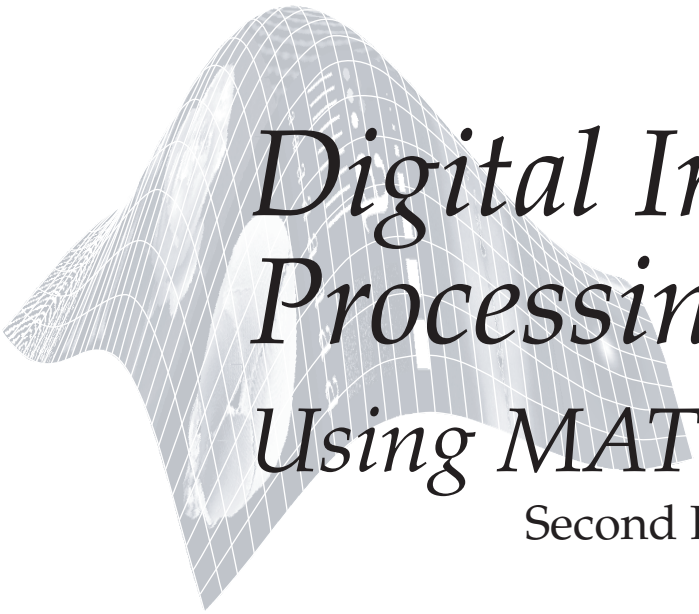


Digital Image Processing Using MATLAB®

Second Edition

Rafael C. Gonzalez
University of Tennessee

Richard E. Woods
MedData Interactive

Steven L. Eddins
The MathWorks, Inc.

Gatesmark Publishing®
A Division of Gatesmark,® LLC
www.gatesmark.com

Library of Congress Cataloging-in-Publication Data on File

Library of Congress Control Number: 2009902793

Gatesmark Publishing
A Division of Gatesmark, LLC
www.gatesmark.com

Printed in the United States of America
10 9 8 7 6 5 4 3 2 1

ISBN 978-0-9820854-0-0

3 Intensity Transformations and Spatial Filtering

Preview

The term *spatial domain* refers to the image plane itself, and methods in this category are based on direct manipulation of pixels in an image. In this chapter we focus attention on two important categories of spatial domain processing: *intensity* (*gray-level*) *transformations* and *spatial filtering*. The latter approach sometimes is referred to as *neighborhood processing*, or *spatial convolution.* In the following sections we develop and illustrate MATLAB formulations representative of processing techniques in these two categories. We also introduce the concept of fuzzy image processing and develop several new M-functions for their implementation. In order to carry a consistent theme, most of the examples in this chapter are related to image enhancement. This is a good way to introduce spatial processing because enhancement is highly intuitive and appealing, especially to beginners in the field. As you will see throughout the book, however, these techniques are general in scope and have uses in numerous other branches of digital image processing.

3.1 Background

As noted in the preceding paragraph, spatial domain techniques operate directly on the pixels of an image. The spatial domain processes discussed in this chapter are denoted by the expression

$$g(x, y) = T\left[f(x, y)\right]$$

where $f(x, y)$ is the input image, $g(x, y)$ is the output (processed) image, and T is an operator on f defined over a specified neighborhood about point (x, y). In addition, T can operate on a set of images, such as performing the addition of K images for noise reduction.

The principal approach for defining spatial neighborhoods about a point (x, y) is to use a square or rectangular region *centered* at (x, y), as in Fig. 3.1. The center of the region is moved from pixel to pixel starting, say, at the top, left corner, and, as it moves, it encompasses different neighborhoods. Operator T is applied at each location (x, y) to yield the output, g, at that location. Only the pixels in the neighborhood centered at (x, y) are used in computing the value of g at (x, y).

Most of the remainder of this chapter deals with various implementations of the preceding equation. Although this equation is simple conceptually, its computational implementation in MATLAB requires that careful attention be paid to data classes and value ranges.

3.2 Intensity Transformation Functions

The simplest form of the transformation T is when the neighborhood in Fig. 3.1 is of size 1×1 (a single pixel). In this case, the value of g at (x, y) depends only on the intensity of f at that point, and T becomes an *intensity* or *gray-level transformation function*. These two terms are used interchangeably when dealing with monochrome (i.e., gray-scale) images. When dealing with color images, the term *intensity* is used to denote a color image component in certain color spaces, as described in Chapter 7.

Because the output value depends only on the intensity value at a point, and not on a neighborhood of points, intensity transformation functions frequently are written in simplified form as

$$s = T(r)$$

where r denotes the intensity of f and s the intensity of g, both at the same coordinates (x, y) in the images.

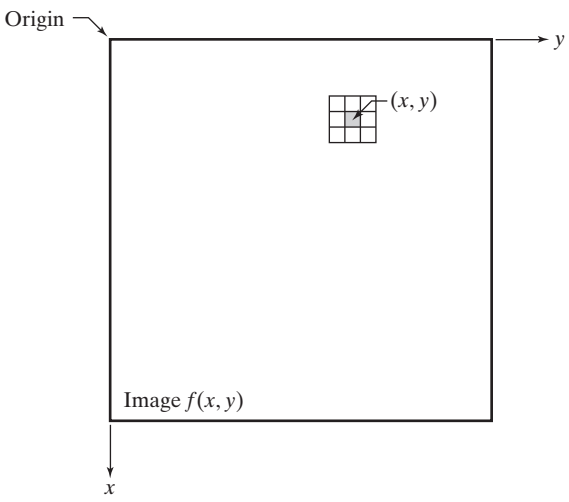


FIGURE 3.1 A neighborhood of size 3×3 centered at point (x, y) in an image.

3.2.1 Functions `imadjust` and `stretchlim`

Function `imadjust` is the basic Image Processing Toolbox function for intensity transformations of gray-scale images. It has the general syntax


```
g = imadjust(f, [low_in high_in], [low_out high_out], gamma)
```

As Fig. 3.2 illustrates, this function maps the intensity values in image `f` to new values in `g`, such that values between `low_in` and `high_in` map to values between `low_out` and `high_out`. Values below `low_in` and above `high_in` are clipped; that is, values below `low_in` map to `low_out`, and those above `high_in` map to `high_out`. The input image can be of class `uint8`, `uint16`, `int16`, `single`, or `double`, and the output image has the same class as the input. All inputs to function `imadjust`, other than `f` and `gamma`, are specified as values between 0 and 1, independently of the class of `f`. If, for example, `f` is of class `uint8`, `imadjust` multiplies the values supplied by 255 to determine the actual values to use. Using the empty matrix (`[ ]`) for `[low_in high_in]` or for `[low_out high_out]` results in the default values `[0 1]`. If `high_out` is less than `low_out`, the output intensity is reversed.

Recall from the discussion in Section 2.7 that function `mat2gray` can be used for converting an image to class `double` and scaling its intensities to the range [0, 1], independently of the class of the input image.

Parameter `gamma` specifies the shape of the curve that maps the intensity values in `f` to create `g`. If `gamma` is less than 1, the mapping is weighted toward higher (brighter) output values, as in Fig. 3.2(a). If `gamma` is greater than 1, the mapping is weighted toward lower (darker) output values. If it is omitted from the function argument, `gamma` defaults to 1 (linear mapping).

EXAMPLE 3.1: Using function `imadjust`.

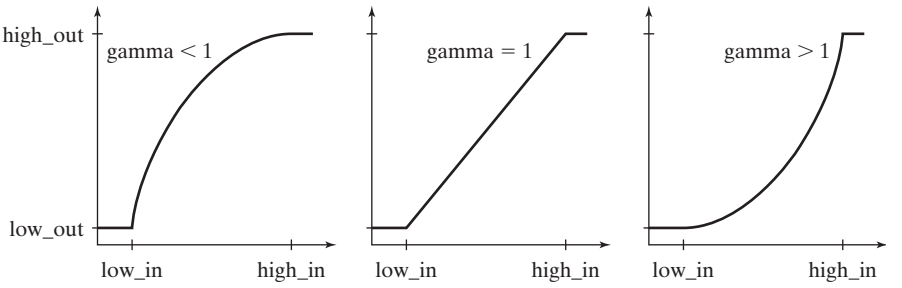
■ Figure 3.3(a) is a digital mammogram image, `f`, showing a small lesion, and Fig. 3.3(b) is the negative image, obtained using the command

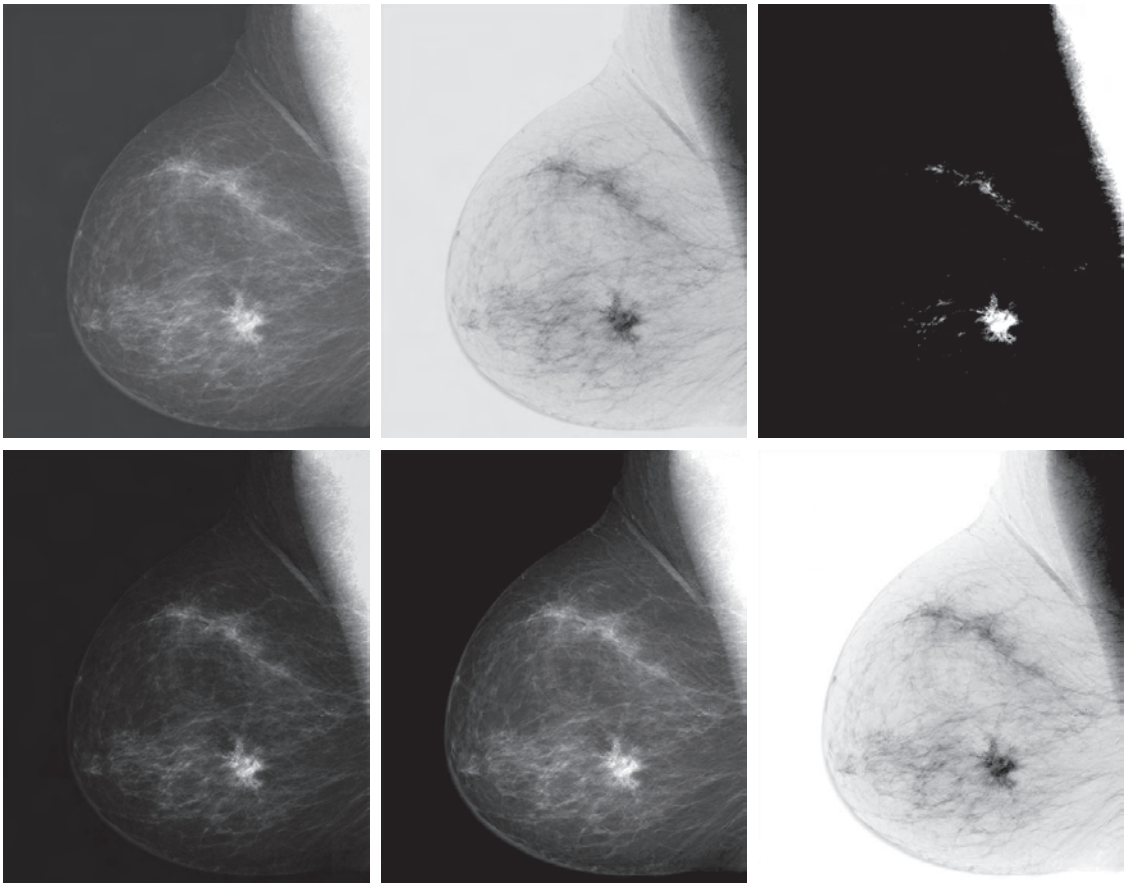
```
>> g1 = imadjust(f, [0 1], [1 0]);
```

This process, which is the digital equivalent of obtaining a photographic negative, is particularly useful for enhancing white or gray detail embedded in a large, predominantly dark region. Note, for example, how much easier it is to analyze the breast tissue in Fig. 3.3(b). The negative of an image can be obtained also with toolbox function `imcomplement`:

a b c

FIGURE 3.2 The various mappings available in function `imadjust`.





a b c
d e f

FIGURE 3.3 (a) Original digital mammogram. (b) Negative image. (c) Result of expanding the intensities in the range [0.5, 0.75]. (d) Result of enhancing the image with `gamma` = 2. (e) and (f) Results of using function `stretchlim` as an automatic input into function `imadjust`. (Original image courtesy of G. E. Medical Systems.)

```
g = imcomplement(f)
```

Figure 3.3(c) is the result of using the command

```
>> g2 = imadjust(f, [0.5 0.75], [0 1]);
```

which expands the gray scale interval between 0.5 and 0.75 to the full [0, 1] range. This type of processing is useful for highlighting an intensity band of interest. Finally, using the command

```
>> g3 = imadjust(f, [ ], [ ], 2);
```

produced a result similar to (but with more gray tones than) Fig. 3.3(c) by compressing the low end and expanding the high end of the gray scale [Fig. 3.3(d)].

Sometimes, it is of interest to be able to use function `imadjust` "automatically," without having to be concerned about the low and high parameters discussed above. Function `stretchlim` is useful in that regard; its basic syntax is


```
Low_High = stretchlim(f)
```

where `Low_High` is a two-element vector of a lower and upper limit that can be used to achieve *contrast stretching* (see the following section for a definition of this term). By default, values in `Low_High` specify the intensity levels that saturate the bottom and top 1% of all pixel values in `f`. The result is used in vector `[low_in high_in]` in function `imadjust`, as follows:

```
>> g = imadjust(f, stretchlim(f), [ ]);
```

Figure 3.3(e) shows the result of performing this operation on Fig. 3.3(a). Observe the increase in contrast. Similarly, Fig. 3.3(f) was obtained using the command

```
>> g = imadjust(f, stretchlim(f), [1 0]);
```

As you can see by comparing Figs. 3.3(b) and (f), this operation enhanced the contrast of the negative image. ■

A slightly more general syntax for `stretchlim` is

```
Low_High = stretchlim(f, tol)
```

where `tol` is a two-element vector `[low_frac high_frac]` that specifies the fraction of the image to saturate at low and high pixel values.

