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* (or *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. 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 will be seen 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

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

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 Fig. 3.1 shows. The center of the region is moved from pixel to pixel starting, say, at the top, left

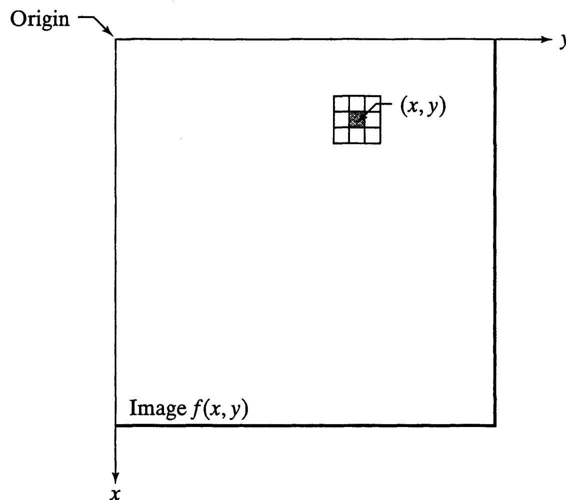


FIGURE 3.1 A neighborhood of size 3 × 3 about a point (x, y) in an image.

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 are used in computing the value of g at (x, y).

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 6.

Because they depend only on intensity values, and not explicitly on (x, y), 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 any corresponding point (x, y) in the images.

3.2.1 Function `imadjust`

Function `imadjust` is the basic IPT tool for intensity transformations of gray-scale images. It has the syntax

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

As illustrated in Fig. 3.2, 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

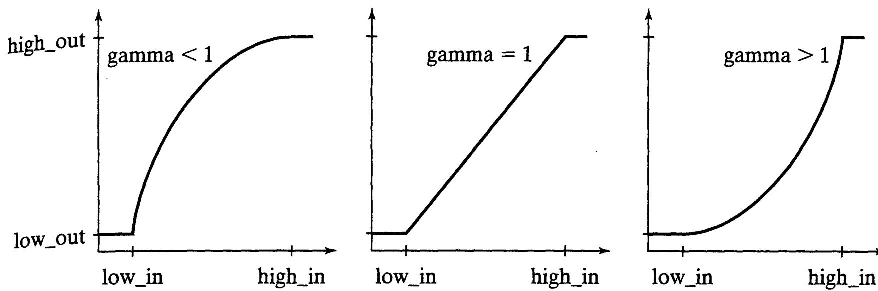


FIGURE 3.2 The various mappings available in function `imadjust`.

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`, or `double`, and the output image has the same class as the input. All inputs to function `imadjust`, other than `f`, are specified as values between 0 and 1, regardless of the class of `f`. If `f` is of class `uint8`, `imadjust` multiplies the values supplied by 255 to determine the actual values to use; if `f` is of class `uint16`, the values are multiplied by 65535. 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.

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 Fig. 3.2(a) shows. 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 IPT function `imcomplement`:

```
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 region 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);
```

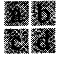


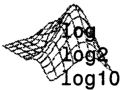
FIGURE 3.3 (a) Original digital mammogram. (b) Negative image. (c) Result of expanding the intensity range [0.5, 0.75]. (d) Result of enhancing the image with `gamma = 2`. (Original image courtesy of G. E. Medical Systems.)

produces 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 [see Fig. 3.3(d)]. ■

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

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



`log` *is the natural logarithm.* `log2` *and* `log10` *are the base* 2 *and base* 10 *logarithms, respectively.*

where `c` is a constant. The shape of this transformation is similar to the gamma curve shown 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 for 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, 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));
```

Use of `mat2gray` brings the values to the range [0, 1] and `im2uint8` brings them to the range [0, 255]. Later, in Section 3.2.3, we discuss a scaling function that automatically detects the class of the input and applies the appropriate conversion.

The function shown in Fig. 3.4(a) is called a *contrast-stretching* transformation function because it compresses the input levels lower than m into a narrow range of dark levels in the output image; similarly, it compresses the values above m into a narrow band of light levels in the output. 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 10, 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 represents 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 an entire image as

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

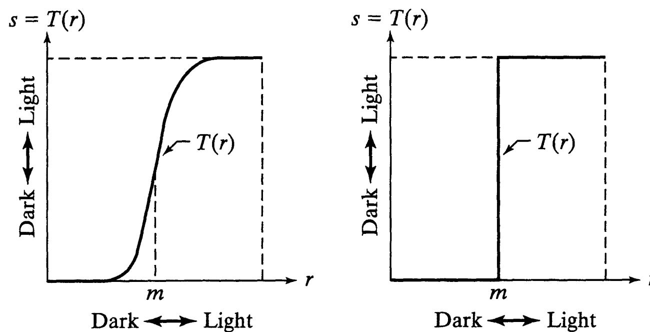


FIGURE 3.4 (a) Contrast-stretching transformation. (b) Thresholding transformation.

Note the use of eps (see Table 2.10) to prevent overflow if f has any 0 values. Since the limiting value of $T(r)$ is 1, output values are scaled to 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 1.5×10^6, displayed on a linearly scaled, 8-bit 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 quite evident. ■

3.2.3 Some Utility M-Functions for Intensity Transformations

In this section we develop two M-functions that incorporate various aspects of the intensity transformations introduced in the previous two 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 IPT 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.

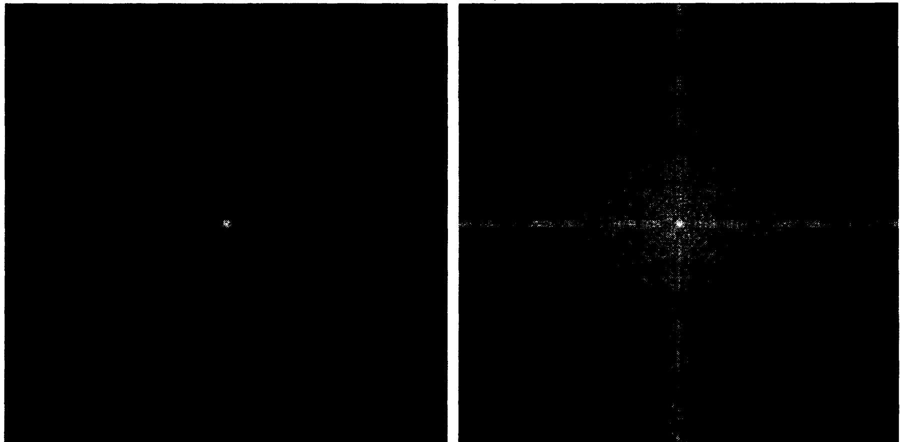


FIGURE 3.5 (a) A Fourier spectrum. (b) Result obtained by performing a log transformation.

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
```

For example, suppose that we execute the following 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 were passed. The syntax is

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

This function returns the message `Not enough input parameters` if `number` is less than `low` or `Too many input parameters` 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.

