It is possible to use more neighbors in interpolation, and there are more complex techniques, such as using *splines* or *wavelets*, that in some instances can yield better results than the methods just discussed. While preserving fine detail is an exceptionally important consideration in image generation for 3-D graphics (for example, see Hughes and Andries [2013]), the extra computational burden seldom is justifiable for general-purpose digital image processing, where bilinear or bicubic interpolation typically are the methods of choice.

2.5 SOME BASIC RELATIONSHIPS BETWEEN PIXELS

In this section, we discuss several important relationships between pixels in a digital image. When referring in the following discussion to particular pixels, we use lowercase letters, such as p and q.

NEIGHBORS OF A PIXEL

A pixel p at coordinates (x, y) has two horizontal and two vertical neighbors with coordinates

$$(x+1, y), (x-1, y), (x, y+1), (x, y-1)$$

This set of pixels, called the 4-*neighbors* of p, is denoted $N_4(p)$.

The four *diagonal* neighbors of p have coordinates

$$(x+1, y+1), (x+1, y-1), (x-1, y+1), (x-1, y-1)$$

and are denoted $N_D(p)$. These neighbors, together with the 4-neighbors, are called the 8-*neighbors* of p, denoted by $N_8(p)$. The set of image locations of the neighbors of a point p is called the *neighborhood* of p. The neighborhood is said to be *closed* if it contains p. Otherwise, the neighborhood is said to be *open*.

ADJACENCY, CONNECTIVITY, REGIONS, AND BOUNDARIES

Let V be the set of intensity values used to define adjacency. In a binary image, $V = \{1\}$ if we are referring to adjacency of pixels with value 1. In a grayscale image, the idea is the same, but set V typically contains more elements. For example, if we are dealing with the adjacency of pixels whose values are in the range 0 to 255, set V could be any subset of these 256 values. We consider three types of adjacency:

1. 4-*adjacency*. Two pixels p and q with values from V are 4-adjacent if q is in the set $N_4(p)$.
2. 8-*adjacency*. Two pixels p and q with values from V are 8-adjacent if q is in the set $N_8(p)$.
3. *m-adjacency* (also called *mixed adjacency*). Two pixels p and q with values from V are m-adjacent if

(a) q is in $N_4(p)$, *or*

(b) q is in $N_D(p)$ *and* the set $N_4(p) \cap N_4(q)$ has no pixels whose values are from V.

We use the symbols $\cap$ and $\cup$ to denote set intersection and union, respectively. Given sets A and B, recall that their intersection is the set of elements that are members of both A and B. The union of these two sets is the set of elements that are members of A, of B, or of both. We will discuss sets in more detail in Section 2.6.

Mixed adjacency is a modification of 8-adjacency, and is introduced to eliminate the ambiguities that may result from using 8-adjacency. For example, consider the pixel arrangement in Fig. 2.28(a) and let $V = \{1\}$. The three pixels at the top of Fig. 2.28(b) show multiple (ambiguous) 8-adjacency, as indicated by the dashed lines. This ambiguity is removed by using m-adjacency, as in Fig. 2.28(c). In other words, the center and upper-right diagonal pixels are not m-adjacent because they do not satisfy condition (b).

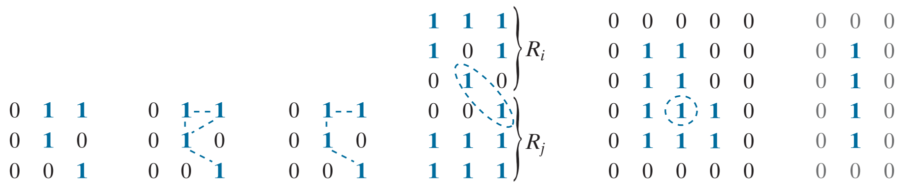
A *digital path* (or *curve*) from pixel p with coordinates (x_0, y_0) to pixel q with coordinates (x_n, y_n) is a sequence of distinct pixels with coordinates

$$(x_0, y_0), (x_1, y_1), \ldots, (x_n, y_n)$$

where points (x_i, y_i) and (x_{i-1}, y_{i-1}) are adjacent for $1 \le i \le n$. In this case, n is the *length* of the path. If $(x_0, y_0) = (x_n, y_n)$ the path is a *closed* path. We can define 4-, 8-, or m-paths, depending on the type of adjacency specified. For example, the paths in Fig. 2.28(b) between the top right and bottom right points are 8-paths, and the path in Fig. 2.28(c) is an m-path.

Let S represent a subset of pixels in an image. Two pixels p and q are said to be *connected in* S if there exists a path between them consisting entirely of pixels in S. For any pixel p in S, the set of pixels that are connected to it in S is called a *connected component* of S. If it only has one component, and that component is connected, then S is called a *connected set*.

Let R represent a subset of pixels in an image. We call R a *region* of the image if R is a connected set. Two regions, R_i and R_j are said to be *adjacent* if their union forms a connected set. Regions that are not adjacent are said to be *disjoint*. We consider 4- and 8-adjacency when referring to regions. For our definition to make sense, the type of adjacency used must be specified. For example, the two regions of 1's in Fig. 2.28(d) are adjacent only if 8-adjacency is used (according to the definition in the previous



a b c d e f

FIGURE 2.28 (a) An arrangement of pixels. (b) Pixels that are 8-adjacent (adjacency is shown by dashed lines). (c) m-adjacency. (d) Two regions (of 1's) that are 8-adjacent. (e) The circled point is on the boundary of the 1-valued pixels only if 8-adjacency between the region and background is used. (f) The inner boundary of the 1-valued region does not form a closed path, but its outer boundary does.

paragraph, a 4-path between the two regions does not exist, so their union is not a connected set).

Suppose an image contains K disjoint regions, $R_k, k = 1, 2, \ldots, K$, none of which touches the image border.[†] Let R_u denote the union of all the K regions, and let $(R_u)^c$ denote its complement (recall that the *complement* of a set A is the set of points that are not in A). We call all the points in R_u the *foreground*, and all the points in $(R_u)^c$ the *background* of the image.

The *boundary* (also called the *border* or *contour*) of a region R is the set of pixels in R that are adjacent to pixels in the complement of R. Stated another way, the border of a region is the set of pixels in the region that have at least one background neighbor. Here again, we must specify the connectivity being used to define adjacency. For example, the point circled in Fig. 2.28(e) is not a member of the border of the 1-valued region if 4-connectivity is used between the region and its background, because the only possible connection between that point and the background is diagonal. As a rule, adjacency between points in a region and its background is defined using 8-connectivity to handle situations such as this.

The preceding definition sometimes is referred to as the *inner border* of the region to distinguish it from its *outer border*, which is the corresponding border in the background. This distinction is important in the development of border-following algorithms. Such algorithms usually are formulated to follow the outer boundary in order to guarantee that the result will form a closed path. For instance, the inner border of the 1-valued region in Fig. 2.28(f) is the region itself. This border does not satisfy the definition of a closed path. On the other hand, the outer border of the region does form a closed path around the region.

If R happens to be an entire image, then its *boundary* (or *border*) is defined as the set of pixels in the first and last rows and columns of the image. This extra definition is required because an image has no neighbors beyond its border. Normally, when we refer to a region, we are referring to a subset of an image, and any pixels in the boundary of the region that happen to coincide with the border of the image are included implicitly as part of the region boundary.

The concept of an *edge* is found frequently in discussions dealing with regions and boundaries. However, there is a key difference between these two concepts. The boundary of a finite region forms a closed path and is thus a "global" concept. As we will discuss in detail in Chapter 10, edges are formed from pixels with derivative values that exceed a preset threshold. Thus, an edge is a "local" concept that is based on a measure of intensity-level discontinuity at a point. It is possible to link edge points into edge segments, and sometimes these segments are linked in such a way that they correspond to boundaries, but this is not always the case. The one exception in which edges and boundaries correspond is in binary images. Depending on the type of connectivity and edge operators used (we will discuss these in Chapter 10), the edge extracted from a binary region will be the same as the region boundary. This is

[†] We make this assumption to avoid having to deal with special cases. This can be done without loss of generality because if one or more regions touch the border of an image, we can simply pad the image with a 1-pixel-wide border of background values.

intuitive. Conceptually, until we arrive at Chapter 10, it is helpful to think of edges as intensity discontinuities, and of boundaries as closed paths.

DISTANCE MEASURES

For pixels p, q, and s, with coordinates $(x, y), (u, v)$, and (w, z), respectively, D is a *distance function* or *metric* if

(a) $D(p,q) \geq 0 \quad (D(p,q) = 0 \text{ iff } p = q)$,
(b) $D(p,q) = D(q,p)$, and
(c) $D(p,s) \leq D(p,q) + D(q,s)$.

The *Euclidean distance* between p and q is defined as

$$D_e(p,q) = \left[(x-u)^2 + (y-v)^2\right]^{\frac{1}{2}} \tag{2-19}$$

For this distance measure, the pixels having a distance less than or equal to some value r from (x, y) are the points contained in a disk of radius r centered at (x, y).

The D_4 *distance*, (called the *city-block distance*) between p and q is defined as

$$D_4(p,q) = |x-u| + |y-v| \tag{2-20}$$

In this case, pixels having a D_4 distance from (x, y) that is less than or equal to some value d form a diamond centered at (x, y). For example, the pixels with D_4 distance ≤ 2 from (x, y) (the center point) form the following contours of constant distance:

$$\begin{array}{ccccc} & & 2 & & \\ & 2 & 1 & 2 & \\ 2 & 1 & 0 & 1 & 2 \\ & 2 & 1 & 2 & \\ & & 2 & & \end{array}$$

The pixels with $D_4 = 1$ are the 4-neighbors of (x, y).

The D_8 *distance* (called the *chessboard distance*) between p and q is defined as

$$D_8(p,q) = \max(|x-u|, |y-v|) \tag{2-21}$$

In this case, the pixels with D_8 distance from (x, y) less than or equal to some value d form a square centered at (x, y). For example, the pixels with D_8 distance ≤ 2 form the following contours of constant distance:

$$\begin{array}{ccccc} 2 & 2 & 2 & 2 & 2 \\ 2 & 1 & 1 & 1 & 2 \\ 2 & 1 & 0 & 1 & 2 \\ 2 & 1 & 1 & 1 & 2 \\ 2 & 2 & 2 & 2 & 2 \end{array}$$

The pixels with $D_8 = 1$ are the 8-neighbors of the pixel at (x, y).

Note that the D_4 and D_8 distances between p and q are independent of any paths that might exist between these points because these distances involve only the coordinates of the points. In the case of m-adjacency, however, the D_m distance between two points is defined as the shortest m-path between the points. In this case, the distance between two pixels will depend on the values of the pixels along the path, as well as the values of their neighbors. For instance, consider the following arrangement of pixels and assume that p, p_2, and p_4 have a value of 1, and that p_1 and p_3 can be 0 or 1:

$$\begin{matrix} & p_3 & p_4 \\ p_1 & p_2 & \\ p & & \end{matrix}$$

Suppose that we consider adjacency of pixels valued 1 (i.e., $V = \{1\}$). If p_1 and p_3 are 0, the length of the shortest m-path (the D_m distance) between p and p_4 is 2. If p_1 is 1, then p_2 and p will no longer be m-adjacent (see the definition of m-adjacency given earlier) and the length of the shortest m-path becomes 3 (the path goes through the points $p p_1 p_2 p_4$). Similar comments apply if p_3 is 1 (and p_1 is 0); in this case, the length of the shortest m-path also is 3. Finally, if both p_1 and p_3 are 1, the length of the shortest m-path between p and p_4 is 4. In this case, the path goes through the sequence of points $p p_1 p_2 p_3 p_4$.

2.6 INTRODUCTION TO THE BASIC MATHEMATICAL TOOLS USED IN DIGITAL IMAGE PROCESSING

You may find it helpful to download and study the review material dealing with probability, vectors, linear algebra, and linear systems. The review is available in the Tutorials section of the book website.

This section has two principal objectives: (1) to introduce various mathematical tools we use throughout the book; and (2) to help you begin developing a “feel” for how these tools are used by applying them to a variety of basic image-processing tasks, some of which will be used numerous times in subsequent discussions.

ELEMENTWISE VERSUS MATRIX OPERATIONS

An *elementwise operation* involving one or more images is carried out on a *pixel-by-pixel* basis. We mentioned earlier in this chapter that images can be viewed equivalently as matrices. In fact, as you will see later in this section, there are many situations in which operations between images are carried out using matrix theory. It is for this reason that a clear distinction must be made between elementwise and matrix operations. For example, consider the following 2×2 images (matrices):

$$\begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \quad \text{and} \quad \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix}$$

The elementwise product of two matrices is also called the *Hadamard product* of the matrices.