If `tol` is a scalar, `low_frac = tol`, and `high_frac = 1 - low_frac`; this saturates equal fractions at low and high pixel values. If you omit it from the argument, `tol` defaults to [0.01 0.99], giving a saturation level of 2%. If you choose `tol = 0`, then `Low_High = [min(f(:)) max(f(:))]`.

3.2.2 Logarithmic and Contrast-Stretching Transformations

Logarithmic and contrast-stretching transformations are basic tools for dynamic range manipulation. Logarithm transformations are implemented using the expression

`log`, `log2`, and `log10` are the base e, base 2, and base 10 logarithms, respectively.

```
g = c*log(1 + f)
```

where `c` is a constant and `f` is floating point. The shape of this transformation is similar to the gamma curve in Fig. 3.2(a) with the low values set at 0 and the

high values set to 1 on both scales. Note, however, that the shape of the gamma curve is variable, whereas the shape of the log function is fixed.

One of the principal uses of the log transformation is to compress dynamic range. For example, it is not unusual to have a Fourier spectrum (Chapter 4) with values in the range $[0, 10^6]$ or higher. When displayed on a monitor that is scaled linearly to 8 bits, the high values dominate the display, resulting in lost visual detail in the lower intensity values in the spectrum. By computing the log, a dynamic range on the order of, for example, 10^6, is reduced to approximately 14 [i.e., $\log_e(10^6) = 13.8$], which is much more manageable.

When performing a logarithmic transformation, it is often desirable to bring the resulting compressed values back to the full range of the display. For 8 bits, the easiest way to do this in MATLAB is with the statement

```
>> gs = im2uint8(mat2gray(g));
```

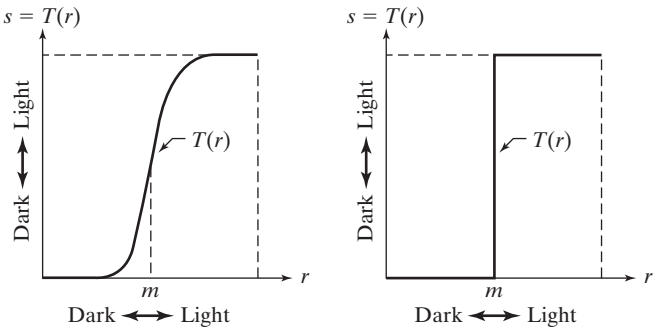
Using `mat2gray` brings the values to the range $[0, 1]$ and using `im2uint8` brings them to the range $[0, 255]$, converting the image to class `uint8`.

The function in Fig. 3.4(a) is called a *contrast-stretching* transformation function because it expands a narrow range of input levels into a wide (stretched) range of output levels. The result is an image of higher contrast. In fact, in the limiting case shown in Fig. 3.4(b), the output is a binary image. This limiting function is called a *thresholding* function, which, as we discuss in Chapter 11, is a simple tool used for image segmentation. Using the notation introduced at the beginning of this section, the function in Fig. 3.4(a) has the form

$$s = T(r) = \frac{1}{1 + (m/r)^E}$$

where r denotes the intensities of the input image, s the corresponding intensity values in the output image, and E controls the slope of the function. This equation is implemented in MATLAB for a floating point image as

```
g = 1./(1 + (m./f).^E)
```

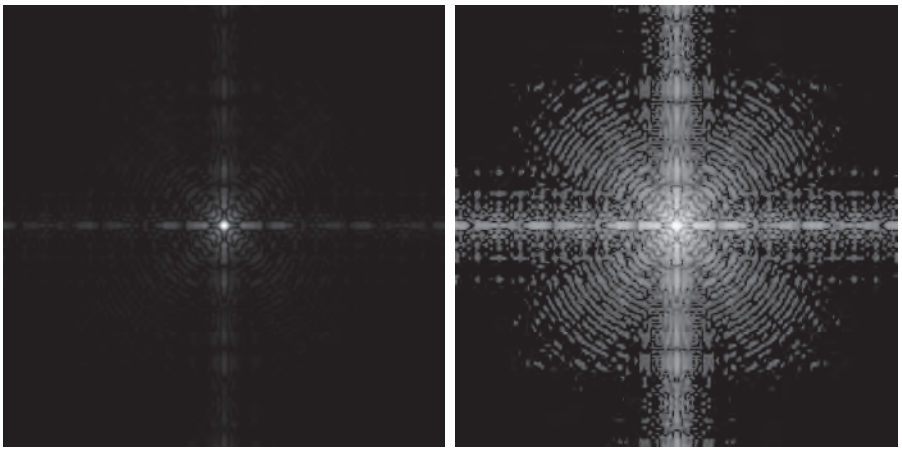


a b

FIGURE 3.4 (a) Contrast-stretching transformation. (b) Thresholding transformation.

a b

FIGURE 3.5
(a) A Fourier spectrum. (b) Result of using a log transformation.



Because the limiting value of `g` is 1, output values cannot exceed the range [0, 1] when working with this type of transformation. The shape in Fig. 3.4(a) was obtained with `E = 20`.

EXAMPLE 3.2: Using a `log` transformation to reduce dynamic range.

■ Figure 3.5(a) is a Fourier spectrum with values in the range 0 to 10^6, displayed on a linearly scaled, 8-bit display system. Figure 3.5(b) shows the result obtained using the commands

```
>> g = im2uint8(mat2gray(log(1 + double(f))));
>> imshow(g)
```

The visual improvement of `g` over the original image is evident. ■

3.2.3 Specifying Arbitrary Intensity Transformations

Suppose that it is necessary to transform the intensities of an image using a specified transformation function. Let `T` denote a column vector containing the values of the transformation function. For example, in the case of an 8-bit image, `T(1)` is the value to which intensity 0 in the input image is mapped, `T(2)` is the value to which 1 is mapped, and so on, with `T(256)` being the value to which intensity 255 is mapped.

Programming is simplified considerably if we express the input and output images in floating point format, with values in the range [0 1]. This means that all elements of column vector `T` must be floating-point numbers in that same range. A simple way to implement intensity mappings is to use function `interp1` which, for this particular application, has the syntax


```
g = interp1(z, T, f)
```

where `f` is the input image, `g` is the output image, `T` is the column vector just explained, and `z` is a column vector of the same length as `T`, formed as follows:

```
z = linspace(0, 1, numel(T))';
```

See Section 2.8.1 regarding function `linspace`.

For a pixel value in `f`, `interp1` first finds that value in the abscissa (`z`). It then finds (interpolates)† the corresponding value in `T` and outputs the interpolated value to `g` in the corresponding pixel location. For example, suppose that `T` is the negative transformation, `T = [1 0]'`. Then, because `T` only has two elements, `z = [0 1]'`. Suppose that a pixel in `f` has the value 0.75. The corresponding pixel in `g` would be assigned the value 0.25. This process is nothing more than the mapping from input to output intensities illustrated in Fig. 3.4(a), but using an arbitrary transformation function $T(r)$. Interpolation is required because we only have a given number of discrete points for T, while r can have any value in the range [0 1].

3.2.4 Some Utility M-Functions for Intensity Transformations

In this section we develop two custom M-functions that incorporate various aspects of the intensity transformations introduced in the previous three sections. We show the details of the code for one of them to illustrate error checking, to introduce ways in which MATLAB functions can be formulated so that they can handle a variable number of inputs and/or outputs, and to show typical code formats used throughout the book. From this point on, detailed code of new M-functions is included in our discussions only when the purpose is to explain specific programming constructs, to illustrate the use of a new MATLAB or Image Processing Toolbox function, or to review concepts introduced earlier. Otherwise, only the syntax of the function is explained, and its code is included in Appendix C. Also, in order to focus on the basic structure of the functions developed in the remainder of the book, this is the last section in which we show extensive use of error checking. The procedures that follow are typical of how error handling is programmed in MATLAB.

Handling a Variable Number of Inputs and/or Outputs

To check the number of arguments input into an M-function we use function `nargin`,

```
n = nargin
```

which returns the actual number of arguments input into the M-function. Similarly, function `nargout` is used in connection with the outputs of an M-function. The syntax is

```
n = nargout
```

†Because `interp1` provides interpolated values at discrete points, this function sometimes is interpreted as performing *lookup table* operations. In fact, MATLAB documentation refers to `interp1` parenthetically as a table lookup function. We use a multidimensional version of this function for just that purpose in `approxfcn`, a custom function developed in Section 3.6.4 for fuzzy image processing.

For example, suppose that we execute the following hypothetical M-function at the prompt:

```
>> T = testhv(4, 5);
```

Use of `nargin` within the body of this function would return a 2, while use of `nargout` would return a 1.

Function `nargchk` can be used in the body of an M-function to check if the correct number of arguments was passed. The syntax is


```
msg = nargchk(low, high, number)
```

This function returns the message `Not enough input arguments` if `number` is less than `low` or `Too many input arguments` if `number` is greater than `high`. If `number` is between `low` and `high` (inclusive), `nargchk` returns an empty matrix. A frequent use of function `nargchk` is to stop execution via the `error` function if the incorrect number of arguments is input. The number of actual input arguments is determined by the `nargin` function. For example, consider the following code fragment:

```
function G = testhv2(x, y, z)
 .
 .
error(nargchk(2, 3, nargin));
 .
 .
```

Typing

```
>> testhv2(6);
```

which only has one input argument would produce the error

```
Not enough input arguments.
```

and execution would terminate.

It is useful to be able to write functions in which the number of input and/or output arguments is variable. For this, we use the variables `varargin` and `varargout`. In the declaration, `varargin` and `varargout` must be lowercase. For example,


```
function [m, n] = testhv3(varargin)
```

accepts a variable number of inputs into function `testhv3.m`, and

```
function [varargout] = testhv4(m, n, p)
```

returns a variable number of outputs from function `testhv4`. If function `testhv3` had, say, one fixed input argument, `x`, followed by a variable number of input arguments, then

```
function [m, n] = testhv3(x, varargin)
```

would cause `varargin` to start with the second input argument supplied by the user when the function is called. Similar comments apply to `varargout`. It is acceptable to have a function in which both the number of input and output arguments is variable.

When `varargin` is used as the input argument of a function, MATLAB sets it to a cell array (see Section 2.10.7) that contains the arguments provided by the user. Because `varargin` is a cell array, an important aspect of this arrangement is that the call to the function can contain a mixed set of inputs. For example, assuming that the code of our hypothetical function `testhv3` is equipped to handle it, a perfectly acceptable syntax having a mixed set of inputs could be

```
>> [m, n] = testhv3(f, [0  0.5  1.5], A, 'label');
```

where `f` is an image, the next argument is a row vector of length 3, `A` is a matrix, and `'label'` is a character string. This is a powerful feature that can be used to simplify the structure of functions requiring a variety of different inputs. Similar comments apply to `varargout`.

Another M-Function for Intensity Transformations

In this section we develop a function that computes the following transformation functions: negative, log, gamma and contrast stretching. These transformations were selected because we will need them later, and also to illustrate the mechanics involved in writing an M-function for intensity transformations. In writing this function we use function `tofloat`,

```
[g, revertclass] = tofloat(f)
```

introduced in Section 2.7. Recall from that discussion that this function converts an image of class `logical`, `uint8`, `uint16`, or `int16` to class `single`, applying the appropriate scale factor. If `f` is of class `double` or `single`, then `g = f`; also, recall that `revertclass` is a function handle that can be used to covert the output back to the same class as `f`.

