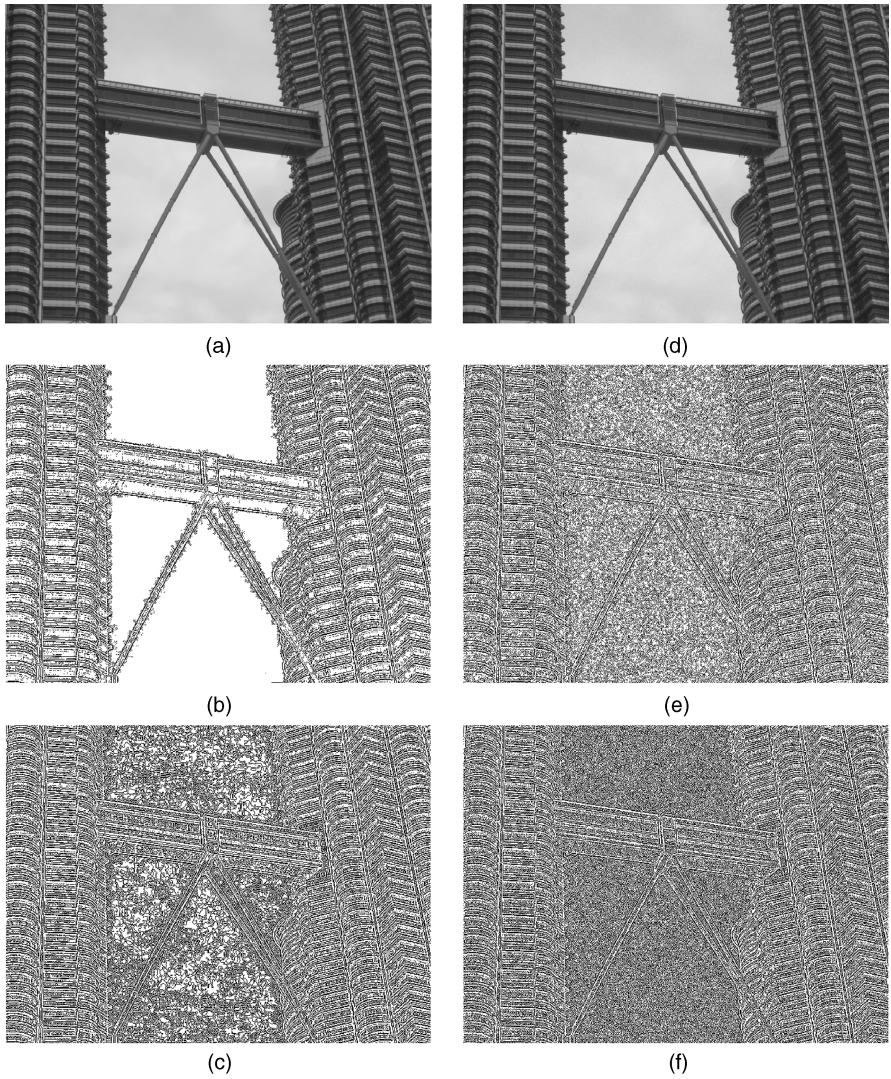



**FIGURE 14.9** Edge detection using the zero-cross edge detector: (a) input image (without noise); (b) results using default values; (c) results using threshold zero; (d) noisy input image; (e) results using default values; (f) results using threshold zero. Edge results have been inverted for clarity.

### 14.4.1 Laplacian of Gaussian

The Laplacian of Gaussian (LoG) edge detector works by smoothing the image with a Gaussian low-pass filter (LPF), and then applying a Laplacian edge detector to the result. The resulting transfer function (which resembles a Mexican hat in its 3D view) is represented in Figure 14.10. The LoG filter can sometimes be approximated by

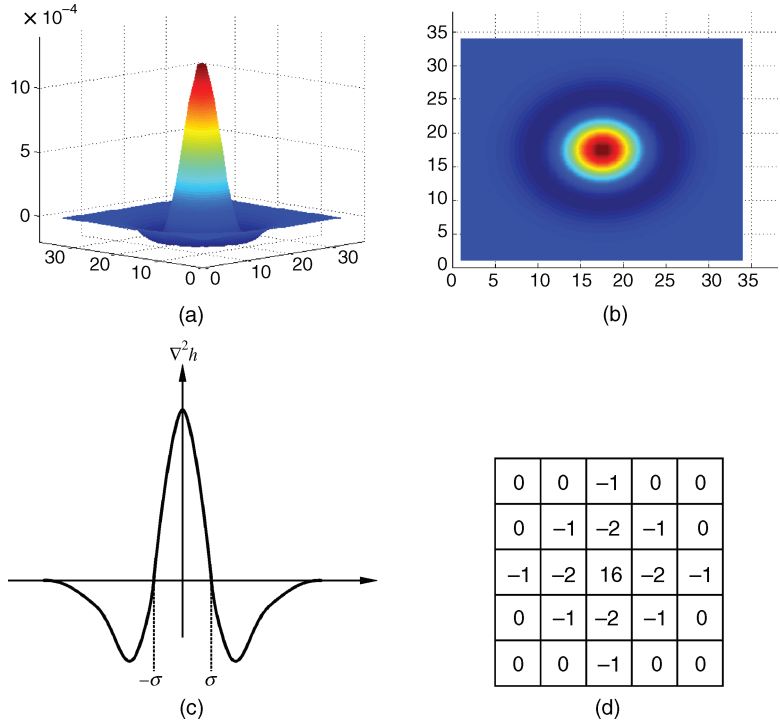



**FIGURE 14.10** Laplacian of Gaussian: (a) 3D plot; (b) 2D intensity plot; (c) cross section of (a).

taking the differences of two Gaussians of different widths in a method known as *difference of Gaussians* (DoG).