The *elementwise product* (often denoted using the symbol $\odot$ or $\otimes$) of these two images is

The symbol $\ominus$ is often used to denote *elementwise division*.

$$\begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \odot \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix} = \begin{bmatrix} a_{11}b_{11} & a_{12}b_{12} \\ a_{21}b_{21} & a_{22}b_{22} \end{bmatrix}$$

That is, the elementwise product is obtained by multiplying pairs of *corresponding* pixels. On the other hand, the *matrix product* of the images is formed using the rules of matrix multiplication:

$$\begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix}\begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix} = \begin{bmatrix} a_{11}b_{11} + a_{12}b_{21} & a_{11}b_{12} + a_{12}b_{22} \\ a_{21}b_{11} + a_{22}b_{21} & a_{21}b_{12} + a_{22}b_{22} \end{bmatrix}$$

We assume elementwise operations throughout the book, unless stated otherwise. For example, when we refer to raising an image to a power, we mean that each individual pixel is raised to that power; when we refer to dividing an image by another, we mean that the division is between corresponding pixel pairs, and so on. The terms *elementwise addition* and *subtraction* of two images are redundant because these are elementwise operations by definition. However, you may see them used sometimes to clarify notational ambiguities.

LINEAR VERSUS NONLINEAR OPERATIONS

One of the most important classifications of an image processing method is whether it is linear or nonlinear. Consider a general operator, $\mathcal{H}$, that produces an output image, $g(x,y)$, from a given input image, $f(x,y)$:

$$\mathcal{H}\left[f(x,y)\right] = g(x,y) \tag{2-22}$$

Given two arbitrary constants, a and b, and two arbitrary images $f_1(x,y)$ and $f_2(x,y)$, $\mathcal{H}$ is said to be a *linear operator* if

$$\begin{aligned} \mathcal{H}\left[af_1(x,y) + bf_2(x,y)\right] &= a\mathcal{H}\left[f_1(x,y)\right] + b\mathcal{H}\left[f_2(x,y)\right] \\ &= ag_1(x,y) + bg_2(x,y) \end{aligned} \tag{2-23}$$

This equation indicates that the output of a linear operation applied to the sum of two inputs is the same as performing the operation individually on the inputs and then summing the results. In addition, the output of a linear operation on a constant multiplied by an input is the same as the output of the operation due to the original input multiplied by that constant. The first property is called the property of *additivity*, and the second is called the property of *homogeneity*. By definition, an operator that fails to satisfy Eq. (2-23) is said to be *nonlinear*.

As an example, suppose that $\mathcal{H}$ is the sum operator, Σ. The function performed by this operator is simply to sum its inputs. To test for linearity, we start with the left side of Eq. (2-23) and attempt to prove that it is equal to the right side:

These are image summations, not the sums of all the elements of an image.

$$\begin{aligned} \sum\left[af_1(x,y) + bf_2(x,y)\right] &= \sum af_1(x,y) + \sum bf_2(x,y) \\ &= a\sum f_1(x,y) + b\sum f_2(x,y) \\ &= ag_1(x,y) + bg_2(x,y) \end{aligned}$$

where the first step follows from the fact that summation is distributive. So, an expansion of the left side is equal to the right side of Eq. (2-23), and we conclude that the sum operator is linear.

On the other hand, suppose that we are working with the max operation, whose function is to find the maximum value of the pixels in an image. For our purposes here, the simplest way to prove that this operator is nonlinear is to find an example that fails the test in Eq. (2-23). Consider the following two images

$$f_1 = \begin{bmatrix} 0 & 2 \\ 2 & 3 \end{bmatrix} \quad \text{and} \quad f_2 = \begin{bmatrix} 6 & 5 \\ 4 & 7 \end{bmatrix}$$

and suppose that we let $a = 1$ and $b = -1$. To test for linearity, we again start with the left side of Eq. (2-23):

$$\begin{aligned} \max\left\{(1)\begin{bmatrix} 0 & 2 \\ 2 & 3 \end{bmatrix} + (-1)\begin{bmatrix} 6 & 5 \\ 4 & 7 \end{bmatrix}\right\} &= \max\left\{\begin{bmatrix} -6 & -3 \\ -2 & -4 \end{bmatrix}\right\} \\ &= -2 \end{aligned}$$

Working next with the right side, we obtain

$$(1)\max\left\{\begin{bmatrix} 0 & 2 \\ 2 & 3 \end{bmatrix}\right\} + (-1)\max\left\{\begin{bmatrix} 6 & 5 \\ 4 & 7 \end{bmatrix}\right\} = 3 + (-1)7 = -4$$

The left and right sides of Eq. (2-23) are not equal in this case, so we have proved that the max operator is nonlinear.

As you will see in the next three chapters, linear operations are exceptionally important because they encompass a large body of theoretical and practical results that are applicable to image processing. The scope of nonlinear operations is considerably more limited. However, you will encounter in the following chapters several nonlinear image processing operations whose performance far exceeds what is achievable by their linear counterparts.

ARITHMETIC OPERATIONS

Arithmetic operations between two images $f(x,y)$ and $g(x,y)$ are denoted as

$$\begin{aligned} s(x,y) &= f(x,y) + g(x,y) \\ d(x,y) &= f(x,y) - g(x,y) \\ p(x,y) &= f(x,y) \times g(x,y) \\ v(x,y) &= f(x,y) \div g(x,y) \end{aligned} \qquad (2\text{-}24)$$

These are elementwise operations which, as noted earlier in this section, means that they are performed between corresponding pixel pairs in f and g for $x = 0, 1, 2, \ldots, M-1$ and $y = 0, 1, 2, \ldots, N-1$. As usual, M and N are the row and column sizes of the images. Clearly, s, d, p, and v are images of size $M \times N$ also. Note that image arithmetic in the manner just defined involves images of the same size. The following examples illustrate the important role of arithmetic operations in digital image processing.

EXAMPLE 2.5: Using image addition (averaging) for noise reduction.

Suppose that $g(x, y)$ is a corrupted image formed by the addition of noise, $\eta(x, y)$, to a *noiseless* image $f(x, y)$; that is,

$$g(x, y) = f(x, y) + \eta(x, y) \tag{2-25}$$

where the assumption is that at every pair of coordinates (x, y) the noise is uncorrelated[†] and has zero average value. We assume also that the noise and image values are uncorrelated (this is a typical assumption for additive noise). The objective of the following procedure is to reduce the noise content of the output image by adding a set of noisy input images, $\{g_i(x, y)\}$. This is a technique used frequently for image enhancement.

If the noise satisfies the constraints just stated, it can be shown (Problem 2.26) that if an image $\bar{g}(x, y)$ is formed by averaging K different noisy images,

$$\bar{g}(x, y) = \frac{1}{K}\sum_{i=1}^{K} g_i(x, y) \tag{2-26}$$

then it follows that

$$E\{\bar{g}(x, y)\} = f(x, y) \tag{2-27}$$

and

$$\sigma^2_{\bar{g}(x,y)} = \frac{1}{K}\sigma^2_{\eta(x,y)} \tag{2-28}$$

where $E\{\bar{g}(x, y)\}$ is the expected value of $\bar{g}(x, y)$, and $\sigma^2_{\bar{g}(x,y)}$ and $\sigma^2_{\eta(x,y)}$ are the variances of $\bar{g}(x, y)$ and $\eta(x, y)$, respectively, all at coordinates (x, y). These variances are arrays of the same size as the input image, and there is a scalar variance value for each pixel location.

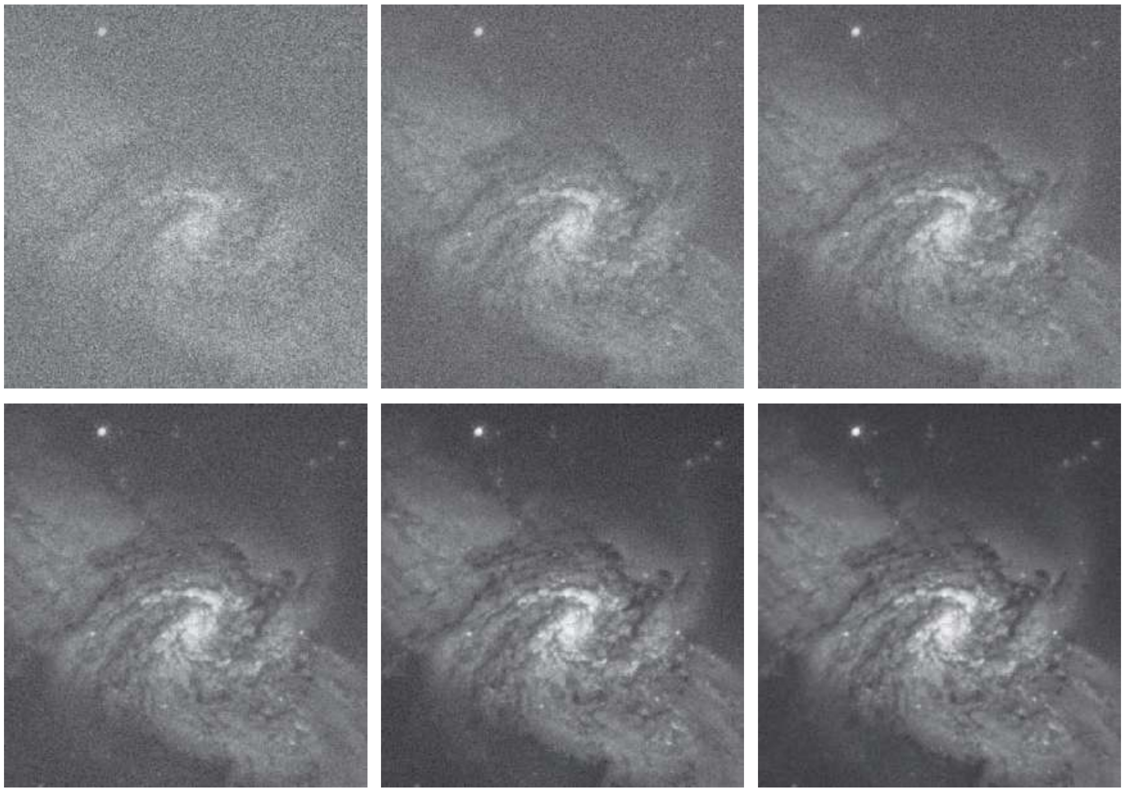
The standard deviation (square root of the variance) at any point (x, y) in the average image is

$$\sigma_{\bar{g}(x,y)} = \frac{1}{\sqrt{K}}\sigma_{\eta(x,y)} \tag{2-29}$$

As K increases, Eqs. (2-28) and (2-29) indicate that the variability (as measured by the variance or the standard deviation) of the pixel values at each location (x, y) decreases. Because $E\{\bar{g}(x, y)\} = f(x, y)$, this means that $\bar{g}(x, y)$ approaches the noiseless image $f(x, y)$ as the number of noisy images used in the averaging process increases. In order to avoid blurring and other artifacts in the output (average) image, it is necessary that the images $g_i(x, y)$ be *registered* (i.e., spatially aligned).

An important application of image averaging is in the field of astronomy, where imaging under very low light levels often cause sensor noise to render individual images virtually useless for analysis (lowering the temperature of the sensor helps reduce noise). Figure 2.29(a) shows an 8-bit image of the Galaxy Pair NGC 3314, in which noise corruption was simulated by adding to it Gaussian noise with zero mean and a standard deviation of 64 intensity levels. This image, which is representative of noisy astronomical images taken under low light conditions, is useless for all practical purposes. Figures 2.29(b) through (f) show the results of averaging 5, 10, 20, 50, and 100 images, respectively. We see from Fig. 2.29(b) that an average of only 10 images resulted in some visible improvement. According to Eq.

[†]The variance of a random variable z with mean $\bar{z}$ is defined as $E\{(z - \bar{z})^2\}$, where $E\{\cdot\}$ is the expected value of the argument. The covariance of two random variables z_i and z_j is defined as $E\{(z_i - \bar{z}_i)(z_j - \bar{z}_j)\}$. If the variables are uncorrelated, their covariance is 0, and vice versa. (Do not confuse correlation and statistical independence. If two random variables are statistically independent, their correlation is zero. However, the converse is not true in general.)