Often, 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`, 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 2.10.5) that accepts a variable number of inputs 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, it would be perfectly acceptable to have a mixed set of inputs, such as

```
>> [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 indeed 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

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

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 `changeclass`, which has the syntax

```
g = changeclass(newclass, f)
```

This function converts image `f` to the class specified in parameter `newclass` and outputs it as `g`. Valid values for `newclass` are `'uint8'`, `'uint16'`, and `'double'`.

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.

intrans

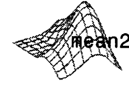
```
function g = intrans(f, varargin)
%INTRANS Performs intensity (gray-level) transformations.
%   G = INTRANS(F, 'neg') computes the negative of input image F.
%
%   G = INTRANS(F, 'log', C, CLASS) computes C*log(1 + F) and
%   multiplies the result by (positive) constant C. If the last two
%   parameters 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 +
%   eps)).^E). Parameter M must be in the range [0, 1]. The default
%   value for M is mean2(im2double(F)), and the default value for E
%   is 4.
%
%   For the 'neg', 'gamma', and 'stretch' transformations, double
%   input images whose maximum value is greater than 1 are scaled
%   first using MAT2GRAY. Other images are converted to double first
%   using IM2DOUBLE. For the 'log' transformation, double images are
%   transformed without being scaled; other images are converted to
%   double first using IM2DOUBLE.
%
%   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))

% Store the class of the input for use later.
classin = class(f);
```

```
% If the input is of class double, and it is outside the range
% [0, 1], and the specified transformation is not 'log', convert the
% input to the range [0, 1].
if strcmp(class(f), 'double') & max(f(:)) > 1 & . . .
      ~strcmp(varargin{1}, 'log')
   f = mat2gray(f);
else % Convert to double, regardless of class(f).
   f = im2double(f);
end

% Determine the type of transformation specified.
method = varargin{1};

% Perform the intensity transformation specified.
switch method
case 'neg'
   g = imcomplement(f);

case 'log'
   if length(varargin) == 1
      c = 1;
   elseif length(varargin) == 2
      c = varargin{2};
   elseif length(varargin) == 3
      c = varargin{2};
      classin = varargin{3};
   else
      error('Incorrect number of inputs for the log option.')
   end
   g = c*(log(1 + double(f)));

case 'gamma'
   if length(varargin) < 2
      error('Not enough inputs for the gamma option.')
   end
   gam = varargin{2};
   g = imadjust(f, [ ], [ ], gam);

case 'stretch'
   if length(varargin) == 1
      % Use defaults.
      m = mean2(f);
      E = 4.0;
   elseif length(varargin) == 3
      m = varargin{2};
      E = varargin{3};
   else error('Incorrect number of inputs for the stretch option.')
   end
   g = 1./(1 + (m./(f + eps)).^E);

otherwise
   error('Unknown enhancement method.')
end

% Convert to the class of the input image.
g = changeclass(classin, 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(im2double(f)), 0.9);
>> figure, imshow(g)
```

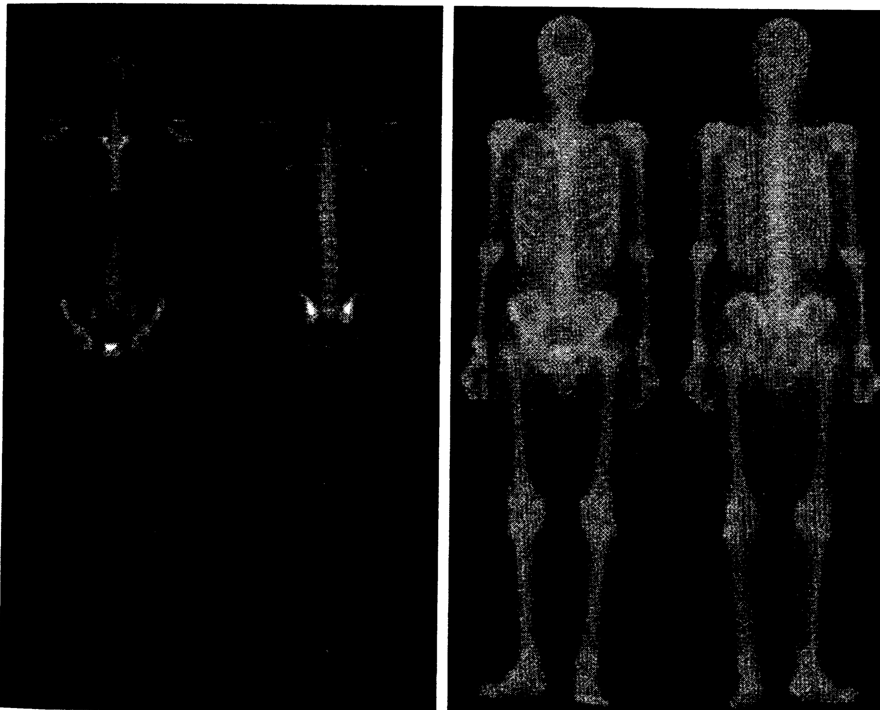


`m = mean2(A)` *computes the mean (average) value of the elements of matrix* `A`.

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 `double` using `im2double` 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, results whose pixels span a wide negative to positive range of values 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 often is desirable to scale the image to the full, maximum range, [0, 255] or [0, 65535]. The following 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.



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.)

The syntax of function `gscale` is

gscale

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

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 of class `double` 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 basic 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 images of class `double`. Keep in mind that indices in MATLAB cannot be 0, so r_1 corresponds to intensity level 0, r_2 corresponds to intensity level 1, and so on, with r_L corresponding to level G. Note also that $G = L - 1$ for images of class `uint8` and `uint16`.

Often, it is useful 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:

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

for $k = 1, 2, \ldots, L$. 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`, which has the following basic syntax:

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

where `f` is the input image, `h` is its histogram, $h(r_k)$, 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 simply 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 is to use `imhist` with no output specified:

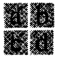
```
>> imhist(f);
```

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 often are plotted using *bar* graphs. For this purpose we can use the function