Note in the following M-function, which we call `intrans`, how function options are formatted in the Help section of the code, how a variable number of inputs is handled, how error checking is interleaved in the code, and how the class of the output image is matched to the class of the input. Keep in mind when studying the following code that `varargin` is a cell array, so its elements are selected by using curly braces.

```
function g = intrans(f, method, varargin)                    intrans
%INTRANS Performs intensity (gray-level) transformations.
%   G = INTRANS(F, 'neg') computes the negative of input image F.
%
%   G = INTRANS(F, 'log', C, CLASS) computes C*log(1 + F) and
```

```
%   multiplies the result by (positive) constant C. If the last two
%   parameters are omitted, C defaults to 1. Because the log is used
%   frequently to display Fourier spectra, parameter CLASS offers
%   the option to specify the class of the output as 'uint8' or
%   'uint16'. If parameter CLASS is omitted, the output is of the
%   same class as the input.
%
%   G = INTRANS(F, 'gamma', GAM) performs a gamma transformation on
%   the input image using parameter GAM (a required input).
%
%   G = INTRANS(F, 'stretch', M, E) computes a contrast-stretching
%   transformation using the expression 1./(1 + (M./F).^E).
%   Parameter M must be in the range [0, 1]. The default value for
%   M is mean2(tofloat(F)), and the default value for E is 4.
%
%   G = INTRANS(F, 'specified', TXFUN) performs the intensity
%   transformation s = TXFUN(r) where r are input intensities, s are
%   output intensities, and TXFUN is an intensity transformation
%   (mapping) function, expressed as a vector with values in the
%   range [0, 1]. TXFUN must have at least two values.
%
%   For the 'neg', 'gamma', 'stretch' and 'specified'
%   transformations, floating-point input images whose values are
%   outside the range [0, 1] are scaled first using MAT2GRAY. Other
%   images are converted to floating point using TOFLOAT. For the
%   'log' transformation,floating-point images are transformed
%   without being scaled; other images are converted to floating
%   point first using TOFLOAT.
%
%   The output is of the same class as the input, except if a
%   different class is specified for the 'log' option.

% Verify the correct number of inputs.
error(nargchk(2, 4, nargin))

if strcmp(method, 'log')
   % The log transform handles image classes differently than the
   % other transforms, so let the logTransform function handle that
   % and then return.
   g = logTransform(f, varargin{:});
   return;
end

% If f is floating point, check to see if it is in the range [0 1].
% If it is not, force it to be using function mat2gray.
if isfloat(f) && (max(f(:)) > 1 || min(f(:)) < 0)
   f = mat2gray(f);
end
[f, revertclass] = tofloat(f); %Store class of f for use later.

% Perform the intensity transformation specified.
```

```
switch method
case 'neg'
   g = imcomplement(f);

case 'gamma'
   g = gammaTransform(f, varargin{:});

case 'stretch'
   g = stretchTransform(f, varargin{:});

case 'specified'
   g = spcfiedTransform(f, varargin{:});

otherwise
   error('Unknown enhancement method.')
end

% Convert to the class of the input image.
g = revertclass(g);

%-----------------------------------------------------------------%
function g = gammaTransform(f, gamma)
g = imadjust(f, [ ], [ ], gamma);

%-----------------------------------------------------------------%
function g = stretchTransform(f, varargin)
if isempty(varargin)
   % Use defaults.
   m = mean2(f);
   E = 4.0;
elseif length(varargin) == 2
   m = varargin{1};
   E = varargin{2};
else
   error('Incorrect number of inputs for the stretch method.')
end
g = 1./(1 + (m./f).^E);

%-----------------------------------------------------------------%
function g = spcfiedTransform(f, txfun)
% f is floating point with values in the range [0 1].
txfun = txfun(:); % Force it to be a column vector.
if any(txfun) > 1 || any(txfun) <= 0
   error('All elements of txfun must be in the range [0 1].')
end
T = txfun;
X = linspace(0, 1, numel(T))';
g = interp1(X, T, f);

%-----------------------------------------------------------------%
function g = logTransform(f, varargin)
```

```
[f, revertclass] = tofloat(f);
if numel(varargin) >= 2
   if strcmp(varargin{2}, 'uint8')
      revertclass = @im2uint8;
   elseif strcmp(varargin{2}, 'uint16')
      revertclass = @im2uint16;
   else
      error('Unsupported CLASS option for ''log'' method.')
   end
end
if numel(varargin) < 1
   % Set default for C.
   C = 1;
else
   C = varargin{1};
end
g = C * (log(1 + f));
g = revertclass(g);
```

EXAMPLE 3.3: Illustration of function `intrans`.

■ As an illustration of function `intrans`, consider the image in Fig. 3.6(a), which is an ideal candidate for contrast stretching to enhance the skeletal structure. The result in Fig. 3.6(b) was obtained with the following call to `intrans`:

```
>> g = intrans(f, 'stretch', mean2(tofloat(f)), 0.9);
>> figure, imshow(g)
```

Note how function `mean2` was used to compute the mean value of `f` directly inside the function call. The resulting value was used for `m`. Image `f` was converted to floating point using `tofloat` in order to scale its values to the range [0, 1] so that the mean would also be in this range, as required for input `m`. The value of `E` was determined interactively. ■

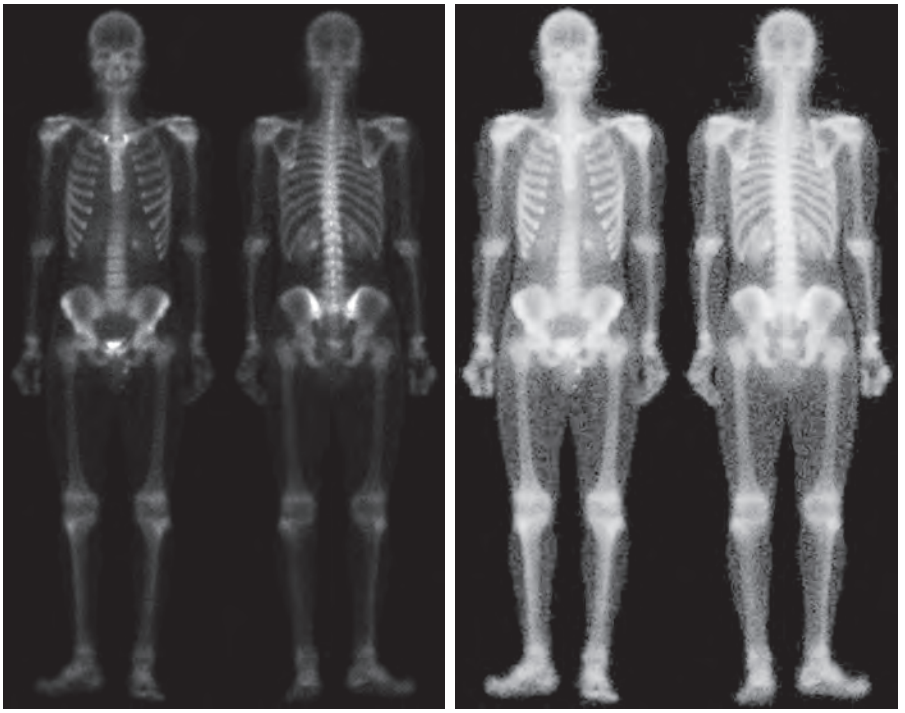
An M-Function for Intensity Scaling

When working with images, computations that result in pixel values that span a wide negative to positive range are common. While this presents no problems during intermediate computations, it does become an issue when we want to use an 8-bit or 16-bit format for saving or viewing an image, in which case it usually is desirable to scale the image to the full, maximum range, [0, 255] or [0, 65535]. The following custom M-function, which we call `gscale`, accomplishes this. In addition, the function can map the output levels to a specified range. The code for this function does not include any new concepts so we do not include it here. See Appendix C for the listing.

The syntax of function `gscale` is

`gscale`

```
g = gscale(f, method, low, high)
```



a b

FIGURE 3.6
(a) Bone scan image. (b) Image enhanced using a contrast-stretching transformation. (Original image courtesy of G. E. Medical Systems.)

where `f` is the image to be scaled. Valid values for `method` are `'full8'` (the default), which scales the output to the full range [0, 255], and `'full16'`, which scales the output to the full range [0, 65535]. If included, parameters `low` and `high` are ignored in these two conversions. A third valid value of `method` is `'minmax'`, in which case parameters `low` and `high`, both in the range [0, 1], must be provided. If `'minmax'` is selected, the levels are mapped to the range `[low, high]`. Although these values are specified in the range [0, 1], the program performs the proper scaling, depending on the class of the input, and then converts the output to the same class as the input. For example, if `f` is of class `uint8` and we specify `'minmax'` with the range [0, 0.5], the output also will be of class `uint8`, with values in the range [0, 128]. If `f` is floating point and its range of values is outside the range [0, 1], the program converts it to this range before proceeding. Function `gscale` is used in numerous places throughout the book.

3.3 Histogram Processing and Function Plotting

Intensity transformation functions based on information extracted from image intensity histograms play a central role in image processing, in areas such as enhancement, compression, segmentation, and description. The focus of this section is on obtaining, plotting, and using histograms for image enhancement. Other applications of histograms are discussed in later chapters.

See Section 4.5.3 for a discussion of 2-D plotting techniques.

3.3.1 Generating and Plotting Image Histograms

The histogram of a digital image with L total possible intensity levels in the range $[0, G]$ is defined as the discrete function

$$h(r_k) = n_k$$

where r_k is the kth intensity level in the interval $[0, G]$ and n_k is the number of pixels in the image whose intensity level is r_k. The value of G is 255 for images of class `uint8`, 65535 for images of class `uint16`, and 1.0 for floating point images. Note that $G = L - 1$ for images of class `uint8` and `uint16`.

Sometimes it is necessary to work with *normalized* histograms, obtained simply by dividing all elements of $h(r_k)$ by the total number of pixels in the image, which we denote by n:

$$p(r_k) = \frac{h(r_k)}{n} = \frac{n_k}{n}$$

where, for integer images, $k = 0, 1, 2, \ldots, L - 1$. From basic probability, we recognize $p(r_k)$ as an estimate of the probability of occurrence of intensity level r_k.

The core function in the toolbox for dealing with image histograms is `imhist`, with the basic syntax:


```
h = imhist(f, b)
```

where `f` is the input image, `h` is its histogram, and `b` is the number of bins used in forming the histogram (if `b` is not included in the argument, `b = 256` is used by default). A bin is simply a subdivision of the intensity scale. For example, if we are working with `uint8` images and we let `b = 2`, then the intensity scale is subdivided into two ranges: 0 to 127 and 128 to 255. The resulting histogram will have two values: `h(1)`, equal to the number of pixels in the image with values in the interval [0, 127] and `h(2)`, equal to the number of pixels with values in the interval [128, 255]. We obtain the normalized histogram by using the expression

```
p = imhist(f, b)/numel(f)
```

Recall from Section 2.10.3 that function `numel(f)` gives the number of elements in array `f` (i.e., the number of pixels in the image).

EXAMPLE 3.4: Computing and plotting image histograms.

■ Consider the image, `f`, from Fig. 3.3(a). The simplest way to plot its histogram on the screen is to use `imhist` with no output specified:

```
>> imhist(f);
```

Figure 3.7(a) shows the result. This is the histogram display default in the toolbox. However, there are many other ways to plot a histogram, and we take this opportunity to explain some of the plotting options in MATLAB that are representative of those used in image processing applications.

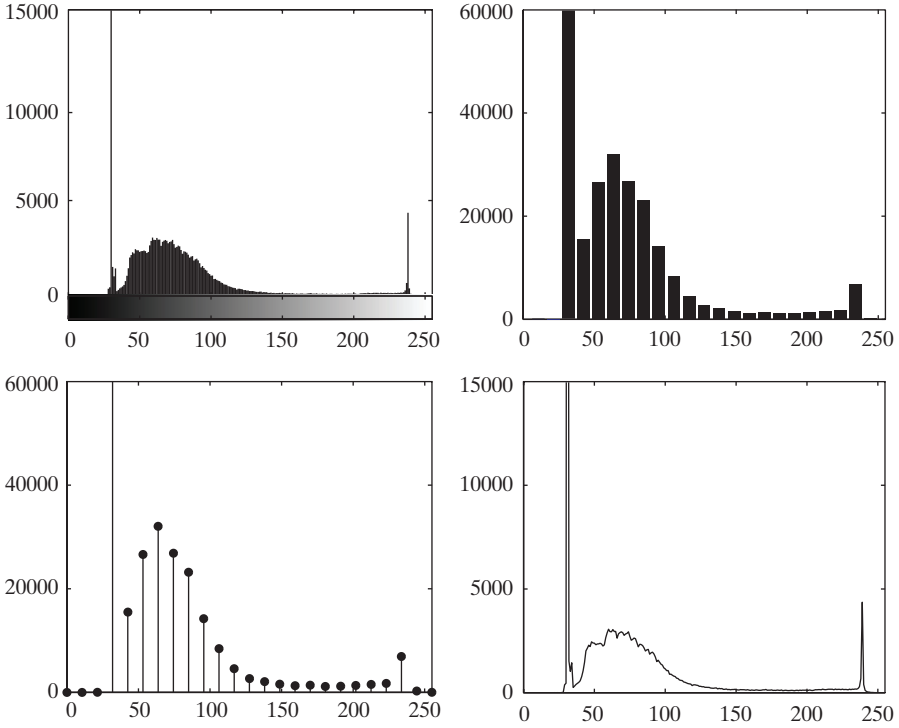
Histograms can be plotted also using *bar* graphs. For this purpose we can use the function

```
bar(horz, z, width)
```

where `z` is a row vector containing the points to be plotted, `horz` is a vector of the same dimension as `z` that contains the increments of the horizontal scale, and `width` is a number between 0 and 1. In other words, the values of `horz` give the horizontal increments and the values of `z` are the corresponding vertical values. If `horz` is omitted, the horizontal axis is divided in units from 0 to `length(z)`. When `width` is 1, the bars touch; when it is 0, the bars are vertical lines. The default value is 0.8. When plotting a bar graph, it is customary to reduce the resolution of the horizontal axis by dividing it into bands.

The following commands produce a bar graph, with the horizontal axis divided into groups of approximately 10 levels:

```
>> h = imhist(f, 25);
>> horz = linspace(0, 255, 25);
```



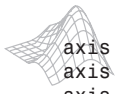
a b
c d

FIGURE 3.7 Various ways to plot an image histogram. (a) `imhist`, (b) `bar`, (c) `stem`, (d) `plot`.

```
>> bar(horz, h)
>> axis([0 255 0 60000])
>> set(gca, 'xtick', 0:50:255)
>> set(gca, 'ytick', 0:20000:60000)
```

Figure 3.7(b) shows the result. The narrow peak located at the high end of the intensity scale in Fig. 3.7(a) is lower in the bar graph because larger horizontal increments were used in that graph. The vertical scale spans a wider range of values than for the full histogram in Fig. 3.7(a) because the height of each bar is determined by all pixels in a range, rather than by all pixels with a single value.