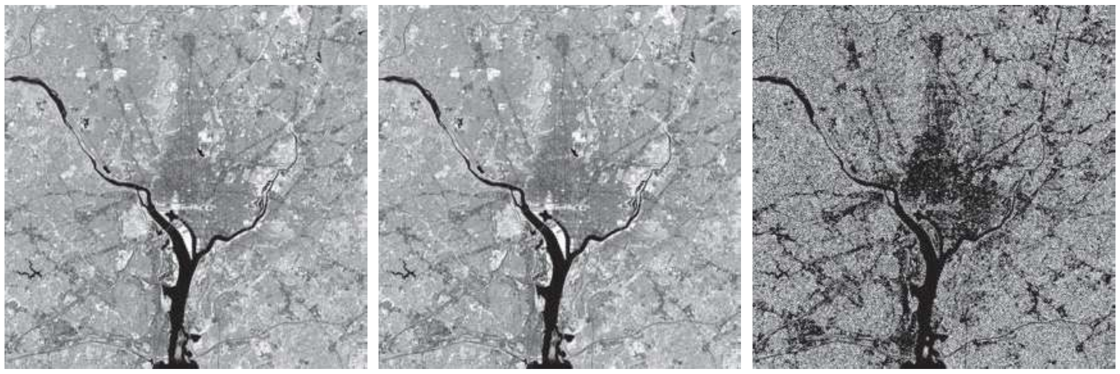
a b c
d e f

FIGURE 2.29 (a) Image of Galaxy Pair NGC 3314 corrupted by additive Gaussian noise. (b)-(f) Result of averaging 5, 10, 20, 50, and 1,00 noisy images, respectively. All images are of size 566×598 pixels, and all were scaled so that their intensities would span the full [0, 255] intensity scale. (Original image courtesy of NASA.)

(2-29), the standard deviation of the noise in Fig. 2.29(b) is less than half ($1/\sqrt{5} = 0.45$) the standard deviation of the noise in Fig. 2.29(a), or $(0.45)(64) \approx 29$ intensity levels. Similarly, the standard deviations of the noise in Figs. 2.29(c) through (f) are 0.32, 0.22, 0.14, and 0.10 of the original, which translates approximately into 20, 14, 9, and 6 intensity levels, respectively. We see in these images a progression of more visible detail as the standard deviation of the noise decreases. The last two images are visually identical for all practical purposes. This is not unexpected, as the difference between the standard deviations of their noise level is only about 3 intensity levels According to the discussion in connection with Fig. 2.5, this difference is below what a human generally is able to detect.

EXAMPLE 2.6: Comparing images using subtraction.

Image subtraction is used routinely for enhancing differences between images. For example, the image in Fig. 2.30(b) was obtained by setting to zero the least-significant bit of every pixel in Fig. 2.30(a). Visually, these images are indistinguishable. However, as Fig. 2.30(c) shows, subtracting one image from

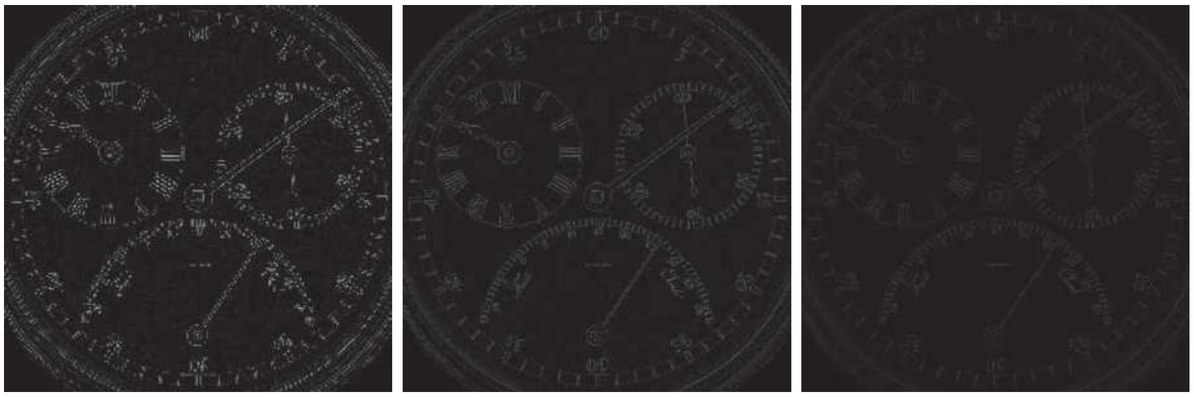


a b c

FIGURE 2.30 (a) Infrared image of the Washington, D.C. area. (b) Image resulting from setting to zero the least significant bit of every pixel in (a). (c) Difference of the two images, scaled to the range [0, 255] for clarity. (Original image courtesy of NASA.)

the other clearly shows their differences. Black (0) values in the difference image indicate locations where there is no difference between the images in Figs. 2.30(a) and (b).

We saw in Fig. 2.23 that detail was lost as the resolution was reduced in the chronometer image shown in Fig. 2.23(a). A vivid indication of image change as a function of resolution can be obtained by displaying the differences between the original image and its various lower-resolution counterparts. Figure 2.31(a) shows the difference between the 930 dpi and 72 dpi images. As you can see, the differences are quite noticeable. The intensity at any point in the difference image is proportional to the magnitude of the numerical difference between the two images at that point. Therefore, we can analyze which areas of the original image are affected the most when resolution is reduced. The next two images in Fig. 2.31 show proportionally less overall intensities, indicating smaller differences between the 930 dpi image and 150 dpi and 300 dpi images, as expected.



a b c

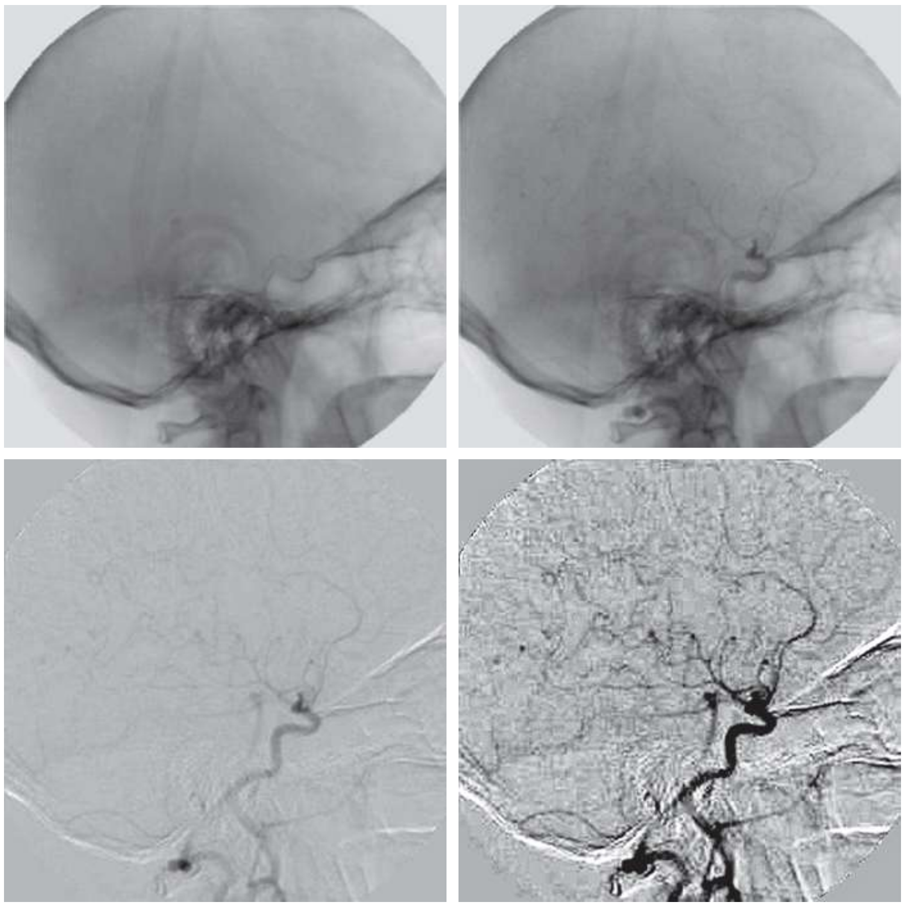
FIGURE 2.31 (a) Difference between the 930 dpi and 72 dpi images in Fig. 2.23. (b) Difference between the 930 dpi and 150 dpi images. (c) Difference between the 930 dpi and 300 dpi images.

As a final illustration, we discuss briefly an area of medical imaging called *mask mode radiography*, a commercially successful and highly beneficial use of image subtraction. Consider image differences of the form

$$g(x,y) = f(x,y) - h(x,y) \tag{2-30}$$

In this case $h(x,y)$, the *mask*, is an X-ray image of a region of a patient's body captured by an intensified TV camera (instead of traditional X-ray film) located opposite an X-ray source. The procedure consists of injecting an X-ray contrast medium into the patient's bloodstream, taking a series of images called *live images* [samples of which are denoted as $f(x,y)$] of the same anatomical region as $h(x,y)$, and subtracting the mask from the series of incoming live images after injection of the contrast medium. The net effect of subtracting the mask from each sample live image is that the areas that are different between $f(x,y)$ and $h(x,y)$ appear in the output image, $g(x,y)$, as enhanced detail. Because images can be captured at TV rates, this procedure outputs a video showing how the contrast medium propagates through the various arteries in the area being observed.

Figure 2.32(a) shows a mask X-ray image of the top of a patient's head prior to injection of an iodine medium into the bloodstream, and Fig. 2.32(b) is a sample of a live image taken after the medium was



a b
c d

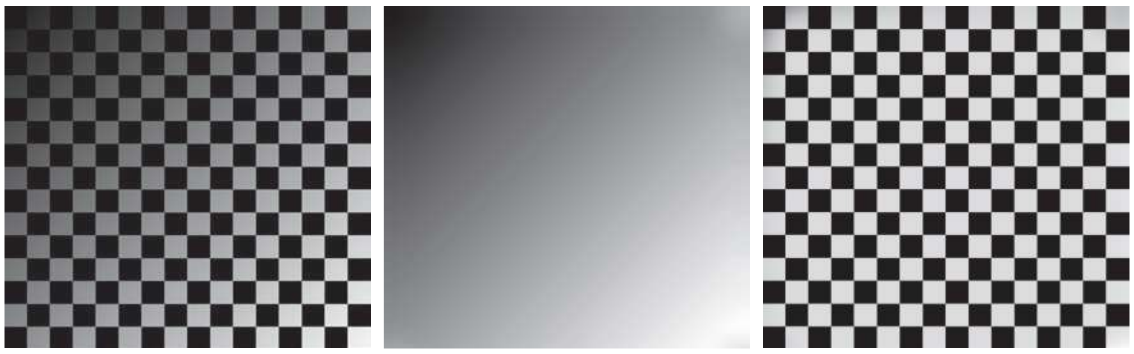
FIGURE 2.32 Digital subtraction angiography. (a) Mask image. (b) A live image. (c) Difference between (a) and (b). (d) Enhanced difference image. (Figures (a) and (b) courtesy of the Image Sciences Institute, University Medical Center, Utrecht, The Netherlands.)

injected. Figure 2.32(c) is the difference between (a) and (b). Some fine blood vessel structures are visible in this image. The difference is clear in Fig. 2.32(d), which was obtained by sharpening the image and enhancing its contrast (we will discuss these techniques in the next chapter). Figure 2.32(d) is a "snapshot" of how the medium is propagating through the blood vessels in the subject's brain.

EXAMPLE 2.7: Using image multiplication and division for shading correction and for masking.

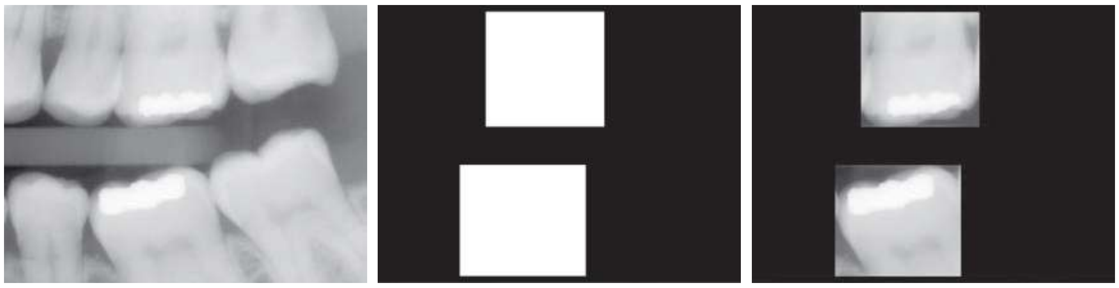
An important application of image multiplication (and division) is *shading correction*. Suppose that an imaging sensor produces images that can be modeled as the product of a "perfect image," denoted by $f(x,y)$, times a shading function, $h(x,y)$; that is, $g(x,y) = f(x,y)h(x,y)$. If $h(x,y)$ is known or can be estimated, we can obtain $f(x,y)$ (or an estimate of it) by multiplying the sensed image by the inverse of $h(x,y)$ (i.e., dividing g by h using elementwise division). If access to the imaging system is possible, we can obtain a good approximation to the shading function by imaging a target of constant intensity. When the sensor is not available, we often can estimate the shading pattern directly from a shaded image using the approaches discussed in Sections 3.5 and 9.8. Figure 2.33 shows an example of shading correction using an estimate of the shading pattern. The corrected image is not perfect because of errors in the shading pattern (this is typical), but the result definitely is an improvement over the shaded image in Fig. 2.33 (a). See Section 3.5 for a discussion of how we estimated Fig. 2.33 (b). Another use of image multiplication is in *masking*, also called *region of interest* (ROI), operations. As Fig. 2.34 shows, the process consists of multiplying a given image by a mask image that has 1's in the ROI and 0's elsewhere. There can be more than one ROI in the mask image, and the shape of the ROI can be arbitrary.