```
bar(horz, v, width)
```

where `v` is a row vector containing the points to be plotted, `horz` is a vector of the same dimension as `v` that contains the increments of the horizontal scale, and `width` is a number between 0 and 1. If `horz` is omitted, the horizontal axis is divided in units from 0 to `length(v)`. When `width` is 1, the bars touch; when it is 0, the bars are simply vertical lines, as in Fig. 3.7(a). 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 statements produce a bar graph, with the horizontal axis divided into groups of 10 levels:



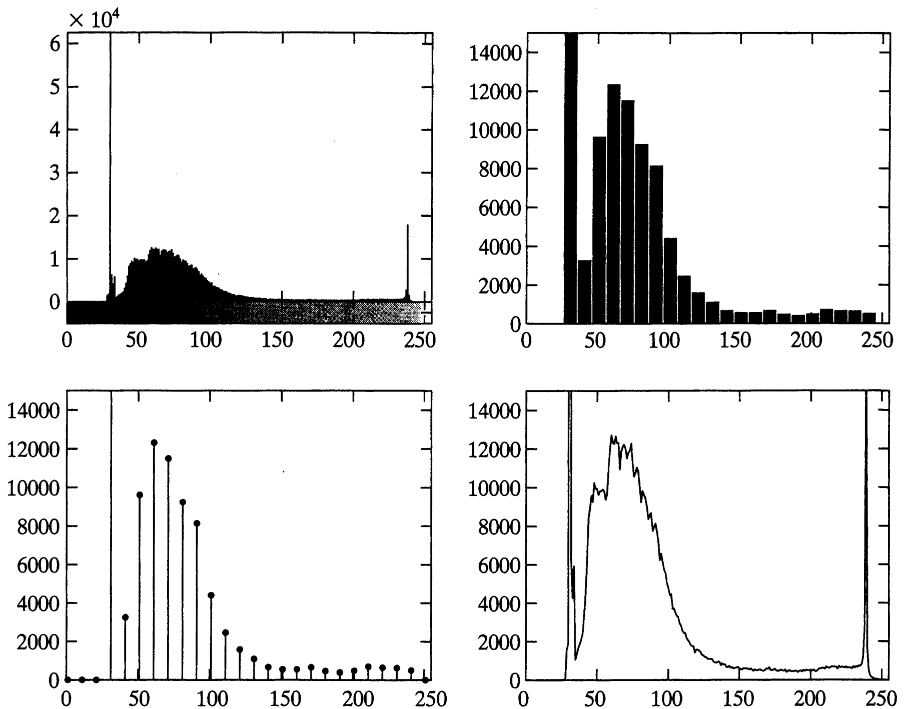


FIGURE 3.7 Various ways to plot an image histogram. (a) imhist, (b) bar, (c) stem, (d) plot.

```
>> h = imhist(f);
>> h1 = h(1:10:256);
>> horz = 1:10:256;
>> bar(horz, h1)
>> axis([0 255 0 15000])
>> set(gca, 'xtick', 0:50:255)
>> set(gca, 'ytick', 0:2000:15000)
```

Figure 3.7(b) shows the result. The peak located at the high end of the intensity scale in Fig. 3.7(a) is missing in the bar graph as a result of the larger horizontal increments used in the plot.

The fifth statement in the preceding code was used to expand the lower range of the vertical axis for visual analysis, and to set the orizontal axis to the same range as in Fig. 3.7(a). The `axis` function has the syntax

```
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.

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)
```

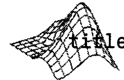
where size is the font size in points. Text can be added to the body of the figure by using function text, as follows:

```
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 3.5. 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, v, 'color_linestyle_marker', 'fill')
```

See the stem *help page for additional options available for this function.*

where v is row vector containing the points to be plotted, and horz is as described for bar. The argument,

```
color_linestyle_marker
```

is a triplet of values from Table 3.1. For example, stem(v, 'r--s') produces a stem plot where the lines and markers are red, the lines are dashed, and the markers are squares. If fill is used, and the marker is a circle, square, or diamond, the marker is filled with the color specified in color. The default color is black, 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);
>> h1 = h(1:10:256);
```

TABLE 3.1 Attributes for functions stem and plot. The none attribute is applicable only to function plot, and must be specified individually. See the syntax for function plot below.

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	none	No line	x	Cross
c	Cyan			s	Square
y	Yellow			d	Diamond
m	Magenta			none	No marker

```
>> horz = 1:10:256;
>> stem(horz, h1, 'fill')
>> axis([0 255 0 15000])
>> set(gca, 'xtick', [0:50:255])
>> set(gca, 'ytick', [0:2000:15000])
```

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

```
plot(horz, v, 'color_linestyle_marker')
```

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

where the arguments are as defined previously for stem plots. The values of color, linestyle, and marker are given in Table 3.1. As in stem, the attributes in plot can be specified as a triplet. When using none for linestyle or for marker, the attributes must be specified individually. For example, the command

```
>> plot(horz, v, 'color', 'g', 'linestyle', 'none', 'marker', 's')
```

plots green squares without connecting lines between them. The defaults for plot are solid black lines with no markers.