In a landmark paper [MH80], David Marr and Ellen Hildreth proposed that LoG filtering explains much of the low-level behavior of the human vision system, since the response profile of an LoG filter approximates the receptive field of retinal cells tuned to respond to edges. They proposed an architecture based on LoG filters with four or five different spreads of $\sigma$ to derive a primal sketch from a scene. The Marr–Hildreth zero-crossing algorithm was eventually supplanted by the Canny edge detector as the favorite edge detection solution among image processing and computer vision practitioners, but Marr's ideas continue to influence researchers in both human as well as computer vision.

## In MATLAB

Edge detection using the LoG filter can be implemented using the `log` option in function `edge`.

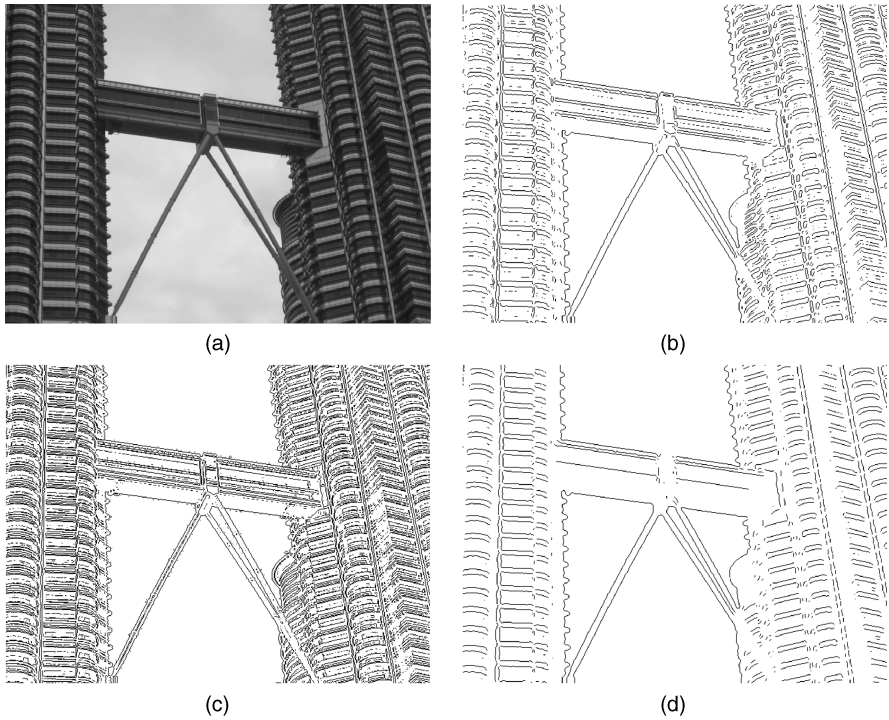



**FIGURE 14.11** Edge detection using the LoG edge detector: (a) input image; (b) results using default values; (c) results using $\sigma = 1$; (d) results using $\sigma = 3$. Edge results have been inverted for clarity.

### ■ EXAMPLE 14.4

Figure 14.11 shows the results of applying the LoG edge detector to an image, and the impact of varying $\sigma$. Part (a) shows the input image; part (b) shows the results of edge detection in (a) using default parameters (i.e., $\sigma = 2$), whereas parts (c) and (d) show the effects of reducing or increasing sigma (to 1 and 3, respectively). Reducing $\sigma$ causes the resulting image to contain more fine details, whereas an increase in $\sigma$ leads to a coarser edge representation, as expected.

## 14.5 THE CANNY EDGE DETECTOR

The Canny edge detector [Can86] is one of the most popular, powerful, and effective edge detection operators available today. Its algorithm can be described as follows:

1. The input image is smoothed using a Gaussian low-pass filter (Section 10.3.3), with a specified value of $\sigma$: large values of $\sigma$ will suppress much of the noise at the expense of weakening potentially relevant edges.

2. The local gradient (intensity and direction) is computed for each point in the smoothed image.
3. The edge points at the output of step 2 result in wide ridges. The algorithm thins those ridges, leaving only the pixels at the top of each ridge, in a process known as *nonmaximal suppression*.
4. The ridge pixels are then thresholded using two thresholds $T_{\text{low}}$ and $T_{\text{high}}$: ridge pixels with values greater than $T_{\text{high}}$ are considered *strong* edge pixels; ridge pixels with values between $T_{\text{low}}$ and $T_{\text{high}}$ are said to be *weak* pixels. This process is known as *hysteresis thresholding*.
5. The algorithm performs edge linking, aggregating weak pixels that are 8-connected[2] to the strong pixels.

### In MATLAB

The `edge` function includes the Canny edge detector, which can be invoked using the following syntax:

```
J = edge(I, 'canny', T, sigma);
```

where `I` is the input image, `T = [T_low T_high]` is a $1 \times 2$ vector containing the two thresholds explained in step 4 of the algorithm, `sigma` is the standard deviation of the Gaussian smoothing filter, and `J` is the output image.