A few comments about implementing image arithmetic operations are in order before we leave this section. In practice, most images are displayed using 8 bits (even 24-bit color images consist of three separate 8-bit channels). Thus, we expect image values to be in the range from 0 to 255. When images are saved in a standard image format, such as TIFF or JPEG, conversion to this range is automatic. When image values exceed the allowed range, clipping or scaling becomes necessary. For example, the values in the difference of two 8-bit images can range from a minimum of −255



a b c

FIGURE 2.33 Shading correction. (a) Shaded test pattern. (b) Estimated shading pattern. (c) Product of (a) by the reciprocal of (b). (See Section 3.5 for a discussion of how (b) was estimated.)



a b c

FIGURE 2.34 (a) Digital dental X-ray image. (b) ROI mask for isolating teeth with fillings (white corresponds to 1 and black corresponds to 0). (c) Product of (a) and (b).

to a maximum of 255, and the values of the sum of two such images can range from 0 to 510. When converting images to eight bits, many software applications simply set all negative values to 0 and set to 255 all values that exceed this limit. Given a digital image g resulting from one or more arithmetic (or other) operations, an approach guaranteeing that the full range of a values is "captured" into a fixed number of bits is as follows. First, we perform the operation

$$g_m = g - \min(g) \tag{2-31}$$

These are elementwise subtraction and division.

which creates an image whose minimum value is 0. Then, we perform the operation

$$g_s = K\left[g_m / \max(g_m)\right] \tag{2-32}$$

which creates a scaled image, g_s, whose values are in the range $[0, K]$. When working with 8-bit images, setting $K = 255$ gives us a scaled image whose intensities span the full 8-bit scale from 0 to 255. Similar comments apply to 16-bit images or higher. This approach can be used for all arithmetic operations. When performing division, we have the extra requirement that a small number should be added to the pixels of the divisor image to avoid division by 0.

SET AND LOGICAL OPERATIONS

In this section, we discuss the basics of set theory. We also introduce and illustrate some important set and logical operations.

Basic Set Operations

A *set* is a collection of distinct objects. If a is an *element* of set A, then we write

$$a \in A \tag{2-33}$$

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

$$a \notin A \tag{2-34}$$

The set with no elements is called the *null* or *empty set*, and is denoted by $\varnothing$.

A set is denoted by the contents of two braces: $\{\bullet\}$. For example, the expression

$$C = \{c \mid c = -d,\ d \in D\}$$

means that C is the set of elements, c, such that c is formed by multiplying each of the elements of set D by -1.

If every element of a set A is also an element of a set B, then A is said to be a *subset* of B, denoted as

$$A \subseteq B \tag{2-35}$$

The *union* of two sets A and B, denoted as

$$C = A \cup B \tag{2-36}$$

is a set C consisting of elements belonging *either* to A, to B, *or* to *both*. Similarly, the *intersection* of two sets A and B, denoted by

$$D = A \cap B \tag{2-37}$$

is a set D consisting of elements belonging to *both* A and B. Sets A and B are said to be *disjoint* or *mutually exclusive* if they have no elements in common, in which case,

$$A \cap B = \varnothing \tag{2-38}$$

The *sample space*, Ω, (also called the *set universe*) is the set of all possible set elements in a given application. By definition, these set elements are members of the sample space for that application. For example, if you are working with the set of real numbers, then the sample space is the real line, which contains all the real numbers. In image processing, we typically define Ω to be the rectangle containing all the pixels in an image.

The *complement* of a set A is the set of elements that are not in A:

$$A^c = \{w \mid w \notin A\} \tag{2-39}$$

The *difference* of two sets A and B, denoted $A - B$, is defined as

$$A - B = \{w \mid w \in A, w \notin B\} = A \cap B^c \tag{2-40}$$

This is the set of elements that belong to A, but not to B. We can define A^c in terms of Ω and the set difference operation; that is, $A^c = \Omega - A$. Table 2.1 shows several important set properties and relationships.

Figure 2.35 shows diagrammatically (in so-called *Venn diagrams*) some of the set relationships in Table 2.1. The shaded areas in the various figures correspond to the set operation indicated above or below the figure. Figure 2.35(a) shows the sample set, Ω. As no earlier, this is the set of all possible elements in a given application. Figure 2.35(b) shows that the complement of a set A is the set of all elements in Ω that are not in A, which agrees with our earlier definition. Observe that Figs. 2.35(e) and (g) are identical, which proves the validity of Eq. (2-40) using Venn diagrams. This

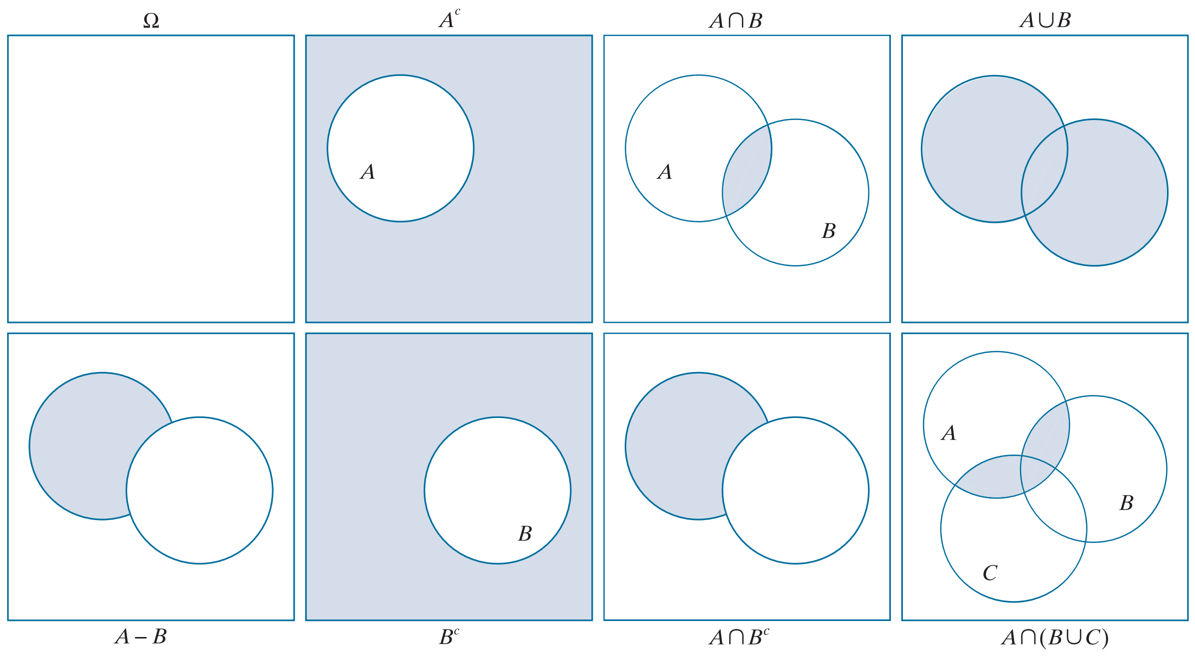
TABLE 2.1 Some important set operations and relationships.

Description	Expressions
Operations between the sample space and null sets	$\Omega^c = \varnothing;\ \varnothing^c = \Omega;\ \Omega \cup \varnothing = \Omega;\ \Omega \cap \varnothing = \varnothing$
Union and intersection with the null and sample space sets	$A \cup \varnothing = A;\ A \cap \varnothing = \varnothing;\ A \cup \Omega = \Omega;\ A \cap \Omega = A$
Union and intersection of a set with itself	$A \cup A = A;\ A \cap A = A$
Union and intersection of a set with its complement	$A \cup A^c = \Omega;\ A \cap A^c = \varnothing$
Commutative laws	$A \cup B = B \cup A$ $A \cap B = B \cap A$
Associative laws	$(A \cup B) \cup C = A \cup (B \cup C)$ $(A \cap B) \cap C = A \cap (B \cap C)$
Distributive laws	$(A \cup B) \cap C = (A \cap C) \cup (B \cap C)$ $(A \cap B) \cup C = (A \cup C) \cap (B \cup C)$
DeMorgan's laws	$(A \cup B)^c = A^c \cap B^c$ $(A \cap B)^c = A^c \cup B^c$

is an example of the usefulness of Venn diagrams for proving equivalences between set relationships.

When applying the concepts just discussed to image processing, we let sets represent objects (regions) in a binary image, and the elements of the sets are the (x, y) coordinates of those objects. For example, if we want to know whether two objects, A and B, of a binary image overlap, all we have to do is compute $A \cap B$. If the result is not the empty set, we know that some of the elements of the two objects overlap. Keep in mind that the only way that the operations illustrated in Fig. 2.35 can make sense in the context of image processing is if the images containing the sets are binary, in which case we can talk about set membership based on coordinates, the assumption being that all members of the sets have the same intensity value (typically denoted by 1). We will discuss set operations involving binary images in more detail in the following section and in Chapter 9.

The preceding concepts are not applicable when dealing with grayscale images, because we have not defined yet a mechanism for assigning intensity values to the pixels resulting from a set operation. In Sections 3.8 and 9.6 we will define the union and intersection operations for grayscale values as the maximum and minimum of corresponding pixel pairs, respectively. We define the *complement* of a grayscale image as the pairwise differences between a constant and the intensity of every pixel in the image. The fact that we deal with corresponding pixel pairs tells us that grayscale set operations are elementwise operations, as defined earlier. The following example is a brief illustration of set operations involving grayscale images. We will discuss these concepts further in the two sections just mentioned.



a b c d
e f g h

FIGURE 2.35 Venn diagrams corresponding to some of the set operations in Table 2.1. The results of the operations, such as A^c, are shown shaded. Figures (e) and (g) are the same, proving via Venn diagrams that $A - B = A \cap B^c$ [see Eq. (2-40)].

EXAMPLE 2.8: Illustration of set operations involving grayscale images.

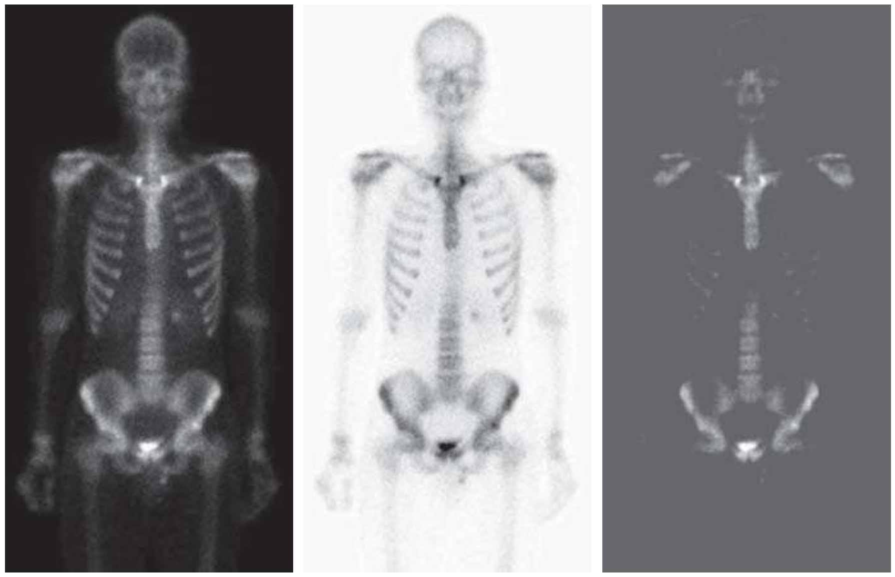
Let the elements of a grayscale image be represented by a set A whose elements are triplets of the form (x, y, z), where x and y are spatial coordinates, and z denotes intensity values. We define the *complement* of A as the set

$$A^c = \left\{(x, y, K - z) \,\middle|\, (x, y, z) \in A\right\}$$

which is the set of pixels of A whose intensities have been subtracted from a constant K. This constant is equal to the maximum intensity value in the image, $2^k - 1$, where k is the number of bits used to represent z. Let A denote the 8-bit grayscale image in Fig. 2.36(a), and suppose that we want to form the negative of A using grayscale set operations. The negative is the set complement, and this is an 8-bit image, so all we have to do is let $K = 255$ in the set defined above:

$$A^c = \left\{(x, y, 255 - z) \,\middle|\, (x, y, z) \in A\right\}$$

Figure 2.36(b) shows the result. We show this only for illustrative purposes. Image negatives generally are computed using an intensity transformation function, as discussed later in this section.



a b c

FIGURE 2.36 Set operations involving grayscale images. (a) Original image. (b) Image negative obtained using grayscale set complementation. (c) The union of image (a) and a constant image. (Original image courtesy of G.E. Medical Systems.)