The fourth statement in the preceding code was used to expand the lower range of the vertical axis for visual analysis, and to set the horizontal axis to the same range as in Fig. 3.7. One of the `axis` function syntax forms is



`axis ij` places the origin of the axis system on the top left. This is the default is when superimposing axes on images. As we show in Example 5.12, sometimes it is useful to have the origin on the bottom left. Using `axis xy` does that.

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

which sets the minimum and maximum values in the horizontal and vertical axes. In the last two statements, `gca` means "get current axis" (i.e., the axes of the figure last displayed), and `xtick` and `ytick` set the horizontal and vertical axes ticks in the intervals shown. Another syntax used frequently is

```
axis tight
```

which sets the axis limits to the range of the data.

Axis labels can be added to the horizontal and vertical axes of a graph using the functions

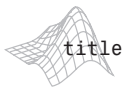

```
xlabel('text string', 'fontsize', size)
ylabel('text string', 'fontsize', size)
```

where `size` is the font size in points. Text can be added to the body of the figure by using function `text`, as follows:


```
text(xloc, yloc, 'text string', 'fontsize', size)
```

where `xloc` and `yloc` define the location where text starts. Use of these three functions is illustrated in Example 3.4. It is important to note that functions that set axis values and labels are used *after* the function has been plotted.

A title can be added to a plot using function `title`, whose basic syntax is




```
title('titlestring')
```

where `titlestring` is the string of characters that will appear on the title, centered above the plot.

A *stem* graph is similar to a bar graph. The syntax is


```
stem(horz, z, 'LineSpec', 'fill')
```

where `z` is row vector containing the points to be plotted, and `horz` is as

described for function `bar`. If `horz` is omitted, the horizontal axis is divided in units from 0 to `length(z)`, as before.

The argument,

```
LineSpec
```

is a triplet of values from Table 3.1. For example, `stem(horz, h, 'r--p')` produces a stem plot where the lines and markers are red, the lines are dashed, and the markers are five-point stars. If `fill` is used, the marker is filled with the color specified in the first element of the triplet. The default color is `blue`, the line default is `solid`, and the default marker is a `circle`. The stem graph in Fig. 3.7(c) was obtained using the statements

```
>> h = imhist(f, 25);
>> horz = linspace(0, 255, 25);
>> stem(horz, h, 'fill')
>> axis([0 255 0 60000])
>> set(gca, 'xtick', [0:50:255])
>> set(gca, 'ytick', [0:20000:60000])
```

Next, we consider function `plot`, which plots a set of points by linking them with straight lines. The syntax is

Color Specifiers		Line Specifiers		Marker Specifiers	
Symbol	**Color**	**Symbol**	**Line Style**	**Symbol**	**Marker**
k	Black	-	Solid	+	Plus sign
w	White	--	Dashed	o	Circle
r	Red	:	Dotted	*	Asterisk
g	Green	-.	Dash-dot	.	Point
b	Blue			x	Cross
c	Cyan			s	Square
y	Yellow			d	Diamond
m	Magenta			^	Upward-pointing triangle
				v	Downward-pointing triangle
				>	Right-pointing triangle
				<	Left-pointing triangle
				p	Pentagram (five-point star)
				h	Hexagram (six-point star)

TABLE 3.1 Color, line, and marker specifiers for use in functions `stem` and `plot`.

See the `plot` help page for additional options available for this function.

```
plot(horz, z, 'LineSpec')
```

where the arguments are as defined previously for stem plots. As in `stem`, the attributes in `plot` are specified as a triplet. The defaults for `plot` are solid blue lines with no markers. If a triplet is specified in which the middle value is blank (or omitted), no lines are plotted. As before, if `horz` is omitted, the horizontal axis is divided in units from 0 to `length(z)`.

The plot in Fig. 3.7(d) was obtained using the following statements:

```
>> hc = imhist(f);
>> plot(hc) % Use the default values.
>> axis([0 255 0 15000])
>> set(gca, 'xtick', [0:50:255])
>> set(gca, 'ytick', [0:2000:15000])
```

Function `plot` is used frequently to display transformation functions (see Example 3.5). ■

Plot defaults are useful for superimposing markers on an image. For example, to place green asterisks at points given in vectors `x` and `y` in an image, `f`, we use:

```
>> imshow(f)
>> hold on
>> plot(y(:),x(:),'g*')
```

where the order of `y(:)` and `x(:)` is reversed to compensate for the fact that image and plot coordinate systems are different in MATLAB. Command `hold on` is explained below.

In the preceding discussion axis limits and tick marks were set manually. To set the limits and ticks automatically, use functions `ylim` and `xlim`, which, for our purposes here, have the syntax forms


```
ylim('auto')
xlim('auto')
```

Among other possible variations of the syntax for these two functions (see the help documentation for details), there is a manual option, given by

```
ylim([ymin  ymax])
xlim([xmin  xmax])
```

which allows manual specification of the limits. If the limits are specified for only one axis, the limits on the other axis are set to `'auto'` by default. We use these functions in the following section. Typing `hold on` at the prompt retains the current plot and certain axes properties so that subsequent graphing commands add to the existing graph.

Another plotting function that is particularly useful when dealing with function handles (see Sections 2.10.4 and 2.10.5) is function `fplot`. The basic syntax is


```
fplot(fhandle, limits, 'LineSpec')
```

See the help page for `fplot` for a discussion of additional syntax forms.

where `fhandle` is a function handle, and `limits` is a vector specifying the x-axis limits, `[xmin  xmax]`. You will recall from the discussion of function `timeit` in Section 2.10.5 that using function handles allows the syntax of the underlying function to be independent of the parameters of the function to be processed (plotted in this case). For example, to plot the hyperbolic tangent function, `tanh`, in the range `[-2 2]` using a dotted line we write

```
>> fhandle = @tanh;
>> fplot(fhandle, [-2 2], ':')
```

Function `fplot` uses an automatic, adaptive increment control scheme to produce a representative graph, concentrating more detail where the rate of change is the greatest. Thus, only the plotting limits have to be specified by the user. While this simplifies plotting tasks, the automatic feature can at times yield unexpected results. For example, if a function is initially 0 for an appreciable interval, it is possible for `fplot` to assume that the function is zero and just plot 0 for the entire interval. In cases such as this, you can specify a minimum number of points for the function to plot. The syntax is

```
fplot(fhandle, limits, 'LineSpec', n)
```

Specifying `n >= 1` forces `fplot` to plot the function with a minimum of `n + 1` points, using a step size of `(1/n)*(upper_lim - lower_lim)`, where upper and lower refer to the upper and lower limits specified in `limits`.

3.3.2 Histogram Equalization

Assume for a moment that intensity levels are continuous quantities normalized to the range [0, 1], and let $p_r(r)$ denote the probability density function (PDF) of the intensity levels in a given image, where the subscript is used for differentiating between the PDFs of the input and output images. Suppose that we perform the following transformation on the input levels to obtain output (processed) intensity levels, s,

$$s = T(r) = \int_0^r p_r(w)\,dw$$

where w is a dummy variable of integration. It can be shown (Gonzalez and Woods [2008]) that the probability density function of the output levels is *uniform*; that is,

$$p_s(s) = \begin{cases} 1 & \text{for } 0 \le s \le 1 \\ 0 & \text{otherwise} \end{cases}$$

In other words, the preceding transformation generates an image whose intensity levels are equally likely, and, in addition, cover the entire range [0, 1]. The net result of this intensity-level *equalization* process is an image with increased dynamic range, which will tend to have higher contrast. Note that the transformation function is really nothing more than the cumulative distribution function (CDF).

When dealing with discrete quantities we work with histograms and call the preceding technique *histogram equalization*, although, in general, the histogram of the processed image will not be uniform, due to the discrete nature of the variables. With reference to the discussion in Section 3.3.1, let $p_r(r_j)$ for $j = 0, 1, 2, \ldots, L - 1$, denote the histogram associated with the intensity levels

of a given image, and recall that the values in a normalized histogram are approximations to the probability of occurrence of each intensity level in the image. For discrete quantities we work with summations, and the equalization transformation becomes

$$\begin{aligned} s_k &= T(r_k) \\ &= \sum_{j=0}^{k} p_r(r_j) \\ &= \sum_{j=0}^{k} \frac{n_j}{n} \end{aligned}$$

for $k = 0, 1, 2, \ldots, L-1$, where s_k is the intensity value in the output (processed) image corresponding to value r_k in the input image.

Histogram equalization is implemented in the toolbox by function `histeq`, which has the syntax


```
g = histeq(f, nlev)
```

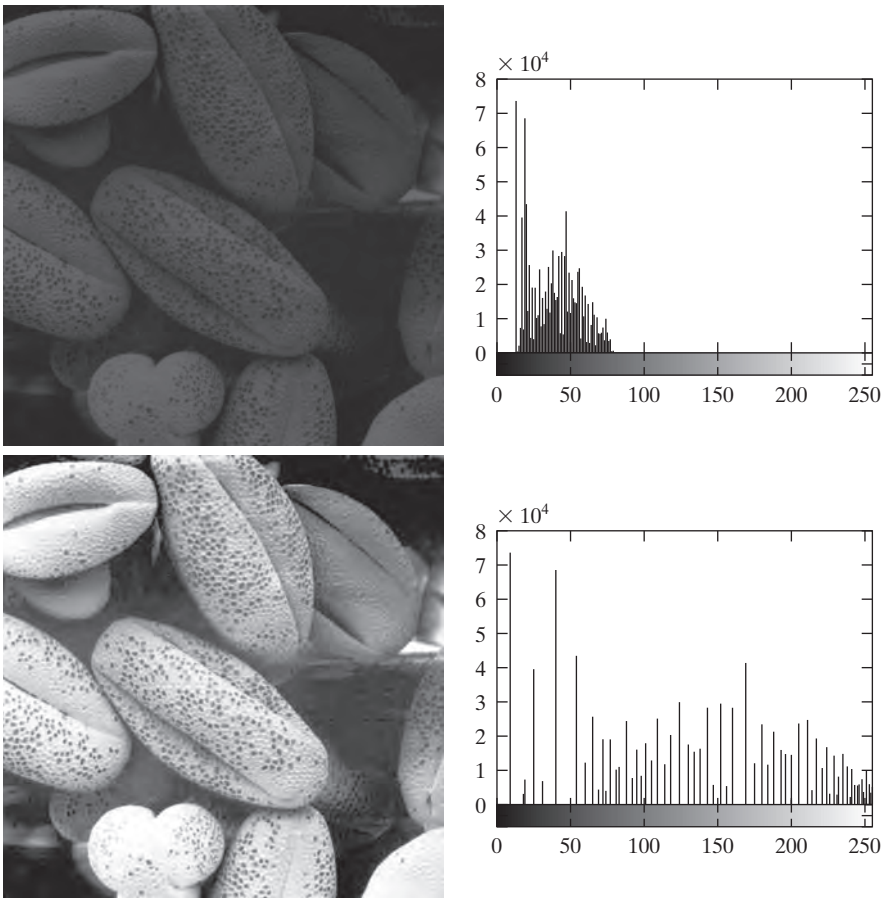
where `f` is the input image and `nlev` is the number of intensity levels specified for the output image. If `nlev` is equal to L (the total number of *possible* levels in the input image), then `histeq` implements the transformation function directly. If `nlev` is less than L, then `histeq` attempts to distribute the levels so that they will approximate a flat histogram. Unlike `imhist`, the default value in `histeq` is `nlev = 64`. For the most part, we use the maximum possible number of levels (generally 256) for `nlev` because this produces a true implementation of the histogram-equalization method just described.

EXAMPLE 3.5: Histogram equalization.

■ Figure 3.8(a) is an electron microscope image of pollen, magnified approximately 700 times. In terms of needed enhancement, the most important features of this image are that it is dark and has a low dynamic range. These characteristics are evident in the histogram in Fig. 3.8(b), in which the dark nature of the image causes the histogram to be biased toward the dark end of the gray scale. The low dynamic range is evident from the fact that the histogram is narrow with respect to the entire gray scale. Letting `f` denote the input image, the following sequence of steps produced Figs. 3.8(a) through (d):

```
>> imshow(f); % Fig. 3.8(a).
>> figure, imhist(f) % Fig. 3.8(b).
>> ylim('auto')
>> g = histeq(f, 256);
>> figure, imshow(g) % Fig. 3.8(c).
>> figure, imhist(g) % Fig. 3.8(d).
>> ylim('auto')
```

The image in Fig. 3.8(c) is the histogram-equalized result. The improvements in average intensity and contrast are evident. These features also are



a b
c d

FIGURE 3.8 Illustration of histogram equalization. (a) Input image, and (b) its histogram. (c) Histogram-equalized image, and (d) its histogram. The improvement between (a) and (c) is evident. (Original image courtesy of Dr. Roger Heady, Research School of Biological Sciences, Australian National University, Canberra.)

evident in the histogram of this image, shown in Fig. 3.8(d). The increase in contrast is due to the considerable spread of the histogram over the entire intensity scale. The increase in overall intensity is due to the fact that the average intensity level in the histogram of the equalized image is higher (lighter) than the original. Although the histogram-equalization method just discussed does not produce a flat histogram, it has the desired characteristic of being able to increase the dynamic range of the intensity levels in an image.