### ■ EXAMPLE 14.5

Figure 14.12 shows the results of applying the Canny detector to an image (Figure 14.5a), and the impact of varying $\sigma$ and the thresholds. Part (a) uses the syntax `BW = edge(J,'canny');`, which results in `t = [0.0625 0.1563]` and $\sigma = 1$. In part (b), we change the value of $\sigma$ (to 0.5) leaving everything else unchanged. In part (c), we change the value of $\sigma$ (to 2) leaving everything else unchanged. Changing $\sigma$ causes the resulting image to contain more (part (b)) or fewer (part (c)) edge points (compared to part (a)), as expected. Finally, in part (d), we keep $\sigma$ in its default value and change the thresholds to `t = [0.01 0.1]`. Since both $T_{\text{low}}$ and $T_{\text{high}}$ were lowered, the resulting image contains more strong and weak pixels, resulting in a larger number of edge pixels (compared to part (a)), as expected.

## 14.6 EDGE LINKING AND BOUNDARY DETECTION

The goal of edge detection algorithms should be to produce an image containing only the edges of the original image. However, due to the many technical challenges discussed earlier (noise, shadows, and occlusion, among others), most edge detection algorithms will output an image containing fragmented edges. In order to turn

[2]In some implementations, only the neighbors along a line normal to the gradient orientation at the edge pixel are considered, not the entire 8-neighborhood.

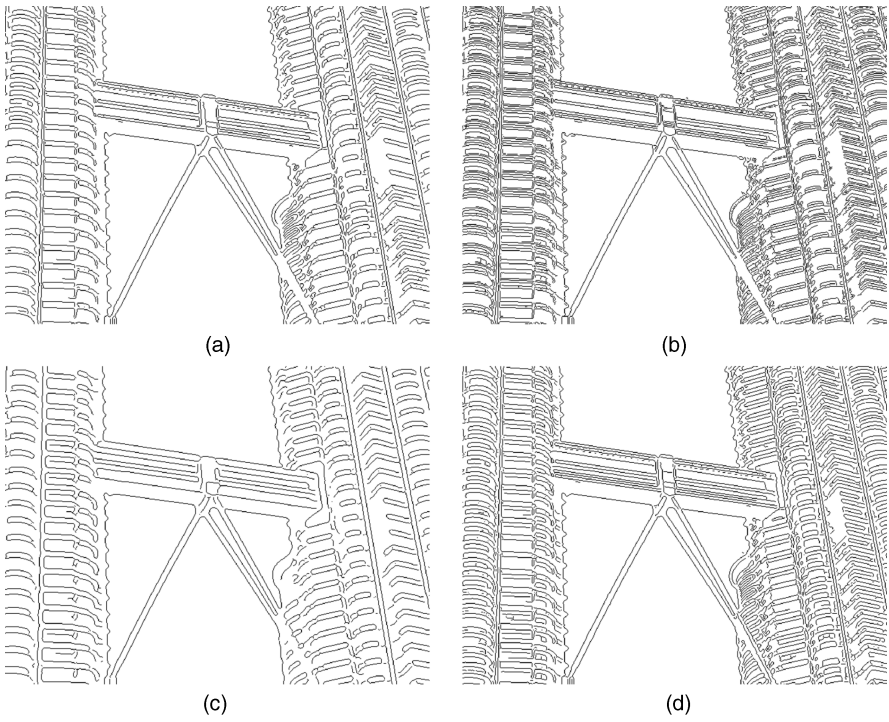



**FIGURE 14.12** Edge detection using the Canny edge detector: (a) default values ($\sigma = 1$, $T_{\text{low}} = 0.0625$, $T_{\text{high}} = 0.1563$); (b) $\sigma = 0.5$; (c) $\sigma = 2$; (d) $\sigma = 1$, $T_{\text{low}} = 0.01$, $T_{\text{high}} = 0.1$.

these fragmented edge segments into useful lines and object boundaries, additional processing is needed. In this section, we discuss a global method for edge linking and boundary detection: the Hough transform.[3]

### 14.6.1 The Hough Transform

The Hough transform [Hou] is a mathematical method designed to find lines in images. It can be used for linking the results of edge detection, turning potentially sparse, broken, or isolated edges into useful lines that correspond to the actual edges in the image.

Let $(x, y)$ be the coordinates of a point in a binary image.[4] The Hough transform stores in an *accumulator array* all pairs $(a, b)$ that satisfy the equation $y = ax + b$. The $(a, b)$ array is called the *transform array*. For example, the point $(x, y) = (1, 3)$ in the input image will result in the equation $b = -a + 3$, which can be plotted as a line that represents all pairs $(a, b)$ that satisfy this equation (Figure 14.13).

[3]Pointers to other edge linking and boundary detection techniques can be found in "Learn More About It" section at the end of the chapter.

[4]This binary image could, of course, be an image containing thresholded edge detection results.