The *union* of two grayscale sets A and B with the same number of elements is defined as the set

$$A \cup B = \left\{ \max_z(a, b) \mid a \in A, b \in B \right\}$$

where it is understood that the max operation is applied to pairs of corresponding elements. If A and B are grayscale images of the same size, we see that their the union is an array formed from the maximum intensity between pairs of spatially corresponding elements. As an illustration, suppose that A again represents the image in Fig. 2.36(a), and let B denote a rectangular array of the same size as A, but in which all values of z are equal to 3 times the mean intensity, $\bar{z}$, of the elements of A. Figure 2.36(c) shows the result of performing the set union, in which all values exceeding $3\bar{z}$ appear as values from A and all other pixels have value $3\bar{z}$, which is a mid-gray value.

We follow convention in using the symbol $\times$ to denote the Cartesian product. This is not to be confused with our use of the same symbol throughout the book to denote the size of an M-by-N image (i.e., $M \times N$).

Before leaving the discussion of sets, we introduce some additional concepts that are used later in the book. The *Cartesian product* of two sets X and Y, denoted $X \times Y$, is the set of *all* possible ordered pairs whose first component is a member of X and whose second component is a member of Y. In other words,

$$X \times Y = \left\{ (x, y) \mid x \in X \text{ and } y \in Y \right\} \tag{2-41}$$

For example, if X is a set of M equally spaced values on the x-axis and Y is a set of N equally spaced values on the y-axis, we see that the Cartesian product of these two sets define the coordinates of an M-by-N rectangular array (i.e., the coordinates of an image). As another example, if X and Y denote the specific x- and y-coordinates of a group of 8-connected, 1-valued pixels in a binary image, then set $X \times Y$ represents the region (object) comprised of those pixels.

A *relation* (or, more precisely, a *binary relation*) on a set A is a collection of ordered pairs of elements from A. That is, a binary relation is a subset of the Cartesian product $A \times A$. A binary relation between *two* sets, A and B, is a subset of $A \times B$.

A *partial order* on a set S is a relation $\mathcal{R}$ on S such that $\mathcal{R}$ is:

(a) *reflexive:* for any $a \in S$, $a\mathcal{R}a$;
(b) *transitive:* for any $a, b, c \in S$, $a\mathcal{R}b$ and $b\mathcal{R}c$ implies that $a\mathcal{R}c$;
(c) *antisymmetric:* for any $a, b \in S$, $a\mathcal{R}b$ and $b\mathcal{R}a$ implies that $a = b$.

where, for example, $a\mathcal{R}b$ reads "a is related to b." This means that a and b are in set $\mathcal{R}$, which itself is a subset of $S \times S$ according to the preceding definition of a relation. A set with a partial order is called a *partially ordered set*.

Let the symbol $\preceq$ denote an ordering relation. An expression of the form

$$a_1 \preceq a_2 \preceq a_3 \preceq \cdots \preceq a_n$$

reads: a_1 precedes a_2 or is the same as a_2, a_2 precedes a_3 or is the same as a_3, and so on. When working with numbers, the symbol $\preceq$ typically is replaced by more traditional symbols. For example, the set of real numbers ordered by the relation "less than or equal to" (denoted by $\leq$) is a partially ordered set (see Problem 2.33). Similarly, the set of natural numbers, paired with the relation "divisible by" (denoted by $\div$), is a partially ordered set.

Of more interest to us later in the book are strict orderings. A *strict ordering* on a set S is a relation $\mathcal{R}$ on S, such that $\mathcal{R}$ is:

(a) *antireflexive:* for any $a \in S$, $\neg a\mathcal{R}a$;
(b) *transitive:* for any $a, b, c \in S$, $a\mathcal{R}b$ and $b\mathcal{R}c$ implies that $a\mathcal{R}c$.

where $\neg a\mathcal{R}a$ means that a is *not related* to a. Let the symbol $\prec$ denote a strict ordering relation. An expression of the form

$$a_1 \prec a_2 \prec a_3 \prec \cdots \prec a_n$$

reads a_1 precedes a_2, a_2 precedes a_3, and so on. A set with a strict ordering is called a *strict-ordered set*.

As an example, consider the set composed of the English alphabet of lowercase letters, $S = \{a, b, c, \cdots, z\}$. Based on the preceding definition, the ordering

$$a \prec b \prec c \prec \cdots \prec z$$

is strict because no member of the set can precede itself (antireflexivity) and, for any three letters in S, if the first precedes the second, and the second precedes the third, then the first precedes the third (transitivity). Similarly, the set of integers paired with the relation "less than $(<)$" is a strict-ordered set.

Logical Operations

Logical operations deal with TRUE (typically denoted by 1) and FALSE (typically denoted by 0) variables and expressions. For our purposes, this means binary images composed of *foreground* (1-valued) pixels, and a *background* composed of 0-valued pixels.

We work with set and logical operators on binary images using one of two basic approaches: (1) we can use the *coordinates* of individual regions of foreground pixels in a single image as sets, or (2) we can work with one or more images of the same size and perform logical operations between corresponding pixels in those arrays.

In the first category, a binary image can be viewed as a Venn diagram in which the coordinates of individual regions of 1-valued pixels are treated as sets. The union of these sets with the set composed of 0-valued pixels comprises the set universe, Ω. In this representation, we work with single images using all the set operations defined in the previous section. For example, given a binary image with two 1-valued regions, R_1 and R_2, we can determine if the regions overlap (i.e., if they have at least one pair of coordinates in common) by performing the set intersection operation $R_1 \cap R_2$ (see Fig. 2.35). In the second approach, we perform logical operations on the pixels of one binary image, or on the corresponding pixels of two or more binary images of the same size.

Logical operators can be defined in terms of truth tables, as Table 2.2 shows for two logical variables a and b. The logical AND operation (also denoted $\wedge$) yields a 1 (TRUE) only when both *a and b* are 1. Otherwise, it yields 0 (FALSE). Similarly, the logical OR ($\vee$) yields 1 when both *a or b* or *both* are 1, and 0 otherwise. The NOT ($\sim$) operator is self explanatory. When applied to two binary images, AND and OR operate on pairs of corresponding pixels between the images. That is, they are elementwise operators (see the definition of elementwise operators given earlier in this chapter) in this context. The operators AND, OR, and NOT are *functionally complete*, in the sense that they can be used as the basis for constructing any other logical operator.

Figure 2.37 illustrates the logical operations defined in Table 2.2 using the second approach discussed above. The NOT of binary image B_1 is an array obtained by changing all 1-valued pixels to 0, and vice versa. The AND of B_1 and B_2 contains a 1 at all spatial locations where the corresponding elements of B_1 and B_2 are 1; the operation yields 0's elsewhere. Similarly, the OR of these two images is an array that contains a 1 in locations where the corresponding elements of B_1, *or* B_2, *or both*, are 1. The array contains 0's elsewhere. The result in the fourth row of Fig. 2.37 corresponds to the set of 1-valued pixels in B_1 but not in B_2. The last row in the figure is the XOR (exclusive OR) operation, which yields 1 in the locations where the corresponding elements of B_1 *or* B_2, (but *not both*) are 1. Note that the logical

TABLE 2.2
Truth table defining the logical operators AND($\wedge$), OR($\vee$), and NOT($\sim$).

a	b	a AND b	a OR b	NOT(a)
0	0	0	0	1
0	1	0	1	1
1	0	0	1	0
1	1	1	1	0

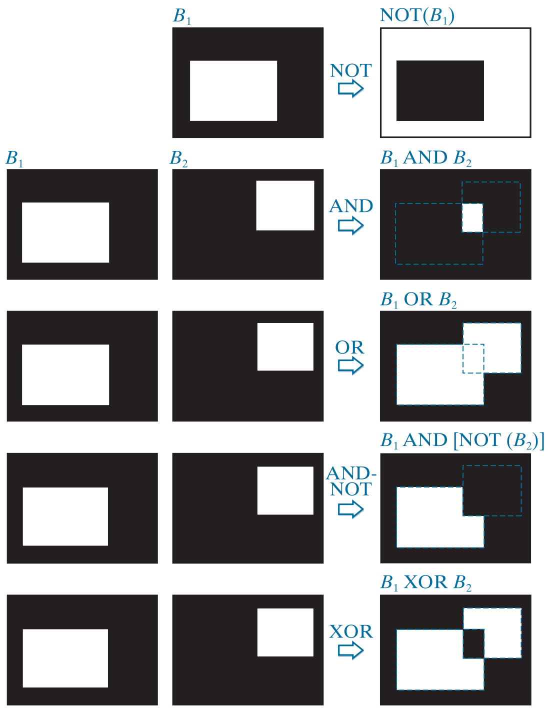


FIGURE 2.37 Illustration of logical operations involving foreground (white) pixels. Black represents binary 0's and white binary 1's. The dashed lines are shown for reference only. They are not part of the result.

expressions in the last two rows of Fig. 2.37 were constructed using operators from Table 2.2; these are examples of the functionally complete nature of these operators.

We can arrive at the same results in Fig. 2.37 using the first approach discussed above. To do this, we begin by labeling the individual 1-valued regions in each of the two images (in this case there is only one such region in each image). Let A and B denote the *set of coordinates* of all the 1-valued pixels in images B_1 and B_2, respectively. Then we form a *single* array by ORing the two images, while keeping the labels A and B. The result would look like the array B_1 OR B_2 in Fig. 2.37, but with the two white regions labeled A and B. In other words, the resulting array would look like a Venn diagram. With reference to the Venn diagrams and set operations defined in the previous section, we obtain the results in the rightmost column of Fig. 2.37 using set operations as follows: $A^c = \text{NOT}(B_1)$, $A \cap B = B_1 \text{ AND } B_2$, $A \cup B = B_1 \text{ OR } B_2$, and similarly for the other results in Fig. 2.37. We will make extensive use in Chapter 9 of the concepts developed in this section.

SPATIAL OPERATIONS

Spatial operations are performed directly on the pixels of an image. We classify spatial operations into three broad categories: (1) single-pixel operations, (2) neighborhood operations, and (3) geometric spatial transformations.

Single-Pixel Operations

The simplest operation we perform on a digital image is to alter the intensity of its pixels individually using a transformation function, T, of the form:

$$s = T(z) \tag{2-42}$$

where z is the intensity of a pixel in the original image and s is the (mapped) intensity of the corresponding pixel in the processed image. For example, Fig. 2.38 shows the transformation used to obtain the *negative* (sometimes called the *complement*) of an 8-bit image. This transformation could be used, for example, to obtain the negative image in Fig. 2.36, instead of using sets.

Our use of the word "negative" in this context refers to the digital equivalent of a photographic negative, not to the numerical negative of the pixels in the image.