As noted earlier, the transformation function used in histogram equalization is the cumulative sum of normalized histogram values. We can use function `cumsum` to obtain the transformation function, as follows:

If `A` is a vector, `B = cumsum(A)` gives the sum of its elements. If `A` is a higher-dimensional array, then `B = cumsum(A, dim)` gives the sum along the dimension specified by `dim`.


```
>> hnorm = imhist(f)./numel(f); % Normalized histogram.
>> cdf = cumsum(hnorm); % CDF.
```

A plot of `cdf`, shown in Fig. 3.9, was obtained using the following commands:

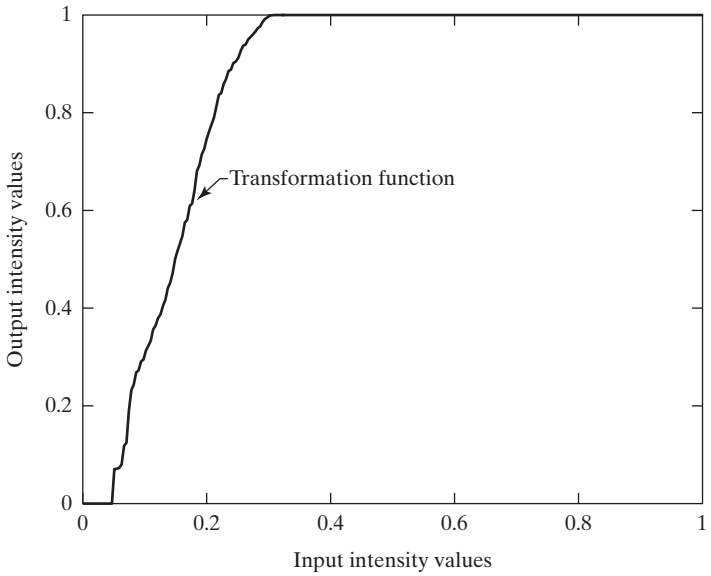


FIGURE 3.9 Transformation function used to map the intensity values from the input image in Fig. 3.7(a) to the values of the output image in Fig. 3.7(c).

```
>> x = linspace(0, 1, 256);   % Intervals for [0,1] horiz
                              % scale.
>> plot(x, cdf)               % Plot cdf vs. x.
>> axis([0 1 0 1]);           % Scale, settings, and labels:
>> set(gca, 'xtick', 0:.2:1)
>> set(gca, 'ytick', 0:.2:1)
>> xlabel('Input intensity values', 'fontsize', 9)
>> ylabel('Output intensity values', 'fontsize', 9)
```

See the help page for this function for details on how to use it.

The text in the body of the graph was inserted using the `TextBox` and `Arrow` commands from the **Insert** menu in the MATLAB figure window containing the plot. You can use function `annotation` to write code that inserts items such as text boxes and arrows on graphs, but the **Insert** menu is considerably easier to use.

You can see by looking at the histograms in Fig. 3.8 that the transformation function in Fig. 3.9 maps a narrow range of intensity levels in the lower end of the input intensity scale to the full intensity range in the output image. The improvement in image contrast is evident by comparing the input and output images in Fig. 3.8. ■

3.3.3 Histogram Matching (Specification)

Histogram equalization produces a transformation function that is adaptive, in the sense that it is based on the histogram of a given image. However, once the transformation function for an image has been computed, it does not change

unless the histogram of the image changes. As noted in the previous section, histogram equalization achieves enhancement by spreading the levels of the input image over a wider range of the intensity scale. We show in this section that this does not always lead to a successful result. In particular, it is useful in some applications to be able to specify the shape of the histogram that we wish the processed image to have. The method used to generate an image that has a specified histogram is called *histogram matching* or *histogram specification*.

The method is simple in principle. Consider for a moment continuous levels that are normalized to the interval [0, 1], and let r and z denote the intensity levels of the input and output images. The input levels have probability density function $p_r(r)$ and the output levels have the *specified* probability density function $p_z(z)$. We know from the discussion in the previous section that he transformation

$$s = T(r) = \int_0^r p_r(w)\,dw$$

results in intensity levels, s, with a uniform probability density function $p_s(s)$. Suppose now that we define a variable z with the property

$$H(z) = \int_0^z p_z(w)\,dw = s$$

Keep in mind that we are after an image with intensity levels, z, that have the specified density $p_z(z)$. From the preceding two equations, it follows that

$$z = H^{-1}(s) = H^{-1}\left[T(r)\right]$$

We can find $T(r)$ from the input image (this is the histogram-equalization transformation discussed in the previous section), so it follows that we can use the preceding equation to find the transformed levels z whose density is the specified $p_z(z)$ provided that we can find H^{-1}. When working with discrete variables, we can guarantee that the inverse of H exists if $p(z_k)$ is a valid histogram (i.e., it has unit area and all its values are nonnegative), *and* none of its components is zero [i.e., no bin of $p(z_k)$ is empty]. As in histogram equalization, the discrete implementation of the preceding method only yields an approximation to the specified histogram.

The toolbox implements histogram matching using the following syntax in `histeq`:

```
g = histeq(f, hspec)
```

where `f` is the input image, `hspec` is the specified histogram (a row vector of specified values), and `g` is the output image, whose histogram approximates the specified histogram, `hspec`. This vector should contain integer counts corresponding to equally spaced bins. A property of `histeq` is that the histogram of `g` generally better matches `hspec` when `length(hspec)` is much smaller than the number of intensity levels in `f`.

EXAMPLE 3.6: Histogram matching.

■ Figure 3.10(a) shows an image, `f`, of the Mars moon, Phobos, and Fig. 3.10(b) shows its histogram, obtained using `imhist(f)`. The image is dominated by large, dark areas, resulting in a histogram characterized by a large concentration of pixels in the dark end of the gray scale. At first glance, one might conclude that histogram equalization would be a good approach to enhance this image, so that details in the dark areas become more visible. However, the result in Fig. 3.10(c), obtained using the command

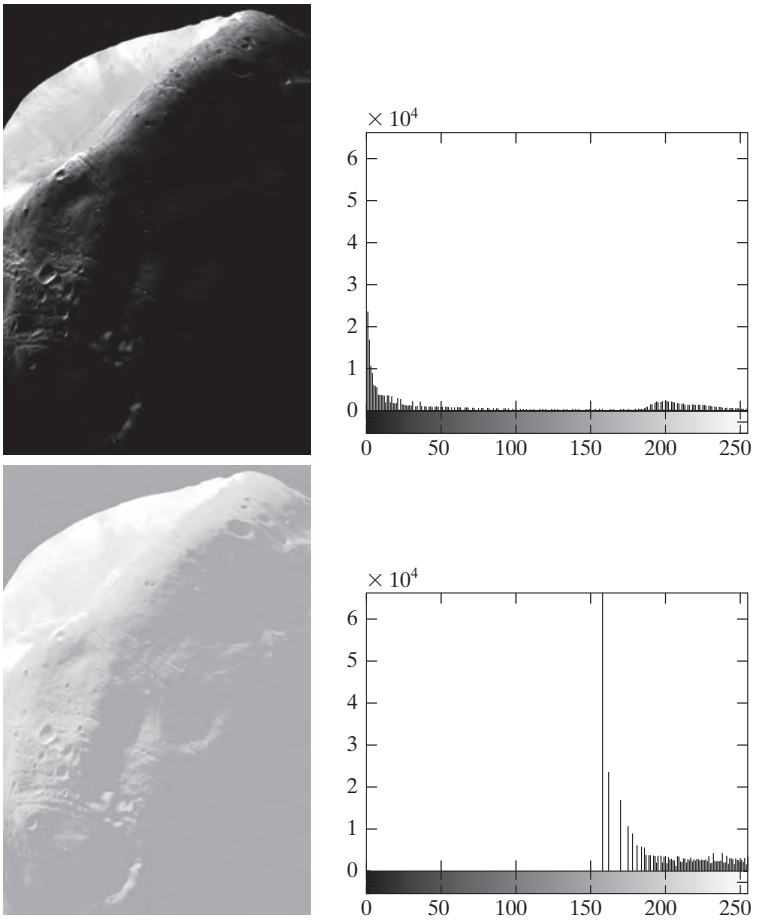
```
>> f1 = histeq(f, 256);
```

shows that histogram equalization in fact produced an image with a "washed-out" appearance—not a particularly good result in this case. The reason for this can be seen by studying the histogram of the equalized image, shown in Fig. 3.10(d). Here, we see that the intensity levels have been shifted to the upper one-half of the gray scale, thus giving the image the low-contrast, washed-out appearance mentioned above. The cause of the shift is the large concentration of dark components at or near 0 in the original histogram. The cumulative transformation function obtained from this histogram is steep, thus mapping the large concentration of pixels in the low end of the gray scale to the high end of the scale.

One possibility for remedying this situation is to use histogram matching, with the desired histogram having a lesser concentration of components in the low end of the gray scale, and maintaining the general shape of the histogram of the original image. We note from Fig. 3.10(b) that the histogram is basically bimodal, with one large mode at the origin, and another, smaller, mode at the high end of the gray scale. These types of histograms can be modeled, for example, by using multimodal Gaussian functions. The following M-function computes a bimodal Gaussian function normalized to unit area, so it can be used as a specified histogram.

`twomodegauss`

```
function p = twomodegauss(m1, sig1, m2, sig2, A1, A2, k)
%TWOMODEGAUSS Generates a two-mode Gaussian function.
%   P = TWOMODEGAUSS(M1, SIG1, M2, SIG2, A1, A2, K) generates a
%   two-mode, Gaussian-like function in the interval [0, 1]. P is a
%   256-element vector normalized so that SUM(P) = 1. The mean and
%   standard deviation of the modes are (M1, SIG1) and (M2, SIG2),
%   respectively. A1 and A2 are the amplitude values of the two
%   modes. Since the output is normalized, only the relative values
%   of A1 and A2 are important. K is an offset value that raises the
%   "floor" of the function. A good set of values to try is M1 =
%   0.15, SIG1 = 0.05, M2 = 0.75, SIG2 = 0.05, A1 = 1, A2 = 0.07,
%   and K = 0.002.

c1 = A1 * (1 / ((2 * pi) ^ 0.5) * sig1);
k1 = 2 * (sig1 ^ 2);
c2 = A2 * (1 / ((2 * pi) ^ 0.5) * sig2);
k2 = 2 * (sig2 ^ 2);
z  = linspace(0, 1, 256);
```



a b
c d

FIGURE 3.10
(a) Image of the Mars moon Phobos. (b) Histogram. (c) Histogram-equalized image. (d) Histogram of (c). (Original image courtesy of NASA.)

```
p = k + c1 * exp(-((z - m1) .^ 2) ./ k1) + ...
    c2 * exp(-((z - m2) .^ 2) ./ k2);
p = p ./ sum(p(:));
```

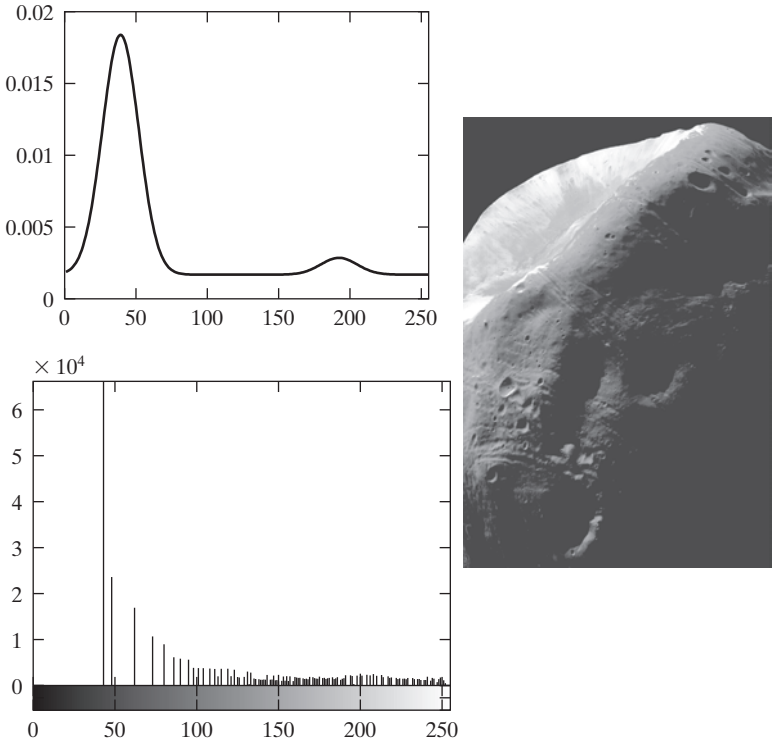
The following interactive function accepts inputs from a keyboard and plots the resulting Gaussian function. Refer to Section 2.10.6 for an explanation of function `input`. Note how the limits of the plots are set.