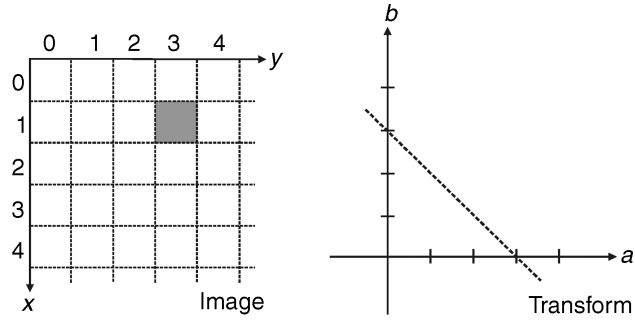



**FIGURE 14.13** The Hough transform maps a point into a line.

Since each point in the image will map to a line in the transform domain, repeating the process for other points will result in many intersecting lines, one per point (Figure 14.14). The meaning of two or more lines intersecting in the transform domain is that the points to which they correspond are aligned in the image. The points with the greatest number of intersections in the transform domain correspond to the longest lines in the image.

Describing lines using the equation $y = ax + b$ (where $a$ represents the gradient) poses a problem, since vertical lines have infinite gradient. This limitation can be circumvented by using the *normal representation* of a line, which consists of two parameters: $\rho$ (the perpendicular distance from the line to the origin) and $\theta$ (the angle between the line's perpendicular and the horizontal axis). In this new representation (Figure 14.15), vertical lines will have $\theta = 0$. It is common to allow $\rho$ to have negative values, therefore restricting $\theta$ to the range $-90^\circ < \theta \leq 90^\circ$.

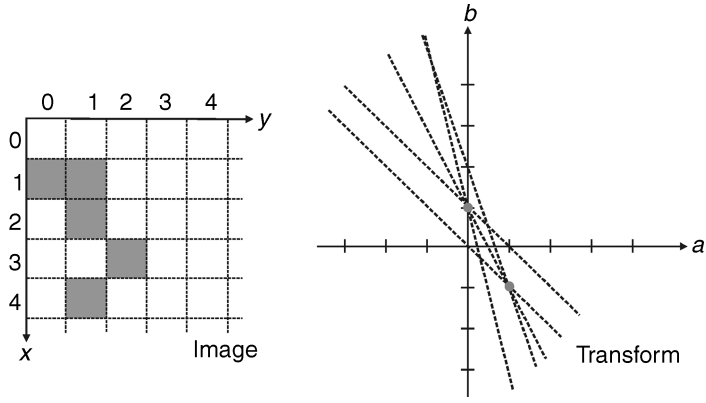



**FIGURE 14.14** The Hough transform: intersections in the transform domain correspond to aligned points in the image.

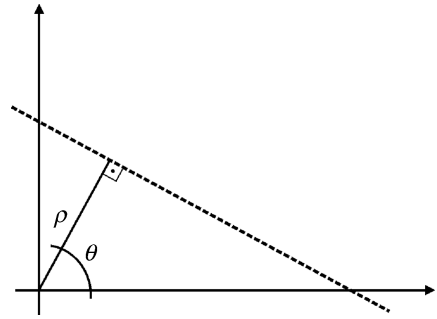



**FIGURE 14.15** The Hough transform: a line and its parameters in the polar coordinate system.

The relationship between $\rho$, $\theta$, and the original coordinates $(x, y)$ is

$$\rho = x\cos\theta + y\sin\theta \tag{14.13}$$

Under the new set of coordinates, the Hough transform can be implemented as follows:

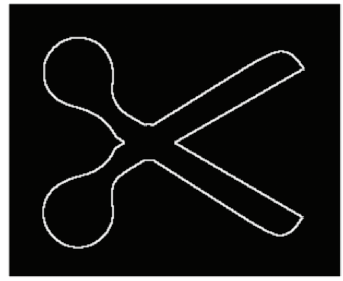
1. Create a 2D array corresponding to a discrete set of values for $\rho$ and $\theta$. Each element in this array is often referred to as an *accumulator cell*.
2. For each pixel $(x, y)$ in the image and for each chosen value of $\theta$, compute $x\cos\theta + y\sin\theta$ and write the result in the corresponding position—$(\rho, \theta)$—in the accumulator array.
3. The highest values in the $(\rho, \theta)$ array will correspond to the most relevant lines in the image.

## IN MATLAB

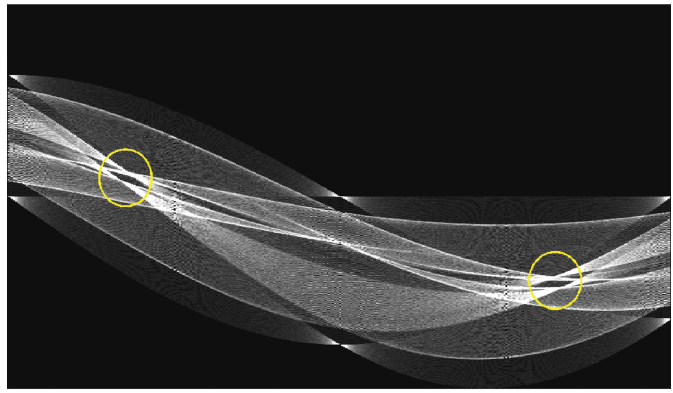
The IPT contains a function for Hough transform calculations, `hough`, which takes a binary image as an input parameter, and returns the corresponding Hough transform matrix and the arrays of $\rho$ and $\theta$ values over which the Hough transform was calculated. Optionally, the resolution of the discretized 2D array for both $\rho$ and $\theta$ can be specified as additional parameters.