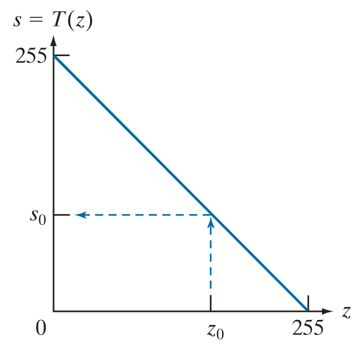
Neighborhood Operations

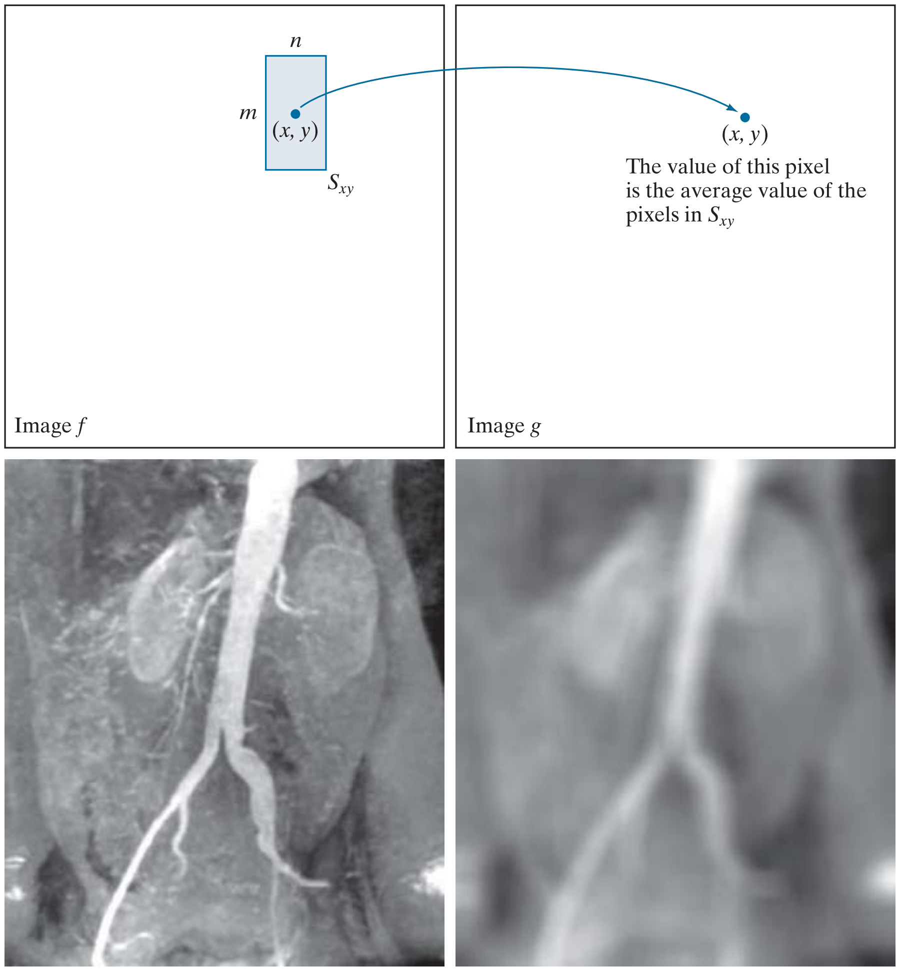
Let S_{xy} denote the set of coordinates of a neighborhood (see Section 2.5 regarding neighborhoods) centered on an arbitrary point (x, y) in an image, f. Neighborhood processing generates a corresponding pixel at the same coordinates in an output (processed) image, g, such that the value of that pixel is determined by a specified operation on the neighborhood of pixels in the input image with coordinates in the set S_{xy}. For example, suppose that the specified operation is to compute the average value of the pixels in a rectangular neighborhood of size $m \times n$ centered on (x, y). The coordinates of pixels in this region are the elements of set S_{xy}. Figures 2.39(a) and (b) illustrate the process. We can express this averaging operation as

$$g(x, y) = \frac{1}{mn} \sum_{(r,c) \in S_{xy}} f(r, c) \tag{2-43}$$

where r and c are the row and column coordinates of the pixels whose coordinates are in the set S_{xy}. Image g is created by varying the coordinates (x, y) so that the center of the neighborhood moves from pixel to pixel in image f, and then repeating the neighborhood operation at each new location. For instance, the image in Fig. 2.39(d) was created in this manner using a neighborhood of size 41×41. The

FIGURE 2.38 Intensity transformation function used to obtain the digital equivalent of photographic negative of an 8-bit image..




n
m
(x, y)
S_{xy}
(x, y)
The value of this pixel is the average value of the pixels in S_{xy}
Image f
Image g

a b
c d

FIGURE 2.39 Local averaging using neighborhood processing. The procedure is illustrated in (a) and (b) for a rectangular neighborhood. (c) An aortic angiogram (see Section 1.3). (d) The result of using Eq. (2-43) with $m = n = 41$. The images are of size 790×686 pixels. (Original image courtesy of Dr. Thomas R. Gest, Division of Anatomical Sciences, University of Michigan Medical School.)

net effect is to perform local blurring in the original image. This type of process is used, for example, to eliminate small details and thus render "blobs" corresponding to the largest regions of an image. We will discuss neighborhood processing in Chapters 3 and 5, and in several other places in the book.

Geometric Transformations

We use geometric transformations modify the spatial arrangement of pixels in an image. These transformations are called *rubber-sheet transformations* because they may be viewed as analogous to "printing" an image on a rubber sheet, then stretching or shrinking the sheet according to a predefined set of rules. Geometric transformations of digital images consist of two basic operations:

1. Spatial transformation of coordinates.
2. Intensity interpolation that assigns intensity values to the spatially transformed pixels.

The transformation of coordinates may be expressed as

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \mathbf{T}\begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} t_{11} & t_{12} \\ t_{21} & t_{22} \end{bmatrix}\begin{bmatrix} x \\ y \end{bmatrix} \tag{2-44}$$

where (x, y) are pixel coordinates in the original image and (x', y') are the corresponding pixel coordinates of the transformed image. For example, the transformation $(x', y') = (x/2, y/2)$ shrinks the original image to half its size in both spatial directions.

Our interest is in so-called *affine transformations*, which include scaling, translation, rotation, and shearing. The key characteristic of an affine transformation in 2-D is that it preserves points, straight lines, and planes. Equation (2-44) can be used to express the transformations just mentioned, except translation, which would require that a constant 2-D vector be added to the right side of the equation. However, it is possible to use homogeneous coordinates to express all four affine transformations using a single 3×3 matrix in the following general form:

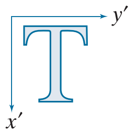
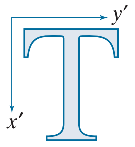
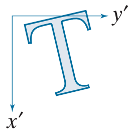
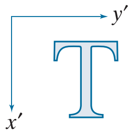
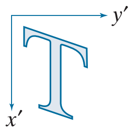
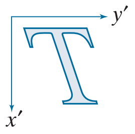
$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \mathbf{A}\begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ 0 & 0 & 1 \end{bmatrix}\begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \tag{2-45}$$

This transformation can *scale*, *rotate*, *translate*, or *sheer* an image, depending on the values chosen for the elements of matrix **A**. Table 2.3 shows the matrix values used to implement these transformations. A significant advantage of being able to perform all transformations using the unified representation in Eq. (2-45) is that it provides the framework for concatenating a sequence of operations. For example, if we want to resize an image, rotate it, and move the result to some location, we simply form a 3×3 matrix equal to the product of the scaling, rotation, and translation matrices from Table 2.3 (see Problems 2.36 and 2.37).

The preceding transformation moves the coordinates of pixels in an image to new locations. To complete the process, we have to assign intensity values to those locations. This task is accomplished using *intensity interpolation*. We already discussed this topic in Section 2.4. We began that discussion with an example of zooming an image and discussed the issue of intensity assignment to new pixel locations. Zooming is simply scaling, as detailed in the second row of Table 2.3, and an analysis similar to the one we developed for zooming is applicable to the problem of assigning intensity values to the relocated pixels resulting from the other transformations in Table 2.3. As in Section 2.4, we consider nearest neighbor, bilinear, and bicubic interpolation techniques when working with these transformations.

We can use Eq. (2-45) in two basic ways. The first, is a *forward mapping*, which consists of scanning the pixels of the input image and, at each location (x, y), computing the spatial location (x', y') of the corresponding pixel in the output image using Eq. (2-45) directly. A problem with the forward mapping approach is that two or more pixels in the input image can be transformed to the same location in the output image, raising the question of how to combine multiple output values into a single output pixel value. In addition, it is possible that some output locations may not be assigned a pixel at all. The second approach, called *inverse mapping*, scans the output pixel locations and, at each location (x', y'), computes the corresponding location in the input image using $(x, y) = A^{-1}(x', y')$. It then interpolates (using one of the techniques discussed in Section 2.4) among the nearest input pixels to determine the intensity of the output pixel value. Inverse mappings are more efficient to implement than forward mappings, and are used in numerous commercial implementations of spatial transformations (for example, MATLAB uses this approach).

TABLE 2.3 Affine transformations based on Eq. (2-45).

Transformation Name	Affine Matrix, A	Coordinate Equations	Example
Identity	$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$	$x' = x$ $y' = y$	
Scaling/Reflection (For reflection, set one scaling factor to −1 and the other to 0)	$\begin{bmatrix} c_x & 0 & 0 \\ 0 & c_y & 0 \\ 0 & 0 & 1 \end{bmatrix}$	$x' = c_x x$ $y' = c_y y$	
Rotation (about the origin)	$\begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}$	$x' = x\cos\theta - y\sin\theta$ $y' = x\sin\theta + y\cos\theta$	
Translation	$\begin{bmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \\ 0 & 0 & 1 \end{bmatrix}$	$x' = x + t_x$ $y' = y + t_y$	
Shear (vertical)	$\begin{bmatrix} 1 & s_v & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$	$x' = x + s_v y$ $y' = y$	
Shear (horizontal)	$\begin{bmatrix} 1 & 0 & 0 \\ s_h & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$	$x' = x$ $y' = s_h x + y$	

EXAMPLE 2.9: Image rotation and intensity interpolation.

The objective of this example is to illustrate image rotation using an affine transform. Figure 2.40(a) shows a simple image and Figs. 2.40(b)–(d) are the results (using inverse mapping) of rotating the original image by $-21°$ (in Table 2.3, clockwise angles of rotation are negative). Intensity assignments were computed using nearest neighbor, bilinear, and bicubic interpolation, respectively. A key issue in image rotation is the preservation of straight-line features. As you can see in the enlarged edge sections in Figs. 2.40(f) through (h), nearest neighbor interpolation produced the most jagged edges and, as in Section 2.4, bilinear interpolation yielded significantly improved results. As before, using bicubic interpolation produced slightly better results. In fact, if you compare the progression of enlarged detail in Figs. 2.40(f) to (h), you can see that the transition from white (255) to black (0) is smoother in the last figure because the edge region has more values, and the distribution of those values is better balanced. Although the small intensity differences resulting from bilinear and bicubic interpolation are not always noticeable in human visual analysis, they can be important in processing image data, such as in automated edge following in rotated images.

The size of the spatial rectangle needed to contain a rotated image is larger than the rectangle of the original image, as Figs. 2.41(a) and (b) illustrate. We have two options for dealing with this: (1) we can crop the rotated image so that its size is equal to the size of the original image, as in Fig. 2.41(c), or we can keep the larger image containing the full rotated original, an Fig. 2.41(d). We used the first option in Fig. 2.40 because the rotation did not cause the object of interest to lie outside the bounds of the original rectangle. The areas in the rotated image that do not contain image data must be filled with some value, 0 (black) being the most common. Note that counterclockwise angles of rotation are considered positive. This is a result of the way in which our image coordinate system is set up (see Fig. 2.19), and the way in which rotation is defined in Table 2.3.

Image Registration

Image registration is an important application of digital image processing used to align two or more images of the same scene. In image registration, we have available an *input* image and a *reference* image. The objective is to transform the input image geometrically to produce an output image that is aligned (registered) with the reference image. Unlike the discussion in the previous section where transformation functions are known, the geometric transformation needed to produce the output, registered image generally is not known, and must be estimated.

Examples of image registration include aligning two or more images taken at approximately the same time, but using different imaging systems, such as an MRI (magnetic resonance imaging) scanner and a PET (positron emission tomography) scanner. Or, perhaps the images were taken at different times using the same instruments, such as satellite images of a given location taken several days, months, or even years apart. In either case, combining the images or performing quantitative analysis and comparisons between them requires compensating for geometric distortions caused by differences in viewing angle, distance, orientation, sensor resolution, shifts in object location, and other factors.