`manualhist`

```
function p = manualhist
%MANUALHIST Generates a two-mode histogram interactively.
%   P = MANUALHIST generates a two-mode histogram using function
%   TWOMODEGAUSS(m1, sig1, m2, sig2, A1, A2, k). m1 and m2 are the
%   means of the two modes and must be in the range [0,1]. SIG1 and
%   SIG2 are the standard deviations of the two modes. A1 and A2 are
%   amplitude values, and k is an offset value that raises the floor
```

```
%   of the the histogram. The number of elements in the histogram
%   vector P is 256 and sum(P) is normalized to 1. MANUALHIST
%   repeatedly prompts for the parameters and plots the resulting
%   histogram until the user types an 'x' to quit, and then it
%   returns the last histogram computed.
%
%   A good set of starting values is: (0.15, 0.05, 0.75, 0.05, 1,
%   0.07, 0.002).

% Initialize.
repeats = true;
quitnow = 'x';

% Compute a default histogram in case the user quits before
% estimating at least one histogram.
p = twomodegauss(0.15, 0.05, 0.75, 0.05, 1, 0.07, 0.002);

% Cycle until an x is input.
while repeats
   s = input('Enter m1, sig1, m2, sig2, A1, A2, k OR x to quit:',...
      's');
   if strcmp(s, quitnow)
      break
   end

   % Convert the input string to a vector of numerical values and
   % verify the number of inputs.
   v = str2num(s);
   if numel(v) ~= 7
      disp('Incorrect number of inputs.')
      continue
   end

   p = twomodegauss(v(1), v(2), v(3), v(4), v(5), v(6), v(7));
   % Start a new figure and scale the axes. Specifying only xlim
   % leaves ylim on auto.
   figure, plot(p)
   xlim([0 255])
end
```

Because the problem with histogram equalization in this example is due primarily to a large concentration of pixels in the original image with levels near 0, a reasonable approach is to modify the histogram of that image so that it does not have this property. Figure 3.11(a) shows a plot of a function (obtained with program `manualhist`) that preserves the general shape of the original histogram, but has a smoother transition of levels in the dark region of the intensity scale. The output of the program, `p`, consists of 256 equally spaced points from this function and is the desired specified histogram. An image with the specified histogram was generated using the command



a
c b

FIGURE 3.11
(a) Specified histogram.
(b) Result of enhancement by histogram matching.
(c) Histogram of (b).

```
>> g = histeq(f, p);
```

Figure 3.11(b) shows the result. The improvement over the histogram-equalized result in Fig. 3.10(c) is evident. Note that the specified histogram represents a rather modest change from the original histogram. This is all that was required to obtain a significant improvement in enhancement. The histogram of Fig. 3.11(b) is shown in Fig. 3.11(c). The most distinguishing feature of this histogram is how its low end has been moved closer to a lighter region of the gray scale, and thus closer to the specified shape. Note, however, that the shift to the right was not as extreme as the shift in the histogram in Fig. 3.10(d), which corresponds to the poorly enhanced image of Fig. 3.10(c). ■

3.3.4 Function `adapthisteq`

This toolbox function performs so-called *contrast-limited adaptive histogram equalization* (CLAHE). Unlike the methods discussed in the previous two sections, which operate on an entire image, this approach consists of processing small regions of the image (called *tiles)* using histogram specification for each tile individually. Neighboring tiles are then combined using bilinear interpolation to eliminate artificially induced boundaries. The contrast, especially in

See Section 6.6 regarding interpolation.

areas of homogeneous intensity, can be limited to avoid amplifying noise. The syntax for `adapthisteq` is


```
g = adapthisteq(f, param1, val1, param2, val2, ...)
```

where `f` is the input image, `g` is the output image, and the `param/val` pairs are as listed in Table 3.2.

EXAMPLE 3.7: Using function `adapthisteq`.

■ Figure 3.12(a) is the same as Fig. 3.10(a) and Fig. 3.12(b) is the result of using all the default settings in function `adapthisteq`:

```
>> g1 = adapthisteq(f);
```

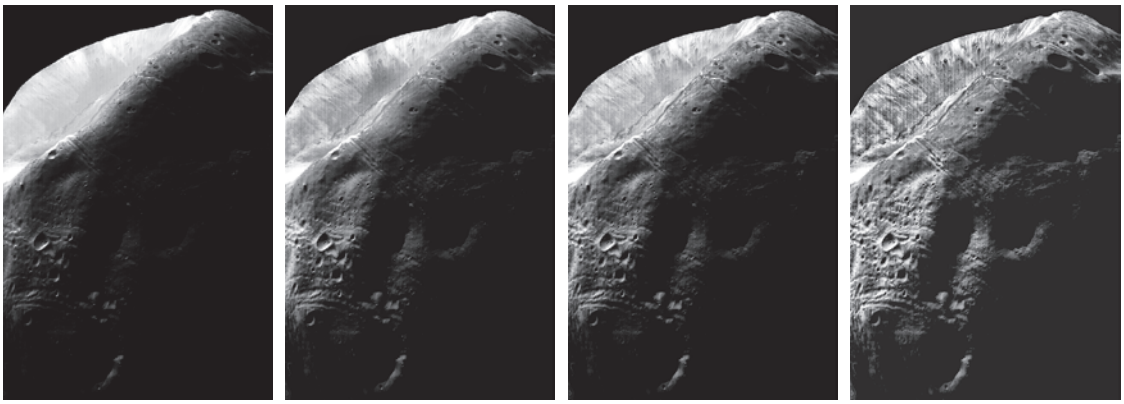
Although this result shows a slight increase in detail, significant portions of the image still are in the shadows. Fig. 3.12(c) shows the result of increasing the size of the tiles to [25 25]:

```
>> g2 = adapthisteq(f, 'NumTiles', [25 25]);
```

Sharpness increased slightly, but no new details are visible. Using the command

TABLE 3.2 Parameters and corresponding values for use in function `adapthisteq`.

Parameter	Value
`'NumTiles'`	Two-element vector of positive integers specifying the number of tiles by row and column, `[r c]`. Both `r` and `c` must be at least 2. The total number of tiles is equal to `r*c`. The default is `[8 8]`.
`'ClipLimit'`	Scalar in the range [0 1] that specifies a contrast enhancement limit. Higher numbers result in more contrast. The default is 0.01.
`'NBins'`	Positive integer scalar specifying the number of bins for the histogram used in building a contrast enhancing transformation. Higher values result in greater dynamic range at the cost of slower processing speed. The default is 256.
`'Range'`	A string specifying the range of the output image data: `'original'` — Range is limited to the range of the original image, `[min(f(:)) max(f(:))]`. `'full'` — Full range of the output image class is used. For example, for `uint8` data, range is [0 255]. This is the default.
`'Distribution'`	A string specifying the desired histogram shape for the image tiles: `'uniform'` — Flat histogram (this is the default). `'rayleigh'` — Bell-shaped histogram. `'exponential'` — Curved histogram. (See Section 5.2.2 for the equations for these distributions.
`'Alpha'`	Nonnegative scalar applicable to the Rayleigh and exponential distributions. The default value is 0.4.



a b c d

FIGURE 3.12 (a) Same as Fig. 3.10(a). (b) Result of using function `adapthisteq` with the default values. (c) Result of using this function with parameter `NumTiles` set to `[25 25]`. Result of using this number of tiles and `ClipLimit` = `0.05`.

```
>> g3 = adapthisteq(f, 'NumTiles', [25 25], 'ClipLimit', 0.05);
```

yielded the result in Fig. 3.12(d). The enhancement in detail in this image is significant compared to the previous two results. In fact, comparing Figs. 3.12(d) and 3.11(b) provides a good example of the advantage that local enhancement can have over global enhancement methods. Generally, the price paid is additional function complexity. ■

3.4 Spatial Filtering

As mentioned in Section 3.1 and illustrated in Fig. 3.1, neighborhood processing consists of (1) selecting a center point, (x, y); (2) performing an operation that involves only the pixels in a predefined neighborhood about (x, y); (3) letting the result of that operation be the "response" of the process at *that* point; and (4) repeating the process for every point in the image. The process of moving the center point creates new neighborhoods, one for each pixel in the input image. The two principal terms used to identify this operation are *neighborhood processing* and *spatial filtering*, with the second term being more prevalent. As explained in the following section, if the computations performed on the pixels of the neighborhoods are linear, the operation is called *linear spatial filtering* (the term *spatial convolution* also used); otherwise it is called *nonlinear spatial filtering*.

3.4.1 Linear Spatial Filtering

The concept of *linear filtering* has its roots in the use of the Fourier transform for signal processing in the frequency domain, a topic discussed in detail in Chapter 4. In the present chapter, we are interested in filtering operations that

are performed directly on the pixels of an image. Use of the term *linear spatial filtering* differentiates this type of process from *frequency domain filtering*.

The linear operations of interest in this chapter consist of multiplying each pixel in the neighborhood by a corresponding coefficient and summing the results to obtain the response at each point (x, y). If the neighborhood is of size $m \times n$, mn coefficients are required. The coefficients are arranged as a matrix, called a *filter*, *mask*, *filter mask*, *kernel*, *template*, or *window*, with the first three terms being the most prevalent. For reasons that will become obvious shortly, the terms *convolution filter*, *convolution mask*, or *convolution kernel*, also are used.

Figure 3.13 illustrates the mechanics of linear spatial filtering. The process consists of moving the center of the filter mask, w, from point to point in an image, f. At each point (x, y), the response of the filter at that point is the sum of products of the filter coefficients and the corresponding neighborhood pixels in the area spanned by the filter mask. For a mask of size $m \times n$, we assume typically that $m = 2a + 1$ and $n = 2b + 1$ where a and b are nonnegative integers. All this says is that our principal focus is on masks of odd sizes, with the smallest meaningful size being 3×3. Although it certainly is not a requirement, working with odd-size masks is more intuitive because they have an unambiguous center point.

There are two closely related concepts that must be understood clearly when performing linear spatial filtering. One is *correlation*; the other is *convolution*. Correlation is the process of passing the mask w by the image array f in the manner described in Fig. 3.13. Mechanically, convolution is the same process, except that w is rotated by 180° prior to passing it by f. These two concepts are best explained by some examples.

Figure 3.14(a) shows a one-dimensional function, f, and a mask, w. The origin of f is assumed to be its leftmost point. To perform the correlation of the two functions, we move w so that its rightmost point coincides with the origin of f, as Fig. 3.14(b) shows. Note that there are points between the two functions that do not overlap. The most common way to handle this problem is to pad f with as many 0s as are necessary to guarantee that there will always be corresponding points for the full excursion of w past f. This situation is illustrated in Fig. 3.14(c).

We are now ready to perform the correlation. The first value of correlation is the sum of products of the two functions in the position shown in Fig. 3.14(c). The sum of products is 0 in this case. Next, we move w one location to the right and repeat the process [Fig. 3.14(d)]. The sum of products again is 0. After four shifts [Fig. 3.14(e)], we encounter the first nonzero value of the correlation, which is (2)(1) = 2. If we proceed in this manner until w moves completely past f [the ending geometry is shown in Fig. 3.14(f)] we would get the result in Fig. 3.14(g). This set of values is the correlation of w and f. If we had padded w, aligned the rightmost element of f with the leftmost element of the padded w, and performed correlation in the manner just explained, the result would have been different (rotated by 180°), so order of the functions matters in correlation.

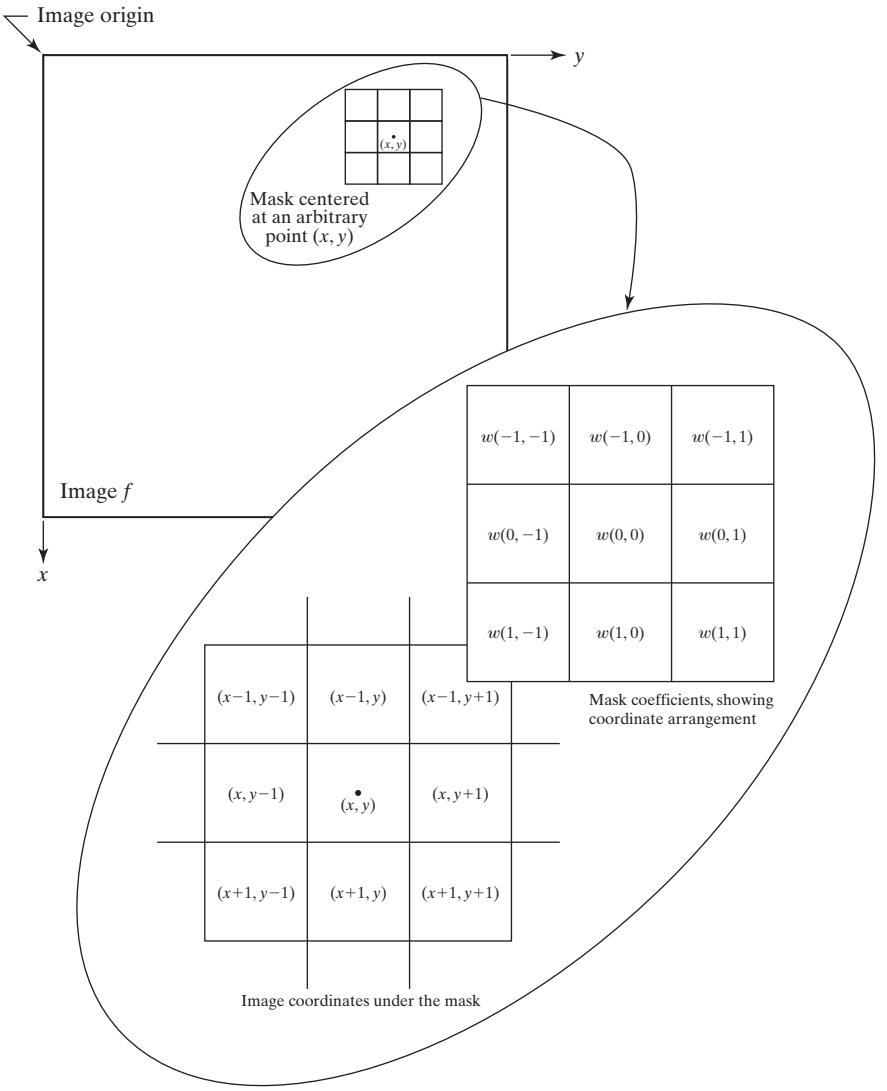


FIGURE 3.13
The mechanics of linear spatial filtering. The magnified drawing shows a 3×3 filter mask and the corresponding image neighborhood directly under it. The image neighborhood is shown displaced out from under the mask for ease of readability.