### ■ EXAMPLE 14.6

In this example, we use the `hough` function to find the strongest lines in a binary image obtained as a result of an edge detection operator (`BW`), using the



(a)



(b)

**FIGURE 14.16** Hough transform example: (a) input image; (b) results of Hough transform, highlighting the intersections corresponding to the predominant lines in the input image.

following steps:

```
[H,T,R] = hough(BW,'RhoResolution',0.5,'ThetaResolution',0.5);
```

Figure 14.16 shows the original image and the results of the Hough transform calculations. You will notice that some of the highest peaks in the transform image (approximately at $\theta = -60^\circ$ and $\theta = 60^\circ$) correspond to the main diagonal lines in the scissors shape.

## IN MATLAB

The IPT also includes two useful companion functions for exploring and plotting the results of Hough transform calculations: `houghpeaks` (which identifies the $k$ most salient peaks in the Hough transform results, where $k$ is passed as a parameter) and `houghlines`, which draws the lines associated with the highest peaks on top of the original image.

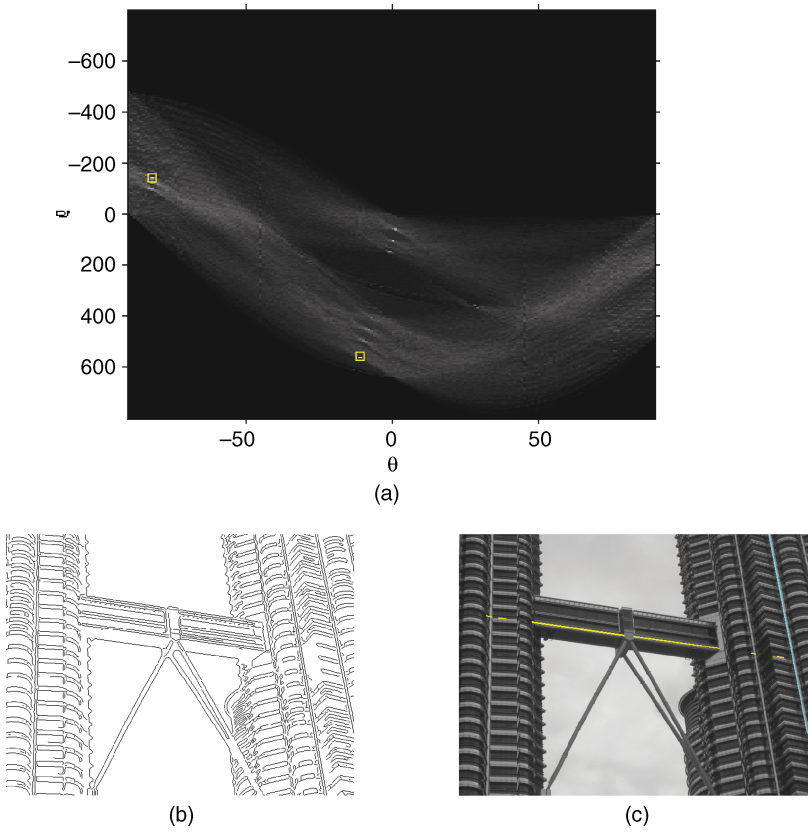



**FIGURE 14.17** Hough transform example: (a) results of Hough transform highlighting the two highest peaks; (b) (negative of) edge detection results; (c) lines corresponding to the longest peaks overlaid on top of original image.

## ■ EXAMPLE 14.7

In this example, we use `hough`, `houghpeaks`, and `houghlines` on a grayscale test image whose edges have been extracted using the Canny edge detector.[5]

Figure 14.17a shows the results of the Hough transform calculations with two small squares indicating the two highest peaks. Figure 14.17b shows the result of the Canny edge detector (displayed with black pixels against a white background for better visualization). Figure 14.17c displays the original image with the highest (cyan) and second highest (yellow) lines overlaid.

The Hough transform can be extended and generalized to find other shapes in images. Refer to "Learn More About It" section at the end of the chapter for useful references.

[5]The complete sequence of MATLAB commands is available at the book web site.

## 14.7 TUTORIAL 14.1: EDGE DETECTION

### Goal

The goal of this tutorial is to learn how to implement edge detection and associated techniques in MATLAB.

### Objectives

- Learn how to use the IPT `edge` function.
- Explore the most popular first-derivative edge detectors: Roberts, Sobel, and Prewitt.
- Explore the Marr–Hildreth Laplacian of Gaussian edge detector.
- Explore the Canny edge detector.
- Learn how to implement edge detection with compass masks (Kirsch and Robinson).

### What You Will Need

- `lenna.tif`
- `mandrill.tif`

### Procedure

First-order edge detection methods, such as Prewitt and Sobel, are defined as convolution kernels. As we have seen in previous tutorials, convolution can be executed in MATLAB using the `imfilter` function. Although we could use this function to implement edge detection, we would be required to perform additional tasks, such as convolving the image twice (once for horizontal edges, and another for vertical ones) and adding the absolute values of these results to yield the final image. In practice, this image might also then be thresholded to produce a binary image where white pixels would represent edges. The function `edge` will do all this for us and will even determine a threshold value if we choose not to specify one.