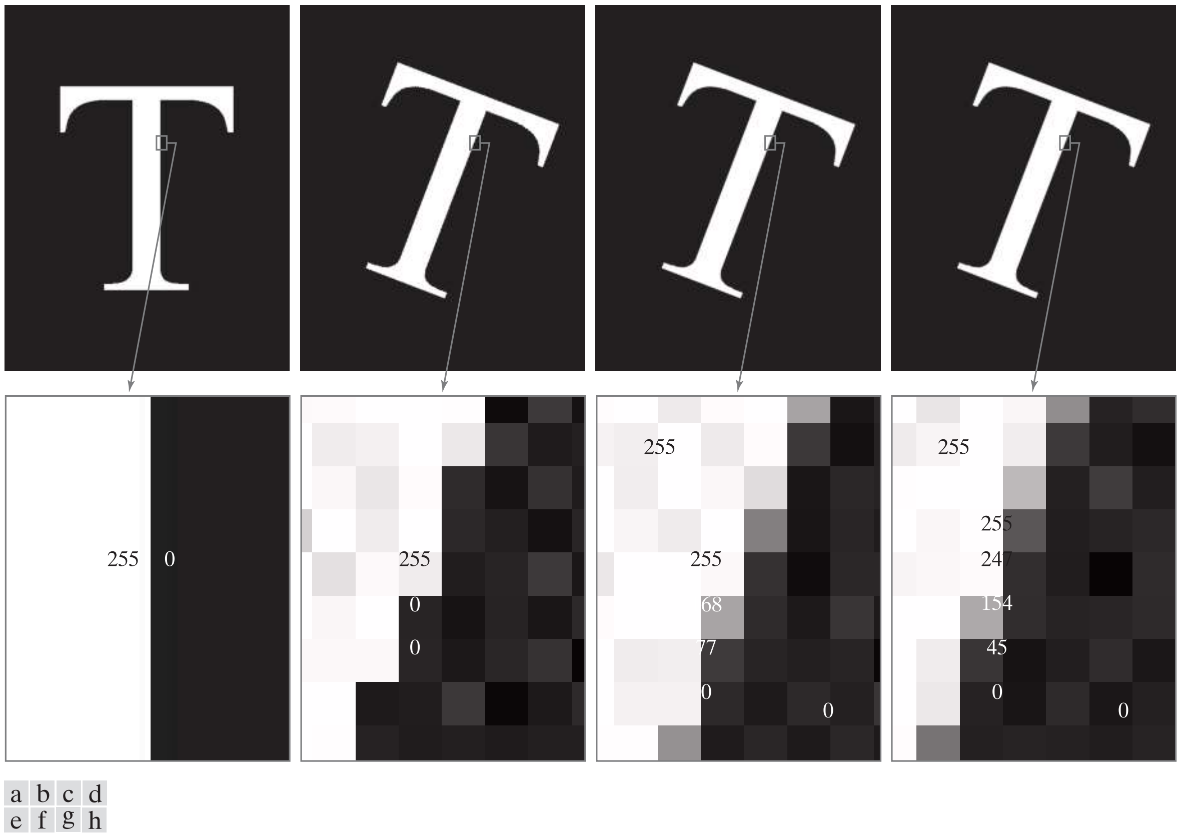


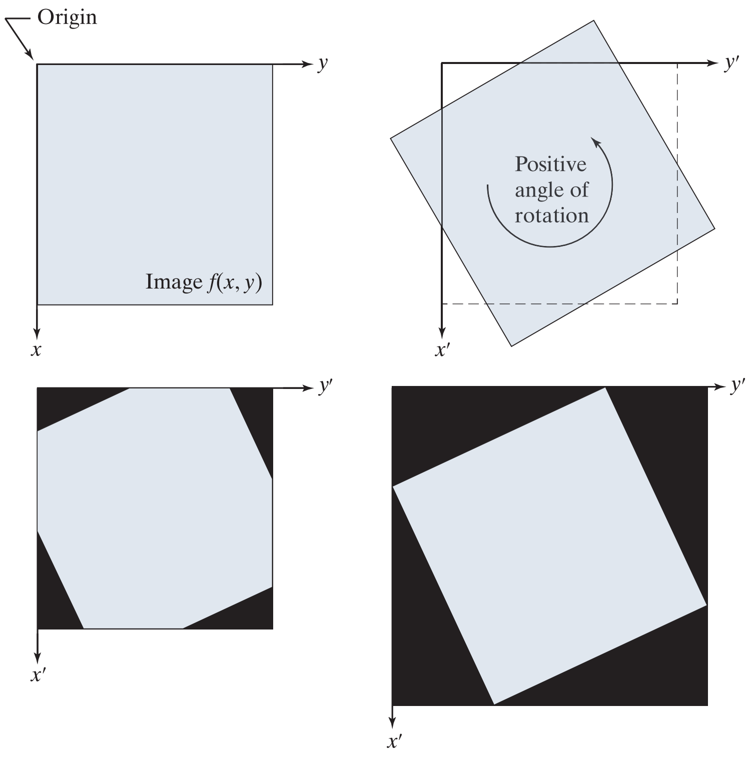
FIGURE 2.40 (a) A 541 × 421 image of the letter T. (b) Image rotated –21° using nearest-neighbor interpolation for intensity assignments. (c) Image rotated –21° using bilinear interpolation. (d) Image rotated –21° using bicubic interpolation. (e)-(h) Zoomed sections (each square is one pixel, and the numbers shown are intensity values).

One of the principal approaches for solving the problem just discussed is to use *tie points* (also called *control points*). These are corresponding points whose locations are known precisely in the input and reference images. Approaches for selecting tie points range from selecting them interactively to using algorithms that detect these points automatically. Some imaging systems have physical artifacts (such as small metallic objects) embedded in the imaging sensors. These produce a set of known points (called *reseau marks* or *fiducial marks*) directly on all images captured by the system. These known points can then be used as guides for establishing tie points.

The problem of estimating the transformation function is one of modeling. For example, suppose that we have a set of four tie points each in an input and a reference image. A simple model based on a bilinear approximation is given by

$$x = c_1 v + c_2 w + c_3 vw + c_4 \tag{2-46}$$

and



a b
c d

FIGURE 2.41 (a) A digital image. (b) Rotated image (note the counterclockwise direction for a positive angle of rotation). (c) Rotated image cropped to fit the same area as the original image. (d) Image enlarged to accommodate the entire rotated image.

$$y = c_5 v + c_6 w + c_7 vw + c_8 \tag{2-47}$$

During the estimation phase, (v, w) and (x, y) are the coordinates of tie points in the input and reference images, respectively. If we have four pairs of corresponding tie points in both images, we can write eight equations using Eqs. (2-46) and (2-47) and use them to solve for the eight unknown coefficients, c_1 through c_8.

Once we have the coefficients, Eqs. (2-46) and (2-47) become our vehicle for transforming all the pixels in the input image. The result is the desired registered image. After the coefficients have been computed, we let (v, w) denote the coordinates of each pixel in the input image, and (x, y) become the corresponding coordinates of the output image. The same set of coefficients, c_1 through c_8, are used in computing all coordinates (x, y); we just step through all (v, w) in the input image to generate the corresponding (x, y) in the output, registered image. If the tie points were selected correctly, this new image should be registered with the reference image, within the accuracy of the bilinear approximation model.

In situations where four tie points are insufficient to obtain satisfactory registration, an approach used frequently is to select a larger number of tie points and then treat the quadrilaterals formed by groups of four tie points as subimages. The subimages are processed as above, with all the pixels within a quadrilateral being transformed using the coefficients determined from the tie points corresponding to that quadrilateral. Then we move to another set of four tie points and repeat the procedure until all quadrilateral regions have been processed. It is possible to use more complex regions than quadrilaterals, and to employ more complex models, such as polynomials fitted by least squares algorithms. The number of control points and sophistication of the model required to solve a problem is dependent on the severity of the geometric distortion. Finally, keep in mind that the transformations defined by Eqs. (2-46) and (2-47), or any other model for that matter, only map the spatial coordinates of the pixels in the input image. We still need to perform intensity interpolation using any of the methods discussed previously to assign intensity values to the transformed pixels.

EXAMPLE 2.10: Image registration.

Figure 2.42(a) shows a reference image and Fig. 2.42(b) shows the same image, but distorted geometrically by vertical and horizontal shear. Our objective is to use the reference image to obtain tie points and then use them to register the images. The tie points we selected (manually) are shown as small white squares near the corners of the images (we needed only four tie points because the distortion is linear shear in both directions). Figure 2.42(c) shows the registration result obtained using these tie points in the procedure discussed in the preceding paragraphs. Observe that registration was not perfect, as is evident by the black edges in Fig. 2.42(c). The difference image in Fig. 2.42(d) shows more clearly the slight lack of registration between the reference and corrected images. The reason for the discrepancies is error in the manual selection of the tie points. It is difficult to achieve perfect matches for tie points when distortion is so severe.

VECTOR AND MATRIX OPERATIONS

Multispectral image processing is a typical area in which vector and matrix operations are used routinely. For example, you will learn in Chapter 6 that color images are formed in RGB color space by using red, green, and blue component images, as Fig. 2.43 illustrates. Here we see that *each* pixel of an RGB image has three components, which can be organized in the form of a column vector

$$\mathbf{z} = \begin{bmatrix} z_1 \\ z_2 \\ z_3 \end{bmatrix} \tag{2-48}$$

where z_1 is the intensity of the pixel in the red image, and z_2 and z_3 are the corresponding pixel intensities in the green and blue images, respectively. Thus, an RGB color image of size $M \times N$ can be represented by three component images of this size, or by a total of MN vectors of size 3×1. A general multispectral case involving n component images (e.g., see Fig. 1.10) will result in n-dimensional vectors:

Recall that an n-dimensional vector can be thought of as a point in n-dimensional Euclidean space.

$$\mathbf{z} = \begin{bmatrix} z_1 \\ z_2 \\ \vdots \\ z_n \end{bmatrix} \tag{2-49}$$

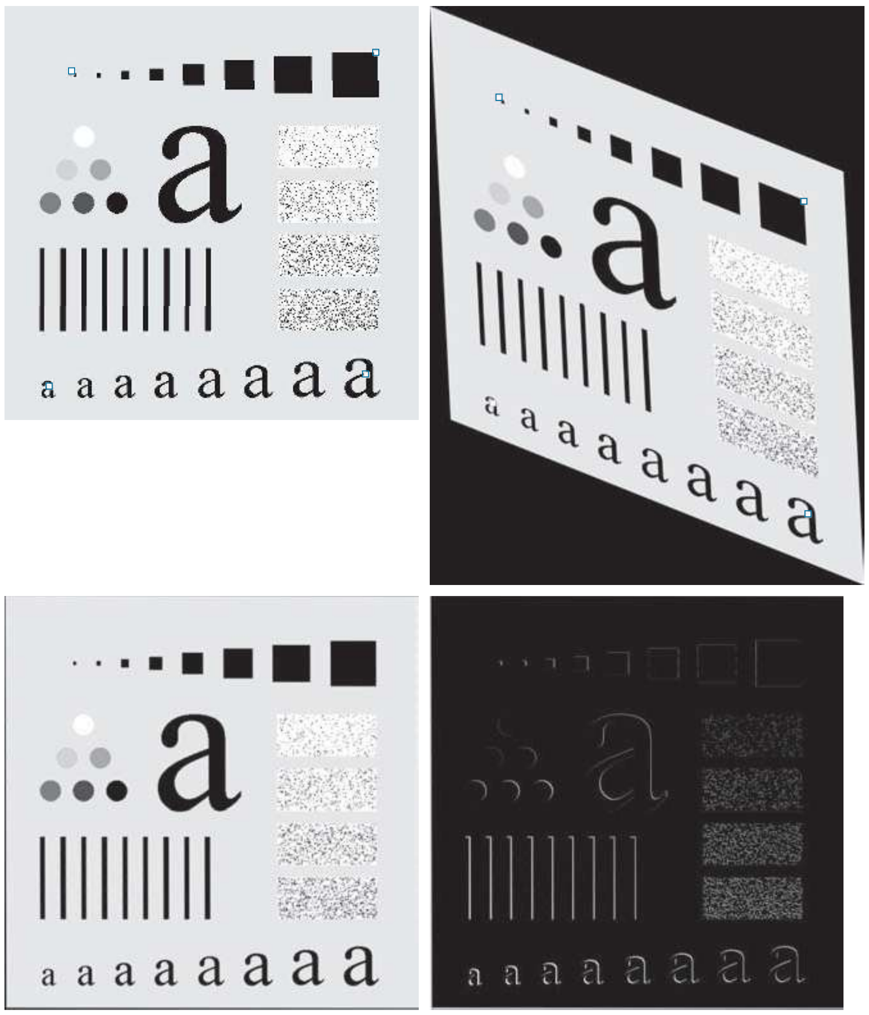

a b
c d

FIGURE 2.42 Image registration. (a) Reference image. (b) Input (geometrically distorted image). Corresponding tie points are shown as small white squares near the corners. (c) Registered (output) image (note the errors in the border). (d) Difference between (a) and (c), showing more registration errors.

We will use this type of vector representation throughout the book.