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

```
>> h = imhist(f);
>> plot(h)   % 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). ■

In the preceding discussion axis limits and tick marks were set manually. It is possible to set the limits and ticks automatically by using 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 online help 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. See Example 10.6 for an illustration.

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 [2002]) that the probability density function of the output levels is *uniform*; that is,

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

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 $p_r(r_j)$, $j = 1, 2, \ldots, L$, 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

$$\begin{aligned} s_k &= T(r_k) \\ &= \sum_{j=1}^{k} p_r(r_j) \\ &= \sum_{j=1}^{k} \frac{n_j}{n} \end{aligned}$$

for $k = 1, 2, \ldots, L$, 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, $T(r_k)$, 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. This can be seen in the histogram in Fig. 3.8(b), in which the dark nature of the image is expected because the histogram is biased toward the dark end of the gray scale. The low dynamic range is evident from the fact that the "width" of 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)
>> figure, imhist(f)
>> ylim('auto')
>> g = histeq(f, 256);
>> figure, imshow(g)
>> figure, imhist(g)
>> ylim('auto')
```

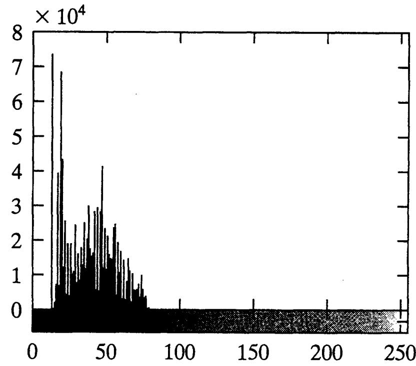
The images were saved to disk in tiff format at 300 dpi using `imwrite`, and the plots were similarly exported to disk using the `print` function discussed in Section 2.4.

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

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

As noted earlier, the transformation function $T(r_k)$ is simply the cumulative sum of normalized histogram values. We can use function `cumsum` to obtain the transformation function, as follows:

```
>> hnorm = imhist(f)./numel(f);
>> cdf = cumsum(hnorm);
```



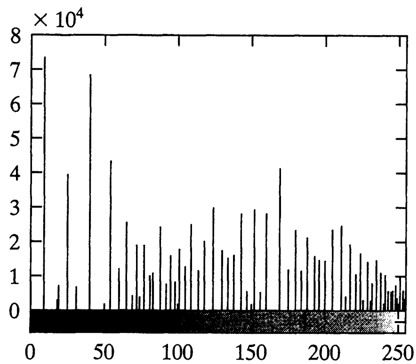


FIGURE 3.8 Illustration of histogram equalization. (a) Input image, and (b) its histogram. (c) Histogram-equalized image, and (d) its histogram. The improvement between (a) and (c) is quite visible. (Original image courtesy of Dr. Roger Heady, Research School of Biological Sciences, Australian National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

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

```
>> x = linspace(0, 1, 256);    % Intervals for [0, 1] horiz scale. Note
                               % the use of linspace from Sec. 2.8.1.
>> 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)
>> % Specify text in the body of the graph:
>> text(0.18, 0.5, 'Transformation function', 'fontsize', 9)
```

We can tell visually from this transformation function that a narrow range of input intensity levels is transformed into the full intensity scale in the output image. ■

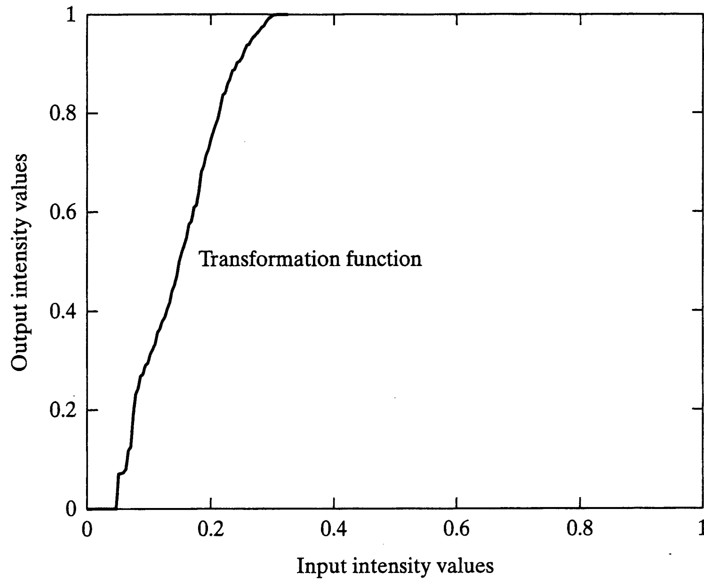


FIGURE 3.9 Transformation function used to map the intensity values from the input image in Fig. 3.8(a) to the values of the output image in Fig. 3.8(c).

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 a processed 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, that have 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, which have the specified density $p_z(z)$. From the preceding two equations, it follows that

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

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 PDF is the specified $p_z(z)$, as long as we can find H^{-1}. When working with discrete variables, we can guarantee that the inverse of H exists if $p_z(z)$ 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 $p_z(z)$ 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)
```

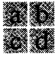
where `f` is the input image, `hspec` is the specified histogram (a row vector of specified values), and `g` is the output image, whose histogram approximates the specified histogram, `hspec`. This vector should contain integer counts corresponding to equally spaced bins. A property of `histeq` is that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

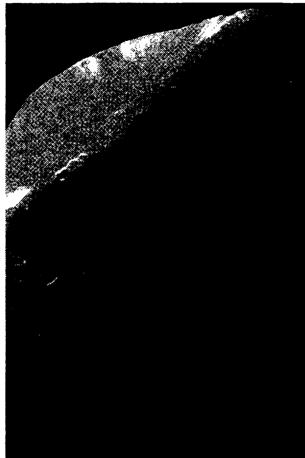
EXAMPLE 3.6: Histogram matching.

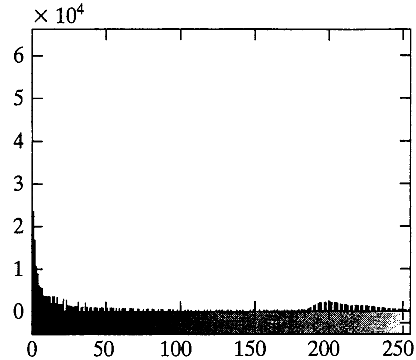
■ Figure 3.10(a) shows an image, `f`, of the Mars moon, Phobos, and Fig. 3.10(b) shows its histogram, obtained using `imhist(f)`. The image is dominated by large, dark areas, resulting in a histogram characterized by a large concentration of pixels in the dark end of the gray scale. At first glance, 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 did not produce 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 that the intensity levels have been shifted to the upper one-half of the gray scale, thus giving the image a washed-out appearance. The cause of the shift is the large concentration of dark components at or near 0 in the original histogram. In turn, 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.







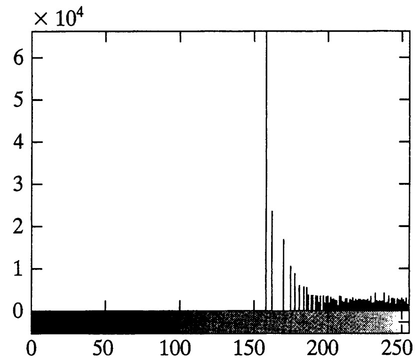


FIGURE 3.10 (a) Image of the Mars moon Phobos. (b) Histogram. (c) Histogram-equalized image. (d) Histogram of (c). (Original image courtesy of NASA).

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 bimodal Gaussian function.
%   P = TWOMODEGAUSS(M1, SIG1, M2, SIG2, A1, A2, K) generates a bimodal,
%   Gaussian-like function in the interval [0, 1]. P is a 256-element
%   vector normalized so that SUM(P) equals 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);

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 2.10.5 for an explanation of the functions `input` and `str2num`. Note how the limits of the plots are set.

manualhist

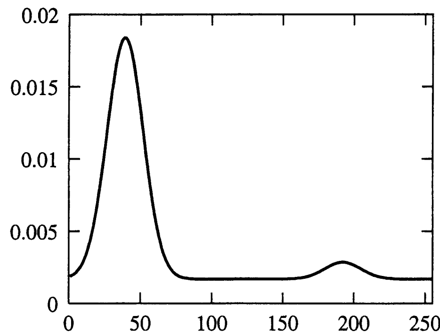
```
function p = manualhist
%MANUALHIST Generates a bimodal histogram interactively.
%   P = MANUALHIST generates a bimodal histogram using
%   TWOMODEGAUSS(m1, sig1, m2, sig2, A1, A2, k). m1 and m2 are the means
%   of the two modes and must be in the range [0, 1]. sig1 and sig2 are
%   the standard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