### Edge Detection Using the Prewitt Operator

1. Load and display the test image.

```
I = imread('lenna.tif');
figure, subplot(2,2,1), imshow(I), title('Original Image');
```

2. Extract the edges in the image using the Prewitt operator.

```
[I_prw1,t1] = edge(I,'prewitt');
subplot(2,2,2), imshow(I_prw1), title('Prewitt, default thresh');
```

***Question 1*** What does the `t1` variable represent?

Edge detection methods are often compared by their ability to detect edges in noisy images. Let us perform the Prewitt operator on the Lenna image with additive Gaussian noise.

3. Add noise to the test image and extract its edges.

```
I_noise = imnoise(I,'gaussian');
[I_prw2,t2] = edge(I_noise,'prewitt');
subplot(2,2,3), imshow(I_noise), title('Image w/ noise');
subplot(2,2,4), imshow(I_prw2), title('Prewitt on noise');
```

***Question 2*** How did the Prewitt edge detector perform in the presence of noise (compared to no noise)?

***Question 3*** Did MATLAB use a different threshold value for the noisy image?

***Question 4*** Try using different threshold values. Do these different values affect the operator's response to noise? How does the threshold value affect the edges of the object?

### Edge Detection Using the Sobel Operator

4. Extract the edges from the test image using the Sobel edge detector.

```
[I_sob1,t1] = edge(I,'sobel');
figure, subplot(2,2,1), imshow(I), title('Original Image');
subplot(2,2,2), imshow(I_sob1), title('Sobel, default thresh');
```

5. Extract the edges from the test image with Gaussian noise using the Sobel edge detector.

```
[I_sob2,t2] = edge(I_noise,'sobel');
subplot(2,2,3), imshow(I_noise), title('Image w/ noise');
subplot(2,2,4), imshow(I_sob2), title('Sobel on noise');
```

***Question 5*** How does the Sobel operator compare with the Prewitt operator with and without noise?

Another feature of the edge function is `'thinning'`, which reduces the thickness of the detected edges. Although this feature is turned on by default, it can be turned off, which results in faster edge detection.

6. Extract the edges from the test image with the Sobel operator with no thinning.

```
I_sob3 = edge(I,'sobel','nothinning');
figure, subplot(1,2,1), imshow(I_sob1), title('Thinning');
subplot(1,2,2), imshow(I_sob3), title('No Thinning');
```

As you already know, the Sobel operator actually performs two convolutions (horizontal and vertical). These individual images can be obtained by using additional output parameters.

7. Display the horizontal and vertical convolution results from the Sobel operator.

```
[I_sob4,t,I_sobv,I_sobh] = edge(I,'sobel');
figure
subplot(2,2,1), imshow(I), title('Original Image');
subplot(2,2,2), imshow(I_sob4), title('Complete Sobel');
subplot(2,2,3), imshow(abs(I_sobv),[]), title('Sobel Vertical');
subplot(2,2,4), imshow(abs(I_sobh),[]), title('Sobel Horizontal');
```

***Question 6*** Why do we display the absolute value of the vertical and horizontal images? *Hint*: Inspect the minimum and maximum values of these images.

***Question 7*** Change the code in step 7 to display thresholded (binarized), not thinned, versions of all images.

As you may have noticed, the `edge` function returns the vertical and horizontal images before any thresholding takes place.

### Edge Detection with the Roberts Operator

Similar options are available with the `edge` function when the Roberts operator is used.

8. Extract the edges from the original image using the Roberts operator.

```
I_rob1 = edge(I,'roberts');
figure
subplot(2,2,1), imshow(I), title('Original Image');
subplot(2,2,2), imshow(I_rob1), title('Roberts, default thresh');
```

9. Apply the Roberts operator to a noisy image.

```
[I_rob2,t] = edge(I_noise,'roberts');
subplot(2,2,3), imshow(I_noise), title('Image w/ noise');
subplot(2,2,4), imshow(I_rob2), title('Roberts on noise');
```

***Question 8*** Compare the Roberts operator with the Sobel and Prewitt operators. How does it hold up to noise?

***Question 9*** If we were to adjust the threshold, would we get better results when filtering the noisy image?

***Question 10*** Suggest a method to reduce the noise in the image before performing edge detection.

### Edge Detection with the Laplacian of a Gaussian Operator

The LoG edge detector can be implemented with the `edge` function as well. Let us see its results.

10. Extract the edges from the original image using the LoG edge detector.

```
I_log1 = edge(I,'log');
figure
subplot(2,2,1), imshow(I), title('Original Image');
subplot(2,2,2), imshow(I_log1), title('LoG, default parameters');
```

11. Apply the LoG edge detector to the noisy image.

```
[I_log2,t] = edge(I_noise,'log');
subplot(2,2,3), imshow(I_noise), title('Image w/ noise');
subplot(2,2,4), imshow(I_log2), title('LoG on noise');
```

***Question 11*** By default, the LoG edge detector uses a value of 2 for $\sigma$ (the standard deviation of the filter). What happens when we increase this value?