The *inner product* (also called the *dot product*) of two n-dimensional column vectors $\mathbf{a}$ and $\mathbf{b}$ is defined as

The product $\mathbf{ab}^T$ is called the *outer product* of $\mathbf{a}$ and $\mathbf{b}$. It is a matrix of size $n \times n$.

$$\begin{aligned}\mathbf{a} \bullet \mathbf{b} &\triangleq \mathbf{a}^T\mathbf{b} \\ &= a_1b_1 + a_2b_2 + \cdots + a_nb_n \\ &= \sum_{i=1}^{n} a_i b_i\end{aligned} \tag{2-50}$$

where T indicates the transpose. The *Euclidean vector norm*, denoted by $\|\mathbf{z}\|$, is defined as the square root of the inner product:

$$\|\mathbf{z}\| = \left(\mathbf{z}^T\mathbf{z}\right)^{\frac{1}{2}} \tag{2-51}$$

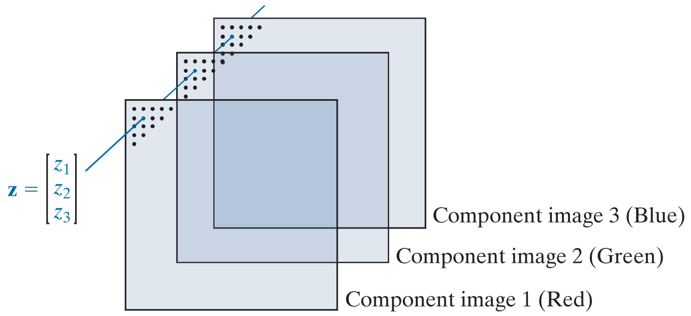

$\mathbf{z} = \begin{bmatrix} z_1 \\ z_2 \\ z_3 \end{bmatrix}$
Component image 3 (Blue)
Component image 2 (Green)
Component image 1 (Red)

FIGURE 2.43 Forming a vector from corresponding pixel values in three RGB component images.

We recognize this expression as the length of vector **z**.

We can use vector notation to express several of the concepts discussed earlier. For example, the Euclidean distance, $D(\mathbf{z}, \mathbf{a})$, between points (vectors) **z** and **a** in n-dimensional space is defined as the Euclidean vector norm:

$$\begin{aligned} D(\mathbf{z},\mathbf{a}) = \|\mathbf{z}-\mathbf{a}\| &= \left[(\mathbf{z}-\mathbf{a})^T(\mathbf{z}-\mathbf{a})\right]^{\frac{1}{2}} \\ &= \left[(z_1-a_1)^2+(z_2-a_2)^2+\cdots+(z_n-a_n)^2\right]^{\frac{1}{2}} \end{aligned} \tag{2-52}$$

This is a generalization of the 2-D Euclidean distance defined in Eq. (2-19).

Another advantage of pixel vectors is in linear transformations, represented as

$$\mathbf{w} = \mathbf{A}(\mathbf{z}-\mathbf{a}) \tag{2-53}$$

where **A** is a matrix of size $m \times n$, and **z** and **a** are column vectors of size $n \times 1$.

As noted in Eq. (2-10), entire images can be treated as matrices (or, equivalently, as vectors), a fact that has important implication in the solution of numerous image processing problems. For example, we can express an image of size $M \times N$ as a column vector of dimension $MN \times 1$ by letting the first M elements of the vector equal the first column of the image, the next M elements equal the second column, and so on. With images formed in this manner, we can express a broad range of linear processes applied to an image by using the notation

$$\mathbf{g} = \mathbf{Hf} + \mathbf{n} \tag{2-54}$$

where **f** is an $MN \times 1$ vector representing an input image, **n** is an $MN \times 1$ vector representing an $M \times N$ noise pattern, **g** is an $MN \times 1$ vector representing a processed image, and **H** is an $MN \times MN$ matrix representing a linear process applied to the input image (see the discussion earlier in this chapter regarding linear processes). It is possible, for example, to develop an entire body of generalized techniques for image restoration starting with Eq. (2-54), as we discuss in Section 5.9. We will mention the use of matrices again in the following section, and show other uses of matrices for image processing in numerous chapters in the book.

IMAGE TRANSFORMS

All the image processing approaches discussed thus far operate directly on the pixels of an input image; that is, they work directly in the spatial domain. In some cases, image processing tasks are best formulated by transforming the input images, carrying the specified task in a *transform domain*, and applying the inverse transform to return to the spatial domain. You will encounter a number of different transforms as you proceed through the book. A particularly important class of 2-D *linear transforms*, denoted $T(u,v)$, can be expressed in the general form

$$T(u,v) = \sum_{x=0}^{M-1}\sum_{y=0}^{N-1} f(x,y)r(x,y,u,v) \tag{2-55}$$

where $f(x,y)$ is an input image, $r(x,y,u,v)$ is called a *forward transformation kernel*, and Eq. (2-55) is evaluated for $u = 0,1,2,\dots,M-1$ and $v = 0,1,2,\dots,N-1$. As before, x and y are spatial variables, while M and N are the row and column dimensions of f. Variables u and v are called the *transform variables*. $T(u,v)$ is called the *forward transform* of $f(x,y)$. Given $T(u,v)$, we can recover $f(x,y)$ using the *inverse transform* of $T(u,v)$:

$$f(x,y) = \sum_{u=0}^{M-1}\sum_{v=0}^{N-1} T(u,v)s(x,y,u,v) \tag{2-56}$$

for $x = 0,1,2,\dots,M-1$ and $y = 0,1,2,\dots,N-1$, where $s(x,y,u,v)$ is called an *inverse transformation kernel*. Together, Eqs. (2-55) and (2-56) are called a *transform pair*.

Figure 2.44 shows the basic steps for performing image processing in the linear transform domain. First, the input image is transformed, the transform is then modified by a predefined operation and, finally, the output image is obtained by computing the inverse of the modified transform. Thus, we see that the process goes from the spatial domain to the transform domain, and then back to the spatial domain.

The forward transformation kernel is said to be *separable* if

$$r(x,y,u,v) = r_1(x,u)r_2(y,v) \tag{2-57}$$

In addition, the kernel is said to be *symmetric* if $r_1(x,u)$ is functionally equal to $r_2(y,v)$, so that

$$r(x,y,u,v) = r_1(x,u)r_1(y,v) \tag{2-58}$$

Identical comments apply to the inverse kernel.

The nature of a transform is determined by its kernel. A transform of particular importance in digital image processing is the *Fourier transform*, which has the following forward and inverse kernels:

$$r(x,y,u,v) = e^{-j2\pi(ux/M + vy/N)} \tag{2-59}$$

and

$$s(x,y,u,v) = \frac{1}{MN}e^{j2\pi(ux/M + vy/N)} \tag{2-60}$$

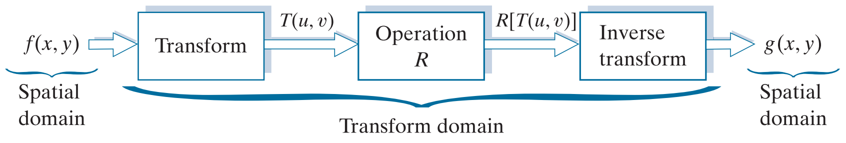


FIGURE 2.44 General approach for working in the linear transform domain.

respectively, where $j = \sqrt{-1}$, so these kernels are complex functions. Substituting the preceding kernels into the general transform formulations in Eqs. (2-55) and (2-56) gives us the *discrete Fourier transform pair*:

The exponential terms in the Fourier transform kernels can be expanded as sines and cosines of various frequencies. As a result, the domain of the Fourier transform is called the *frequency domain*.

$$T(u,v) = \sum_{x=0}^{M-1} \sum_{y=0}^{N-1} f(x,y) e^{-j2\pi(ux/M \,+\, vy/N)} \tag{2-61}$$

and

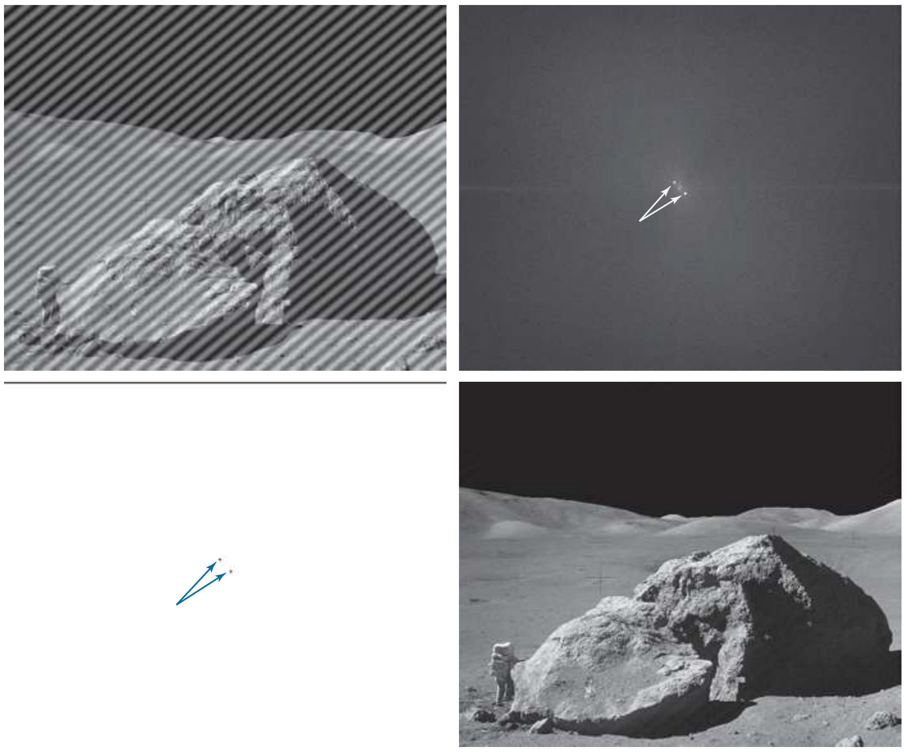
$$f(x,y) = \frac{1}{MN} \sum_{u=0}^{M-1} \sum_{v=0}^{N-1} T(u,v) e^{j2\pi(ux/M \,+\, vy/N)} \tag{2-62}$$

It can be shown that the Fourier kernels are separable and symmetric (Problem 2.39), and that separable and symmetric kernels allow 2-D transforms to be computed using 1-D transforms (see Problem 2.40). The preceding two equations are of fundamental importance in digital image processing, as you will see in Chapters 4 and 5.

EXAMPLE 2.11: Image processing in the transform domain.

Figure 2.45(a) shows an image corrupted by periodic (sinusoidal) interference. This type of interference can be caused, for example, by a malfunctioning imaging system; we will discuss it in Chapter 5. In the spatial domain, the interference appears as waves of intensity. In the frequency domain, the interference manifests itself as bright bursts of intensity, whose location is determined by the frequency of the sinusoidal interference (we will discuss these concepts in much more detail in Chapters 4 and 5). Typically, the bursts are easily observable in an image of the magnitude of the Fourier transform, $|T(u,v)|$. With reference to the diagram in Fig. 2.44, the corrupted image is $f(x,y)$, the transform in the leftmost box is the Fourier transform, and Fig. 2.45(b) is $|T(u,v)|$ displayed as an image. The bright dots shown are the bursts of intensity mentioned above. Figure 2.45(c) shows a mask image (called a *filter*) with white and black representing 1 and 0, respectively. For this example, the operation in the second box of Fig. 2.44 is to multiply the filter by the transform to remove the bursts associated with the interference. Figure 2.45(d) shows the final result, obtained by computing the inverse of the modified transform. The interference is no longer visible, and previously unseen image detail is now made quite clear. Observe, for example, the fiducial marks (faint crosses) that are used for image registration, as discussed earlier.

When the forward and inverse kernels of a transform are separable and symmetric, and $f(x,y)$ is a square image of size $M \times M$, Eqs. (2-55) and (2-56) can be expressed in matrix form:



a b
c d

FIGURE 2.45 (a) Image corrupted by sinusoidal interference. (b) Magnitude of the Fourier transform showing the bursts of energy caused by the interference (the bursts were enlarged for display purposes). (c) Mask used to eliminate the energy bursts. (d) Result of computing the inverse of the modified Fourier transform. (Original image courtesy of NASA.)

$$\mathbf{T} = \mathbf{AFA} \tag{2-63}$$

where **F** is an $M \times M$ matrix containing the elements of $f(x, y)$ [see Eq. (2-9)], **A** is an $M \times M$ matrix with elements $a_{ij} = r_1(i, j)$, and **T** is an $M \times M$ transform matrix with elements $T(u, v)$, for $u, v = 0, 1, 2, \ldots, M-1$.

To obtain the inverse transform, we pre- and post-multiply Eq. (2-63) by an inverse transformation matrix **B**:

$$\mathbf{BTB} = \mathbf{BAFAB} \tag{2-64}$$

If $\mathbf{B} = \mathbf{A}^{-1}$,

$$\mathbf{F} = \mathbf{BTB} \tag{2-65}$$

indicating that **F** or, equivalently, $f(x, y)$, can be recovered completely from its forward transform. If **B** is not equal to $\mathbf{A}^{-1}$, Eq. (2-65) yields an approximation:

$$\hat{\mathbf{F}} = \mathbf{BAFAB} \tag{2-66}$$

In addition to the Fourier transform, a number of important transforms, including the *Walsh*, *Hadamard*, *discrete cosine*, *Haar*, and *slant* transforms, can be expressed in the form of Eqs. (2-55) and (2-56), or, equivalently, in the form of Eqs. (2-63) and (2-65). We will discuss these and other types of image transforms in later chapters.