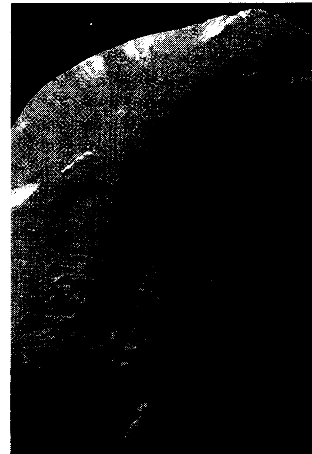
```
        disp('Incorrect number of inputs.')
        continue
    end
    p = twomodegauss(v(1), v(2), v(3), v(4), v(5), v(6), v(7));
    % Start a new figure and scale the axes. Specifying only xlim
    % leaves ylim on auto.
    figure, plot(p)
    xlim([0 255])
end
```

Since 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

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

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





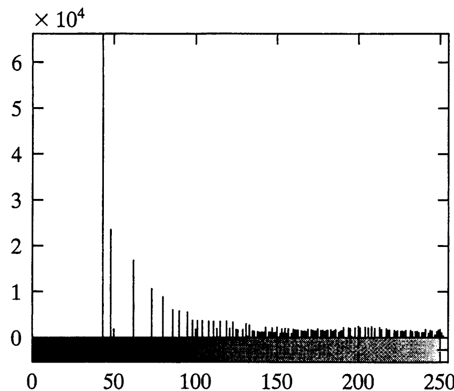


Figure 3.11(b) shows the result. The improvement over the histogram-equalized result in Fig. 3.10(c) is evident by comparing the two images. It is of interest to note that the specified histogram represents a rather modest change from the original histogram. This is all that was required to obtain a 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 the 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 shown in Fig. 3.10(d), which corresponds to the poorly enhanced image of Fig. 3.10(c). ■

3.4 Spatial Filtering

As mentioned in Section 3.1 and illustrated in Fig. 3.1, neighborhood processing consists of (1) defining a center point, (x, y); (2) performing an operation that involves only the pixels in a predefined neighborhood about that center point; (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*, *mask*, or *kernel*, also are used.

The mechanics of linear spatial filtering are illustrated in Fig. 3.12. The process consists simply 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