### Edge Detection with the Canny Operator

12. Extract the edges from the original image using the Canny edge detector.

```
I_can1 = edge(I,'canny');
figure
subplot(2,2,1), imshow(I), title('Original Image');
subplot(2,2,2), imshow(I_log1), title('Canny, default parameters');
```

13. Apply the filter to the noisy image.

```
[I_can2,t] = edge(I_noise,'canny', [], 2.5);
subplot(2,2,3), imshow(I_noise), title('Image w/ noise');
subplot(2,2,4), imshow(I_can2), title('Canny on noise');
```

As you know, the Canny detector first applies a Gaussian smoothing function to the image, followed by edge enhancement. To achieve better results on the noisy image, we can increase the size of the Gaussian smoothing filter through the `sigma` parameter.

14. Apply the Canny detector on the noisy image where `sigma` = 2.

```
[I_can3,t] = edge(I_noise,'canny', [], 2);
figure
subplot(1,2,1), imshow(I_can2), title('Canny, default parameters');
subplot(1,2,2), imshow(I_can3), title('Canny, sigma = 2');
```

***Question 12*** Does increasing the value of `sigma` give us better results when using the Canny detector on a noisy image?

Another parameter of the Canny detector is the threshold value, which affects the sensitivity of the detector.

15. Close any open figures and clear all workspace variables.
16. Load the `mandrill` image and perform the Canny edge detector with default parameters.

```
I = imread('mandrill.tif');
[I_can1,thresh] = edge(I,'canny');
figure
subplot(2,2,1), imshow(I), title('Original Image');
subplot(2,2,2), imshow(I_can1), title('Canny, default parameters');
```

17. Inspect the contents of variable `thresh`.
18. Use a threshold value higher than the one in variable `thresh`.

```
[I_can2,thresh] = edge(I, 'canny', 0.4);
subplot(2,2,3), imshow(I_can2), title('Canny, thresh = 0.4');
```

19. Use a threshold value lower than the one in variable `thresh`.

```
[I_can2,thresh] = edge(I, 'canny', 0.08);
subplot(2,2,4), imshow(I_can2), title('Canny, thresh = 0.08');
```

***Question 13*** How does the sensitivity of the Canny edge detector change when the threshold value is increased?

### Edge Detection with the Kirsch Operator

The remaining edge detection techniques discussed in this tutorial are not included in the current implementation of the `edge` function, so we must implement them as they are defined. We will begin with the Kirsch operator.

20. Close any open figures and clear all workspace variables.
21. Load the `mandrill` image and convert it to `double` format.

```
I = imread('mandrill.tif');
I = im2double(I);
```

Previously, when we were using the `edge` function, we did not need to convert the image to class `double` because the function took care of this for us automatically. Since now we are implementing the remaining edge detectors on our own, we must perform the class conversion to properly handle negative values (preventing unwanted truncation).

Next we will define the eight Kirsch masks. For ease of implementation, we will store all eight masks in a $3 \times 3 \times 8$ matrix. Figure 14.18 illustrates this storage format.

22. Create the Kirsch masks and store them in a preallocated matrix.

```
k = zeros(3,3,8);
k(:,:,1) = [-3 -3 5; -3 0 5; -3 -3 5];
k(:,:,2) = [-3 5 5; -3 0 5; -3 -3 -3];
k(:,:,3) = [5 5 5; -3 0 -3; -3 -3 -3];
k(:,:,4) = [5 5 -3; 5 0 -3; -3 -3 -3];
k(:,:,5) = [5 -3 -3; 5 0 -3; 5 -3 -3];
k(:,:,6) = [-3 -3 -3; 5 0 -3; 5 5 -3];
k(:,:,7) = [-3 -3 -3; -3 0 -3; 5 5 5];
k(:,:,8) = [-3 -3 -3; -3 0 5; -3 5 5];
```

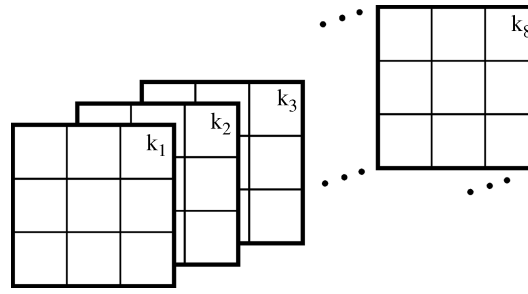



**FIGURE 14.18** Kirsch masks stored in a $3 \times 3 \times 8$ matrix.

Next we must convolve each mask on the image, generating eight images. We will store these images in a three-dimensional matrix just as we did for the masks. Because all the masks are stored in one matrix, we can use a `for` loop to perform all eight convolutions with less lines of code.

23. Convolve each mask with the image using a `for` loop.

```
I_k = zeros(size(I,1), size(I,2), 8);
for i = 1:8
    I_k(:,:,i) = imfilter(I,k(:,:,i));
end
```

24. Display the resulting images.

```
figure
for j = 1:8
    subplot(2,4,j), imshow(abs(I_k(:,:,j)),[]), ...
        title(['Kirsch mask ', num2str(j)]);
end
```

***Question 14*** Why are we required to display the absolute value of each mask? *Hint*: Inspect the minimum and maximum values.

***Question 15*** How did we dynamically display the mask number when displaying all eight images?

Next we must find the maximum value of all the images for each pixel. Again, because they are all stored in one matrix, we can do this in one line of code.

25. Find the maximum values.

```
I_kir = max(I_k,[],3);
figure, imshow(I_kir,[]);
```

***Question 16*** When calculating the maximum values, what does the last parameter in the `max` function call mean?

***Question 17*** Why are we required to scale the image when displaying it?

In the previous step we scaled the result for display purposes. If we wish to threshold the image (as we did with all previous edge detectors), we must first scale the image so that its values are within the range [0, 255] as well as convert to class `uint8`. To do so, we can create a linear transformation function that maps all current values to values within the range we want.