The label `'full'` in the correlation in Fig. 3.14(g) is a flag (to be discussed later) used by the toolbox to indicate correlation using a padded image and computed in the manner just described. The toolbox provides another option, denoted by `'same'` [Fig. 3.14(h)] that produces a correlation that is of the same size as f. This computation also uses zero padding, but the starting position is with the center point of the mask (the point labeled 3 in w) aligned with the origin of f. The last computation is with the center point of the mask aligned with the last point in f.

To perform convolution we rotate w by 180° and place its rightmost point at the origin of f, as Fig. 3.14(j) shows. We then repeat the sliding/computing

process employed in correlation, as illustrated in Figs. 3.14(k) through (n). The `'full'` and `'same'` convolution results are shown in Figs. 3.14(o) and (p), respectively.

Function f in Fig. 3.14 is a discrete unit impulse that is 1 at a point and 0 everywhere else. It is evident from the result in Figs. 3.14(o) or (p) that convolution with an impulse just "copies" w at the location of the impulse. This copying property (called *sifting*) is a fundamental concept in linear system theory, and it is the reason why one of the functions is always rotated by 180° in convolution. Note that, unlike correlation, swapping the order of the functions yields the same convolution result. If the function being shifted is symmetric, it is evident that convolution and correlation yield the same result.

The preceding concepts extend easily to images, as Fig. 3.15 illustrates. The origin is at the top, left corner of image $f(x, y)$ (see Fig. 2.1). To perform correlation, we place the bottom, rightmost point of $w(x, y)$ so that it coincides with the origin of $f(x, y)$ as in Fig. 3.15(c). Note the use of 0 padding for the

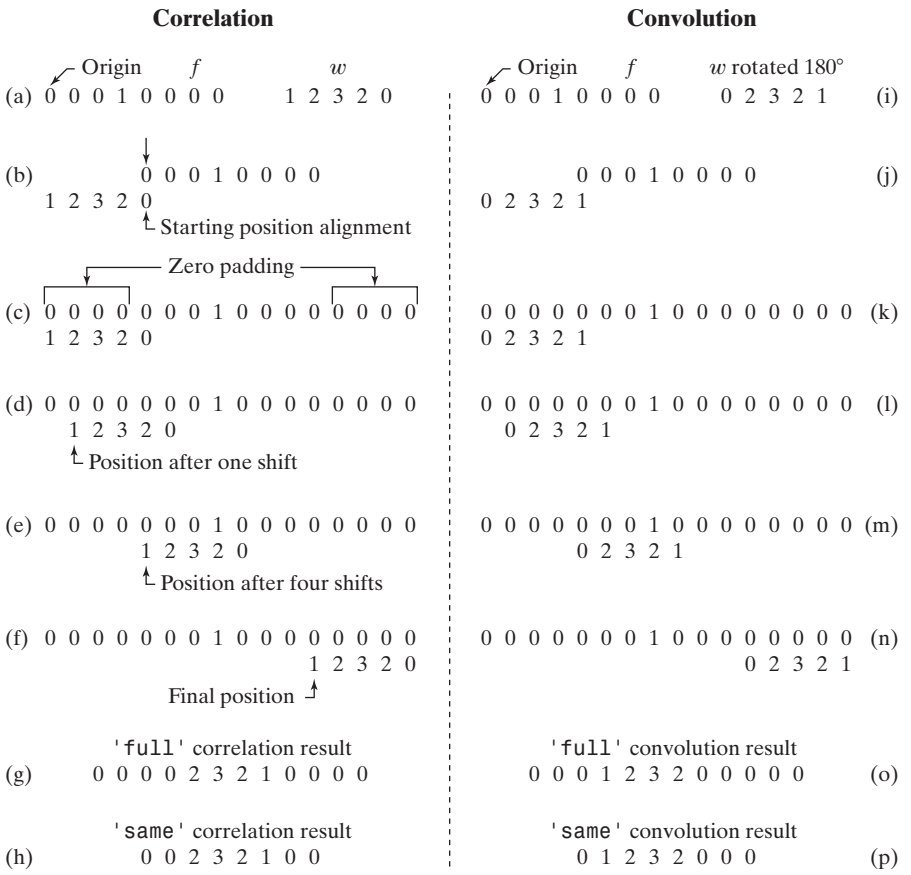


FIGURE 3.14 Illustration of one-dimensional correlation and convolution.

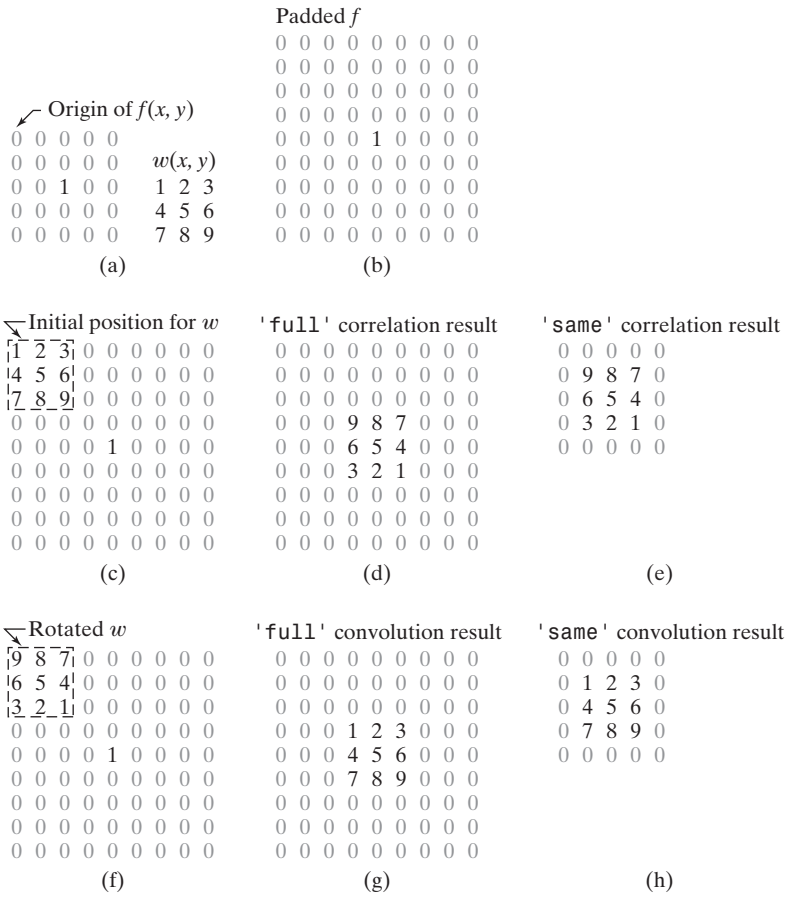


FIGURE 3.15 Illustration of two-dimensional correlation and convolution. The 0s are shown in gray to simplify viewing.

reasons mentioned in the discussion of Fig. 3.14. To perform correlation, we move $w(x, y)$ in all possible locations so that at least one of its pixels overlaps a pixel in the original image $f(x, y)$. This `'full'` correlation is shown in Fig. 3.15(d). To obtain the `'same'` correlation in Fig. 3.15(e), we require that all excursions of $w(x, y)$ be such that its center pixel overlaps the original $f(x, y)$. For convolution, we rotate $w(x, y)$ by 180° and proceed in the same manner as in correlation [see Figs. 3.15(f) through (h)]. As in the one-dimensional example discussed earlier, convolution yields the same result independently of the order of the functions. In correlation the order does matter, a fact that is made clear in the toolbox by assuming that the filter mask is always the function that undergoes translation. Note also the important fact in Figs. 3.15(e) and (h) that the results of spatial correlation and convolution are rotated by 180° with respect to each other. This, of course, is expected because convolution is nothing more than correlation with a rotated filter mask.

Summarizing the preceding discussion in equation form, we have that the correlation of a filter mask $w(x, y)$ of size $m \times n$ with a function $f(x, y)$, denoted by $w(x, y) \star f(x, y)$, is given by the expression

$$w(x, y) \star f(x, y) = \sum_{s=-a}^{a} \sum_{t=-b}^{b} w(s, t) f(x + s, y + t)$$

This equation is evaluated for all values of the *displacement* variables x and y so that all elements of w visit every pixel in f, which we assume has been padded appropriately. Constants a and b are given by $a = (m - 1)/2$ and $b = (n - 1)/2$. For notational convenience, we assume that m and n are odd integers.

In a similar manner, the convolution of $w(x, y)$ and $f(x, y)$, denoted by $w(x, y) \star f(x, y)$, is given by the expression

$$w(x, y) \star f(x, y) = \sum_{s=-a}^{a} \sum_{t=-b}^{b} w(s, t) f(x - s, y - t)$$

where the minus signs on the right of the equation flip f (i.e., rotate it by 180°). Rotating and shifting f instead of w is done to simplify the notation. The result is the same.[†] The terms in the summation are the same as for correlation.

The toolbox implements linear spatial filtering using function `imfilter`, which has the following syntax:


```
g = imfilter(f, w, filtering_mode, boundary_options, size_options)
```

where `f` is the input image, `w` is the filter mask, `g` is the filtered result, and the other parameters are summarized in Table 3.3. The `filtering_mode` is specified as `'corr'` for correlation (this is the default) or as `'conv'` for convolution. The `boundary_options` deal with the border-padding issue, with the size of the border being determined by the size of the filter. These options are explained further in Example 3.8. The `size_options` are either `'same'` or `'full'`, as explained in Figs. 3.14 and 3.15.

The most common syntax for `imfilter` is

```
g = imfilter(f, w, 'replicate')
```

This syntax is used when implementing standard linear spatial filters in the toolbox. These filters, which are discussed in Section 3.5.1, are prerotated by 180°, so we can use the correlation default in `imfilter` (from the discussion of Fig. 3.15, we know that performing correlation with a rotated filter is the same as performing convolution with the original filter). If the filter is symmetric about its center, then both options produce the same result.

[†]Because convolution is commutative, we have that $w(x, y) \star f(x, y) = f(x, y) \star w(x, y)$. This is not true of correlation, as you can see, for example, by reversing the order of the two functions in Fig. 3.14(a).

3.6 Using Fuzzy Techniques for Intensity Transformations and Spatial Filtering

We conclude this chapter with an introduction to fuzzy sets and their application to intensity transformations and spatial filtering. We also develop a set of custom M-functions for implementing the fuzzy methods developed in this section. As you will see shortly, fuzzy sets provide a framework for incorporating human knowledge in the solution of problems whose formulation is based on imprecise concepts.

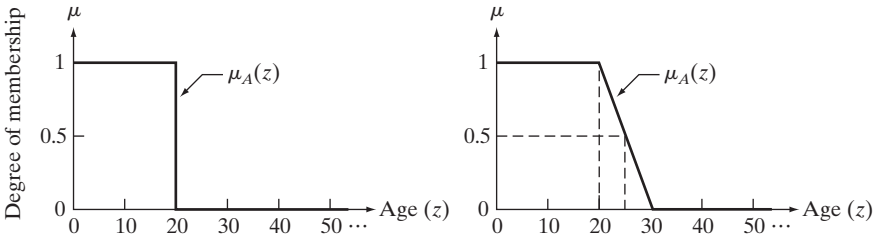
3.6.1 Background

A *set* is a collection of objects (*elements*) and *set theory* consists of tools that deal with operations on and among sets. Central to set theory is the notion of set membership. We are used to dealing with so-called "crisp" sets, whose membership can be only true or false in the traditional sense of bivalued Boolean logic, with 1 typically indicating true and 0 indicating false. For example, let Z denote the set of all people, and suppose that we want to define a subset, A, of Z, called the "set of young people." In order to form this subset, we need to define a *membership function* that assigns a value of 1 or 0 to every element, z, of Z. Because we are dealing with a bivalued logic, the membership function defines a threshold at or below which a person is considered young, and above which a person is considered not young. Figure 3.20(a) summarizes this concept using an age threshold of 20 years, where $\mu_A(z)$ denotes the membership function just discussed.

We see immediately a difficulty with this formulation: A person 20 years of age is considered young, but a person whose age is 20 years and 1 second is not a member of the set of young people. This is a fundamental problem with crisp sets that limits their use in many practical applications. What we need is more flexibility in what we mean by "young;" that is, a *gradual* transition from young to not young. Figure 3.20(b) shows one possibility. The essential feature of this function is that it is infinite-valued, thus allowing a continuous transition between young and not young. This makes it possible to have *degrees* of "youngness." We can make statements now such as a person being young (upper flat end of the curve), relatively young (toward the beginning of the ramp), 50% young (in the middle of the ramp), not so young (toward the end of the ramp), and so on (note that decreasing the slope of the curve in Fig. 3.20(b) introduces more vagueness in what we mean by "young"). These types of vague (*fuzzy*) statements are more consistent with what we humans use when talking imprecisely about age. Thus, we may interpret infinite-valued membership functions as being the foundation of a *fuzzy logic*, and the sets generated using them may be viewed as *fuzzy sets*.