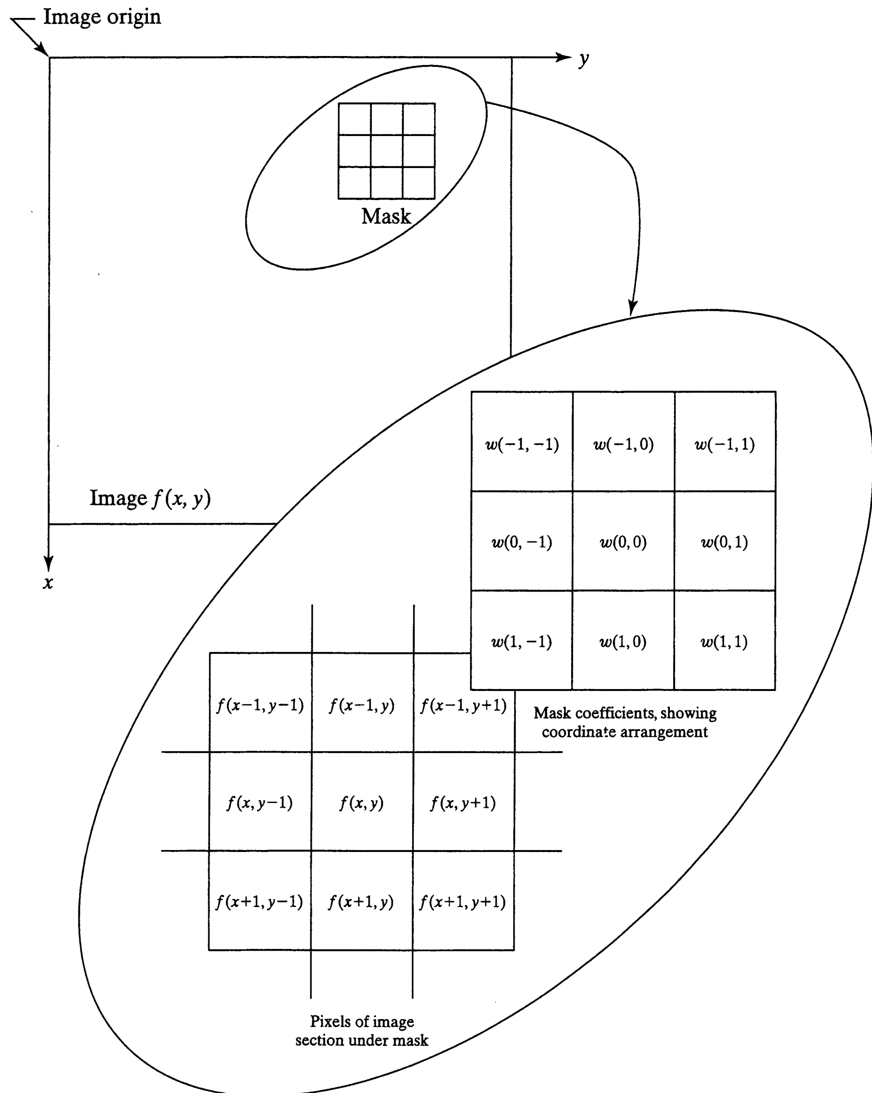


FIGURE 3.12 The mechanics of linear spatial filtering. The magnified drawing shows a 3×3 mask and the corresponding image neighborhood directly under it. The neighborhood is shown displaced out from under the mask for ease of readability.

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 (we exclude from our discussion the trivial case of a 1×1 mask). Although it certainly is not a requirement, working with odd-size masks is more intuitive because they have a unique 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. 3.12. 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 simple examples.

Figure 3.13(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 shown in Fig. 3.13(b). 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 shown in Fig. 3.13(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. 3.13(c). The sum of products is 0 in this case. Next, we move w one location to the right and repeat the process [Fig. 3.13(d)]. The sum of products again is 0. After four shifts [Fig. 3.13(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. 3.13(f)] we would get the result in Fig. 3.13(g). This set of values is the correlation of w and f. Note that, had we left w stationary and had moved f past w instead, the result would have been different, so the order matters.

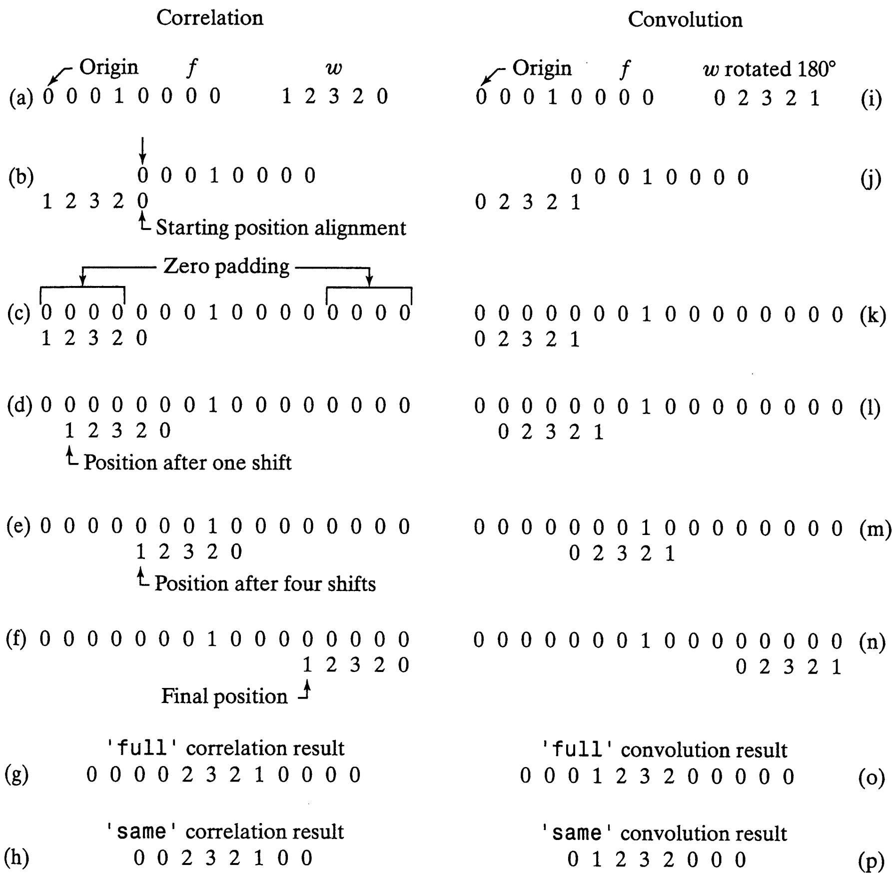


FIGURE 3.13 Illustration of one-dimensional correlation and convolution.

The label `'full'` in the correlation shown in Fig. 3.13(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. 3.13(h)] that produces a correlation that is 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 shown in Fig. 3.13(j). We then repeat the sliding/computing process employed in correlation, as illustrated in Figs. 3.13(k) through (n). The `'full'` and `'same'` convolution results are shown in Figs. 3.13(o) and (p), respectively.

Function f in Fig. 3.13 is a discrete unit impulse function that is 1 at one location and 0 everywhere else. It is evident from the result in Figs. 3.13(o) or (p) that convolution basically just "copied" w at the location of the impulse. This simple 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, reversing 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 illustrated in Fig. 3.14. 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 illustrated in Fig. 3.14(c). Note the use of 0 padding for the reasons mentioned in the discussion of Fig. 3.13. 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. 3.14(d). To obtain the `'same'` correlation shown in Fig. 3.14(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 simply rotate $w(x, y)$ by 180° and proceed in the same manner as in correlation [Figs. 3.14(f) through (h)]. As in the one-dimensional example discussed earlier, convolution yields the same result regardless of which of the two functions undergoes translation. 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. 3.14(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.