26. Create a transformation function to map the image to the grayscale range and perform the transformation.

```
m = 255 / (max(I_kir(:)) - min(I_kir(:)));
I_kir_adj = uint8(m * I_kir);
figure, imshow(I_kir_adj);
```

***Question 18*** Why is it not necessary to scale this image (`I_kir_adj`) when displaying it?

***Question 19*** Make a copy of the `mandrill` image and add Gaussian noise to it. Then perform the Kirsch edge detector on it. Comment on its performance when noise is present.

### Edge Detection with the Robinson Operator

The Robinson edge detector can be implemented in the same manner as the Kirsch detector. The only difference is the masks.

27. Generate the Robinson masks.

```
r = zeros(3,3,8);
r(:,:,1) = [-1 0 1; -2 0 2; -1 0 1];
r(:,:,2) = [0 1 2; -1 0 1; -2 -1 0];
r(:,:,3) = [1 2 1; 0 0 0; -1 -2 -1];
r(:,:,4) = [2 1 0; 1 0 -1; 0 -1 -2];
r(:,:,5) = [1 0 -1; 2 0 -2; 1 0 -1];
r(:,:,6) = [0 -1 -2; 1 0 -1; 2 1 0];
r(:,:,7) = [-1 -2 -1; 0 0 0; 1 2 1];
r(:,:,8) = [-2 -1 0; -1 0 1; 0 1 2];
```

28. Filter the image with the eight Robinson masks and display the output.

```
I_r = zeros(size(I,1), size(I,2), 8);
for i = 1:8
    I_r(:,:,i) = imfilter(I,r(:,:,i));
end
figure
for j = 1:8
    subplot(2,4,j), imshow(abs(I_r(:,:,j)),[]), ...
        title(['Robinson mask ', num2str(j)]);
end
```

29. Calculate the `max` of all eight images and display the result.

```
I_rob = max(I_r,[],3);
figure, imshow(I_kir,[]);
```

***Question 20*** How does the Robinson edge detector compare with the Kirsch detector?

## WHAT HAVE WE LEARNED?

- Edge detection is a fundamental image processing operation that attempts to emulate an ability present in the human visual system. *Edges* in grayscale 2D images are usually defined as a sharp variation of the intensity function. In a more general sense, an *edge* can be defined as a boundary between two image regions having distinct characteristics according to some feature (e.g., gray level, color, or texture). Edge detection is a fundamental step in many image processing techniques: after edges have been detected, the regions enclosed by these edges are segmented and processed accordingly.
- There are numerous edge detection techniques in the image processing literature. They range from simple convolution masks (e.g., Sobel and Prewitt) to biologically inspired techniques (e.g., the Marr–Hildreth method) and the quality of the results they provide vary widely. The Canny edge detector is allegedly the most popular contemporary edge detection method.
- MATLAB has a function `edge` that implements several edge detection methods such as Prewitt, Sobel, Laplacian of Gaussian, and Canny.
- The results of the edge detection algorithm are typically postprocessed by an edge linking algorithm that typically eliminates undesired points, bridges gaps, and results in cleaner edges that are then used in subsequent stages of an edge-based image segmentation solution. The Hough transform is a commonly used technique to find long straight edges (i.e., line segments) within the edge detection results.

## LEARN MORE ABOUT IT

- David Marr's book [Mar82] is one of the most influential books ever written in the field of vision science. Marr's theories remain a source of inspiration to computer vision scientists more than 25 years after they were published.
- Mlsna and Rodriguez discuss first- and second-order derivative edge methods, as well as the Canny edge detector in Chapter 4.11 of [Bov00a].
- Section 15.2 of [Pra07] presents additional first-order derivative edge operators, for example, Frei–Chen, boxcar, truncated pyramid, Argyle, Macleod, and first derivative of Gaussian (FDOG) operators.
- Section 15.5 of [Pra07] discusses several performance criteria that might be used for a comparative analysis of edge detection techniques.

- Other methods for edge linking and boundary detection are discussed in Section 10.2.7 of [GW08], Section 17.4 of [Pra07], and Section 10.3 of [SS01], among other references.
- Chapter 9 of [Dav04] describes in detail the standard Hough transform and its application to line detection.
- Chapters 11, 12, and 14 of [Dav04] discuss the generalized Hough transform (GHT) and its applications to line, ellipse, polygon, and corner detection.
- Computer vision algorithms usually require that other types of relevant primitive properties of the images, such as lines, corners, and points, be detected. For line detection, refer to Section 5.3.9 of [SHB08]. Section 5.4 of [SOS00] covers critical point detection. For corner detection, we recommend Chapter 8 of [BB08] and Section 5.3.10 of [SHB08].

## 14.8 PROBLEMS

**14.1** Write a MATLAB script to generate a test image containing an ideal edge and plot the intensity profile and the first and second derivatives along a horizontal line of the image.

**14.2** Repeat Problem 14.1 for a ramp edge.

**14.3** Show that the LoG edge detector can be implemented using `fspecial` and `imfilter` (instead of `edge`) and provide a reason why this implementation may be preferred.