3.6.2 Introduction to Fuzzy Sets

Fuzzy set theory was introduced by L. A. Zadeh (Zadeh [1965]) more than four decades ago. As the following discussion shows, fuzzy sets provide a formalism for dealing with imprecise information.



a b

FIGURE 3.20 Membership functions of (a) a crisp set, and (b) a fuzzy set.

Definitions

Let Z be a set of elements (objects), with a generic element of Z denoted by z; that is, $Z = \{z\}$. Set Z often is referred to as the *universe of discourse*. A *fuzzy set* A in Z is characterized by a *membership function*, $\mu_A(z)$, that associates with each element of Z a real number in the interval [0, 1]. For a particular element z_0 from Z, the value of $\mu_A(z_0)$ represents the *degree of membership* of z_0 in A.

The term *grade of membership* is used also to denote what we have defined as the degree of membership.

The concept of "belongs to," so familiar in ordinary (crisp) sets, does not have the same meaning in fuzzy set theory. With ordinary sets we say that an element either belongs or does not belong to a set. With fuzzy sets we say that all z's for which $\mu_A(z) = 1$ are *full* members of the set A, all z's for which $\mu_A(z)$ is between 0 and 1 have *partial* membership in the set, and all z's for which $\mu_A(z) = 0$ have *zero* degree of membership in the set (which, for all practical purposes, means that they are not members of the set).

For example, in Fig. 3.20(b) $\mu_A(25) = 0.5$, indicating that a person 25 years old has a 0.5 grade membership in the set of young people. Similarly two people of ages 15 and 35 have 1.0 and 0.0 grade memberships in this set, respectively. Therefore, a fuzzy set, A, is an *ordered pair* consisting of values of z and a membership function that assigns a grade of membership in A to each z. That is,

$$A = \{z, \mu_A(z) \mid z \in Z\}$$

When z is continuous, A can have an infinite number of elements. When z is discrete and its range of values is finite, we can tabulate the elements of A explicitly. For example, if the age in Fig. 3.20 is limited to integers, then A can be written explicitly as

$$A = \{(1,1),(2,1),\ldots,(20,1),(21,0.9),(22,0.8),\ldots,(29,0.1),(30,0),(31,0),\ldots\}$$

Note that, based on the preceding definition, $(30, 0)$ and pairs thereafter are included of A, but their degree of membership in this set is 0. In practice, they typically are not included because interest generally is in elements whose degree of membership is nonzero. Because membership functions determine uniquely the degree of membership in a set, the terms *fuzzy set* and *membership function* are used interchangeably in the literature. This is a frequent source of confusion, so you should keep in mind the routine use of these two

terms to mean the same thing. To help you become comfortable with this terminology, we use both terms interchangeably in this section. When $\mu_A(z)$ can have only two values, say, 0 and 1, the membership function reduces to the familiar characteristic function of ordinary sets. Thus, ordinary sets are a special case of fuzzy sets.

Although fuzzy logic and probability operate over the same [0, 1] interval, there is a significant distinction to be made between the two. Consider the example from Fig. 3.20. A probabilistic statement might read: "There is a 50% chance that a person is young," while a fuzzy statement might read "A person's degree of membership in the set of young people is 0.5." The difference between these two statements is important. In the first statement, a person is considered to be either in the set of young or the set of not young people; we simply have only a 50% chance of knowing to which set the person belongs. The second statement presupposes that a person is young to some degree, with that degree being in this case 0.5. Another interpretation is to say that this is an "average" young person: not really young, but not too near being not young. In other words, fuzzy logic is not probabilistic at all; it just deals with degrees of membership in a set. In this sense, we see that fuzzy logic concepts find application in situations characterized by vagueness and imprecision, rather than by randomness.

The following definitions are basic to the material in the following sections.

Empty set: A fuzzy set is *empty* if and only if its membership function is identically zero in Z.

Equality: Two fuzzy sets A and B are *equal*, written $A = B$, if and only if $\mu_A(z) = \mu_B(z)$ for all $z \in Z$.

The notation "for all $z \in Z$" reads "for all z belonging to Z."

Complement: The *complement* (NOT) of a fuzzy set A, denoted by $\bar{A}$, or NOT(A), is defined as the set whose membership function is

$$\mu_{\bar{A}}(z) = 1 - \mu_A(z)$$

for all $z \in Z$.

Subset: A fuzzy set A is a *subset* of a fuzzy set B if and only if

$$\mu_A(z) \leq \mu_B(z)$$

for all $z \in Z$.

Union: The *union* (OR) of two fuzzy sets A and B, denoted $A \cup B$, or A OR B, is a fuzzy set U with membership function

$$\mu_U(z) = \max\left[\mu_A(z), \mu_B(z)\right]$$

for all $z \in Z$.

Intersection: The *intersection* (AND) of two fuzzy sets A and B, denoted, $A \cap B$ or A AND B, is a fuzzy set I with membership function

$$\mu_I(z) = \min\left[\mu_A(z), \mu_B(z)\right]$$

for all $z \in Z$.

Note that the familiar terms NOT, OR, and AND are used interchangeably with the symbols $\bar{}$, $\cup$, and $\cap$ to denote set complementation, union, and intersection, respectively.

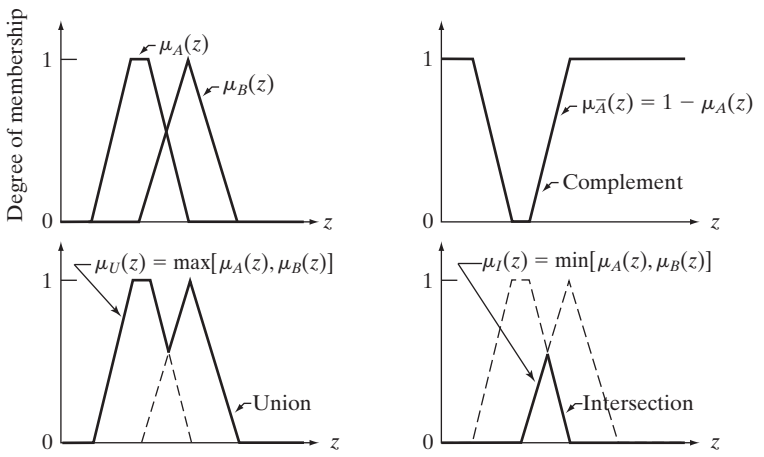
EXAMPLE 3.13: Illustration of fuzzy set definitions.

■ Figure 3.21 illustrates some of the preceding definitions. Figure 3.21(a) shows the membership functions of two sets, A and B, and Fig. 3.21(b) shows the membership function of the complement of A. Figure 3.21(c) shows the membership function of the union of A and B, and Fig. 3.21(d) shows the corresponding result for the intersection of these two sets. The dashed lines in Fig. 3.21are shown for reference only. The results of the fuzzy operations indicated in Figs. 3.21(b)-(d) are the solid lines.

You are likely to encounter examples in the literature in which the area under the curve of the membership function of, say, the intersection of two fuzzy sets, is shaded to indicate the result of the operation. This is a carry over from ordinary set operations and is incorrect. *Only* the points along the membership function itself (solid line) are applicable when dealing with fuzzy sets. This is a good illustration of the comment made earlier that a membership function and its corresponding fuzzy set are one and the same thing. ■

Membership functions

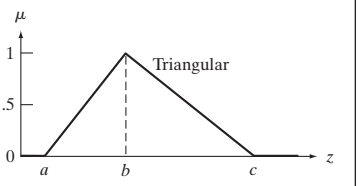
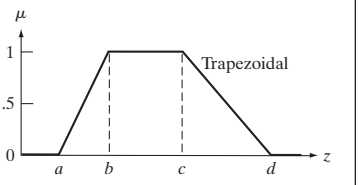
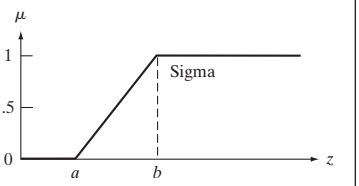
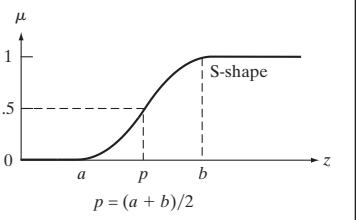
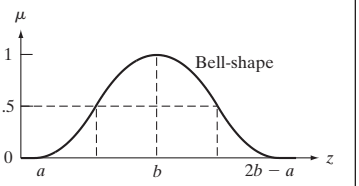
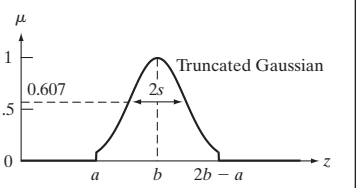
Table 3.6 lists a set of membership functions used commonly for fuzzy set work. The first three functions are piecewise linear, the next two functions are smooth, and the last function is a truncated Gaussian. We develop M-functions in Section 3.6.4 to implement the six membership functions in the table.



a b
c d

FIGURE 3.21
(a) Membership functions of two fuzzy sets, A and B. (b) Membership function of the complement of A. (c) and (d) Membership functions of the union and intersection of A and B.

TABLE 3.6 Some commonly-used membership functions and corresponding plots.

Name	Equation	Plot
Triangular	$\mu(z)=\begin{cases}0 & z<a\\ (z-a)/(b-a) & a\le z<b\\ 1-(z-b)/(c-b) & b\le z<c\\ 0 & c\le z\end{cases}$	
Trapezoidal	$\mu(z)=\begin{cases}0 & z<a\\ (z-a)/(b-a) & a\le z<b\\ 1 & b\le z<c\\ 1-(z-b)/(c-b) & c\le z<d\\ 0 & d\le z\end{cases}$	
Sigma	$\mu(z)=\begin{cases}0 & z<a\\ (z-a)/(b-a) & a\le z<b\\ 1 & b\le z\end{cases}$	
S-shape†	$S(z,a,b)=\begin{cases}0 & z<a\\ 2\left[\frac{z-a}{b-a}\right]^2 & a\le z<p\\ 1-2\left[\frac{z-b}{b-a}\right]^2 & p\le z<b\\ 1 & b\le z\end{cases}$	
Bell-shape	$\mu(z)=\begin{cases}S(z,a,b) & z<b\\ S(2b-z,a,b) & b\le z\end{cases}$	
Truncated Gaussian	$\mu(z)=\begin{cases}e^{-\frac{(z-b)^2}{s^2}} & \lvert z-b\rvert\le(b-a)\\ 0 & \text{otherwise}\end{cases}$	

†Typically, only the independent variable, z, is used as an argument when writing $\mu(z)$ in order to simplify notation. We made an exception in the S-shape curve in order to use its form in writing the equation of the Bell-shape curve.

3.6.3 Using Fuzzy Sets

In this section we develop the foundation for using fuzzy sets, and then apply the concepts developed here to image processing in Sections 3.6.5 and 3.6.6.

We begin the discussion with an example. Suppose that we want to develop a fuzzy system to monitor the health of an electric motor in a power generating station. For our purposes, the health of the motor is determined by the amount of vibration it exhibits. To simplify the discussion, assume that we can accomplish the monitoring task by using a single sensor that outputs a single number: *average vibration frequency*, denoted by z. We are interested in three ranges of average frequency: *low*, *mid*, and *high*. A motor functioning in the low range is said to be operating *normally*, whereas a motor operating in the mid range is said to be performing *marginally*. A motor whose average vibration is in the high range is said to be operating in the *near-failure* mode.

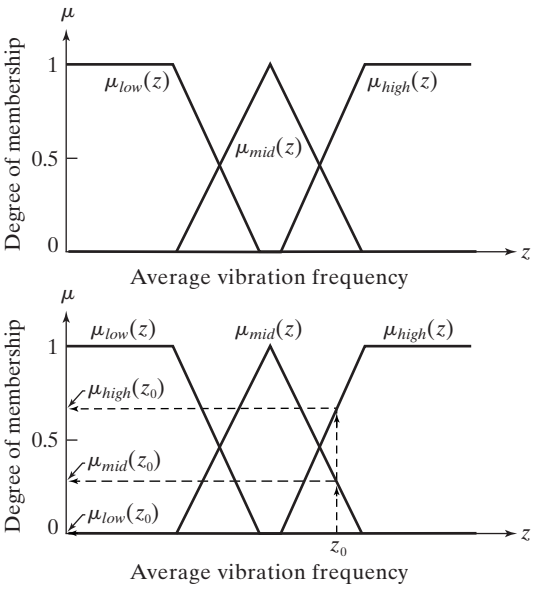
To simplify notation, we use *frequency* to mean *average vibration frequency* from this point on.

The frequency ranges just discussed may be viewed as fuzzy (in a way similar to age in Fig. 3.20), and we can describe the problem using, for example, the fuzzy membership functions in Fig. 3.22(a). Associating variables with fuzzy membership functions is called *fuzzification*. In the present context, frequency is a *linguistic variable*, and a particular value of frequency, z_0, is called a *linguistic value*. A linguistic value is *fuzzified* by using a membership function to map it to the interval $[0, 1]$. Figure 3.22(b) shows an example.

Keeping in mind that the frequency ranges are fuzzy, we can express our knowledge about this problem in terms of the following *fuzzy IF-THEN rules*:

R_1: IF the frequency is *low*, THEN motor operation is *normal*.
OR

The part of an if-then rule to the left of THEN is the *antecedent* (or *premise*). The part to the right is called the *consequent* (or *conclusion*.)



a
b

FIGURE 3.22
(a) Membership functions used to fuzzify frequency measurements. (b) Fuzzifying a specific measurement, z_0.