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

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

```
   Origin of f(x, y)
0 0 0 0 0
0 0 0 0 0    w(x, y)
0 0 1 0 0    1 2 3
0 0 0 0 0    4 5 6
0 0 0 0 0    7 8 9
      (a)
```

```
Padded f
0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0
0 0 0 0 1 0 0 0 0
0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0
      (b)
```

```
  Initial position for w
|1 2 3| 0 0 0 0 0 0
|4 5 6| 0 0 0 0 0 0
|7 8 9| 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0
 0 0 0 0 1 0 0 0 0
 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0
      (c)
```

```
'full' correlation result
0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0
0 0 0 9 8 7 0 0 0
0 0 0 6 5 4 0 0 0
0 0 0 3 2 1 0 0 0
0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0
      (d)
```

```
'same' correlation result
0 0 0 0 0
0 9 8 7 0
0 6 5 4 0
0 3 2 1 0
0 0 0 0 0
      (e)
```

```
  Rotated w
|9 8 7| 0 0 0 0 0 0
|6 5 4| 0 0 0 0 0 0
|3 2 1| 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0
 0 0 0 0 1 0 0 0 0
 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0
      (f)
```

```
'full' convolution result
0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0
0 0 0 1 2 3 0 0 0
0 0 0 4 5 6 0 0 0
0 0 0 7 8 9 0 0 0
0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0
      (g)
```

```
'same' convolution result
0 0 0 0 0
0 1 2 3 0
0 4 5 6 0
0 7 8 9 0
0 0 0 0 0
      (h)
```

FIGURE 3.14 Illustration of two-dimensional correlation and convolution. The 0s are shown in gray to simplify viewing.

where `f` is the input image, `w` is the filter mask, `g` is the filtered result, and the other parameters are summarized in Table 3.2. The `filtering_mode` specifies whether to filter using correlation (`'corr'`) or convolution (`'conv'`). 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 3.7. The `size_options` are either `'same'` or `'full'`, as explained in Figs. 3.13 and 3.14.

The most common syntax for `imfilter` is

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

This syntax is used when implementing IPT standard linear spatial filters. 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. 3.14, 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.

TABLE 3.2 Options for function `imfilter`.

Options	Description
Filtering Mode	
`'corr'`	Filtering is done using correlation (see Figs. 3.13 and 3.14). This is the default.
`'conv'`	Filtering is done using convolution (see Figs. 3.13 and 3.14).
Boundary Options	
`P`	The boundaries of the input image are extended by padding with a value, `P` (written without quotes). This is the default, with value 0.
`'replicate'`	The size of the image is extended by replicating the values in its outer border.
`'symmetric'`	The size of the image is extended by mirror-reflecting it across its border.
`'circular'`	The size of the image is extended by treating the image as one period a 2-D periodic function.
Size Options	
`'full'`	The output is of the same size as the extended (padded) image (see Figs. 3.13 and 3.14).
`'same'`	The output is of the same size as the input. This is achieved by limiting the excursions of the center of the filter mask to points contained in the original image (see Figs. 3.13 and 3.14). This is the default.

When working with filters that are neither pre-rotated nor symmetric, and we wish to perform convolution, we have two options. One is to use the syntax

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

`rot90(w, k)` *rotates* `w` *by* `k*90` *degrees, where* `k` *is an integer.*

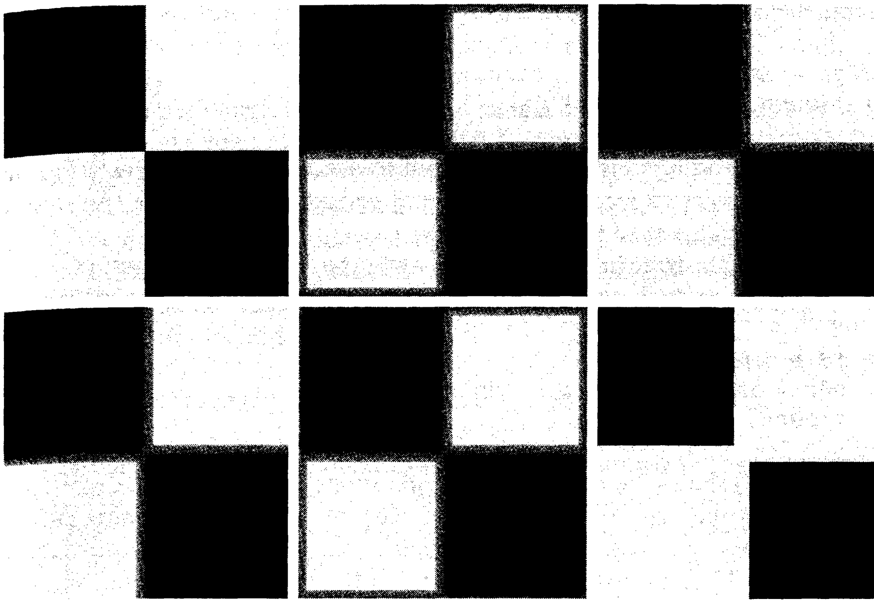
The other approach is to preprocess w by using the function `rot90(w, 2)` to rotate it 180°, and then use `imfilter(f, w, 'replicate')`. Of course these two steps can be combined into one statement. The preceding syntax produces an image g that is of the same size as the input (i.e., the default in computation is the `'same'` mode discussed earlier).

Each element of the filtered image is computed using double-precision, floating-point arithmetic. However, `imfilter` converts the output image to the same class of the input. Therefore, if `f` is an integer array, then output elements that exceed the range of the integer type are truncated, and fractional values are rounded. If more precision is desired in the result, then `f` should be converted to class `double` by using `im2double` or `double` before using `imfilter`.

EXAMPLE 3.7: Using function `imfilter`.

■ Figure 3.15(a) is a class `double` image, `f`, of size 512 × 512 pixels. Consider the simple 31 × 31 filter

```
>> w = ones(31);
```



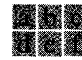


FIGURE 3.15 (a) Original image. (b) Result of using `imfilter` with default zero padding. (c) Result with the `'replicate'` option. (d) Result with the `'symmetric'` option. (e) Result with the `'circular'` option. (f) Result of converting the original image to class `uint8` and then filtering with the `'replicate'` option. A filter of size 31 × 31 with all 1s was used throughout.

which is proportional to an averaging filter. We did not divide the coefficients by $(31)^2$ to illustrate at the end of this example the scaling effects of using `imfilter` with an image of class `uint8`.

Convolving filter `w` with an image produces a blurred result. Because the filter is symmetric, we can use the correlation default in `imfilter`. Figure 3.15(b) shows the result of performing the following filtering operation:

```
>> gd = imfilter(f, w);
>> imshow(gd, [ ])
```

where we used the default boundary option, which pads the border of the image with 0's (black). As expected the edges between black and white in the filtered image are blurred, but so are the edges between the light parts of the image and the boundary. The reason, of course, is that the padded border is black. We can deal with this difficulty by using the `'replicate'` option

```
>> gr = imfilter(f, w, 'replicate');
>> figure, imshow(gr, [ ])
```

As Fig. 3.15(c) shows, the borders of the filtered image now appear as expected. In this case, equivalent results are obtained with the `'symmetric'` option

```
>> gs = imfilter(f, w, 'symmetric');
>> figure, imshow(gs, [ ])
```

Figure 3.15(d) shows the result. However, using the `'circular'` option

```
>> gc = imfilter(f, w, 'circular');
>> figure, imshow(gc, [ ])
```

produced the result in Fig. 3.15(e), which shows the same problem as with zero padding. This is as expected because use of periodicity makes the black parts of the image adjacent to the light areas.

Finally, we illustrate how the fact that `imfilter` produces a result that is of the same class as the input can lead to difficulties if not handled properly:

```
>> f8 = im2uint8(f);
>> g8r = imfilter(f8, w, 'replicate');
>> figure, imshow(g8r, [ ])
```

Figure 3.15(f) shows the result of these operations. Here, when the output was converted to the class of the input (`uint8`) by `imfilter`, clipping caused some data loss. The reason is that the coefficients of the mask did not sum to the range [0, 1], resulting in filtered values outside the [0, 255] range. Thus, to avoid this difficulty, we have the option of normalizing the coefficients so that their sum is in the range [0, 1] (in the present case we would divide the coefficients by $(31)^2$, so the sum would be 1), or inputting the data in `double` format. Note, however, that even if the second option were used, the data usually would have to be normalized to a valid image format at some point (e.g., for storage) anyway. Either approach is valid; the key point is that data ranges have to be kept in mind to avoid unexpected results. ■

3.4.2 Nonlinear Spatial Filtering

Nonlinear spatial filtering is based on neighborhood operations also, and the mechanics of defining $m \times n$ neighborhoods by sliding the center point through an image are the same as discussed in the previous section. However, whereas linear spatial filtering is based on computing the sum of products (which is a linear operation), nonlinear spatial filtering is based, as the name implies, on nonlinear operations involving the pixels of a neighborhood. For example, letting the response at each center point be equal to the maximum pixel value in its neighborhood is a nonlinear filtering operation. Another basic difference is that the concept of a mask is not as prevalent in nonlinear processing. The idea of filtering carries over, but the "filter" should be visualized as a nonlinear function that operates on the pixels of a neighborhood, and whose response constitutes the response of the operation at the center pixel of the neighborhood.

The toolbox provides two functions for performing general nonlinear filtering: `nlfilter` and `colfilt`. The former performs operations directly in 2-D, while `colfilt` organizes the data in the form of columns. Although `colfilt` requires more memory, it generally executes significantly faster than `nlfilter`.

In most image processing applications speed is an overriding factor, so `colfilt` is preferred over `nlfilt` for implementing generalized nonlinear spatial filtering.

Given an input image, `f`, of size $M \times N$, and a neighborhood of size $m \times n$, function `colfilt` generates a matrix, call it `A`, of maximum size $mn \times MN$,[†] in which each column corresponds to the pixels encompassed by the neighborhood centered at a location in the image. For example, the first column corresponds to the pixels encompassed by the neighborhood when its center is located at the top, leftmost point in `f`. All required padding is handled transparently by `colfilt` (using zero padding).

The syntax of function `colfilt` is

```
g = colfilt(f, [m n], 'sliding', @fun, parameters)
```

where, as before, `m` and `n` are the dimensions of the filter region, `'sliding'` indicates that the process is one of sliding the $m \times n$ region from pixel to pixel in the input image `f`, `@fun` references a function, which we denote arbitrarily as `fun`, and `parameters` indicates parameters (separated by commas) that may be required by function `fun`. The symbol `@` is called a *function handle*, a MATLAB data type that contains information used in referencing a function. As will be demonstrated shortly, this is a particularly powerful concept.

Because of the way in which matrix `A` is organized, function `fun` must operate on each of the columns of `A` individually and return a row vector, `v`, containing the results for all the columns. The kth element of `v` is the result of the operation performed by `fun` on the kth column of `A`. Since there can be up to MN columns in `A`, the maximum dimension of `v` is $1 \times MN$.

The linear filtering discussed in the previous section has provisions for padding to handle the border problems inherent in spatial filtering. When using `colfilt`, however, the input image must be padded explicitly before filtering. For this we use function `padarray`, which, for 2-D functions, has the syntax

```
fp = padarray(f, [r c], method, direction)
```

where `f` is the input image, `fp` is the padded image, `[r c]` gives the number of rows and columns by which to pad `f`, and `method` and `direction` are as explained in Table 3.3. For example, if `f = [1 2; 3 4]`, the command

```
>> fp = padarray(f, [3 2], 'replicate', 'post')
```

[†] `A` *always* has mn rows, but the number of columns can vary, depending on the size of the input. Size selection is managed automatically by `colfilt`.

TABLE 3.3 Options for function `padarray`.

Options	Description
Method	
`'symmetric'`	The size of the image is extended by mirror-reflecting it across its border.
`'replicate'`	The size of the image is extended by replicating the values in its outer border.
`'circular'`	The size of the image is extended by treating the image as one period of a 2-D periodic function.
Direction	
`'pre'`	Pad before the first element of each dimension.
`'post'`	Pad after the last element of each dimension.
`'both'`	Pad before the first element and after the last element of each dimension. This is the default.

produces the result

```
fp =

     1   2   2   2
     3   4   4   4
     3   4   4   4
     3   4   4   4
     3   4   4   4
```

If `direction` is not included in the argument, the default is `'both'`. If `method` is not included, the default padding is with 0's. If neither parameter is included in the argument, the default padding is 0 and the default direction is `'both'`. At the end of computation, the image is cropped back to its original size.

EXAMPLE 3.8: Using function `colfilt` to implement a nonlinear spatial filter.

■ As an illustration of function `colfilt`, we implement a nonlinear filter whose response at any point is the geometric mean of the intensity values of the pixels in the neighborhood centered at that point. The geometric mean in a neighborhood of size $m \times n$ is the product of the intensity values in the neighborhood raised to the power $1/mn$. First we implement the nonlinear filter function, call it `gmean`:

```
function v = gmean(A)
mn = size(A, 1); % The length of the columns of A is always mn.
v = prod(A, 1).^(1/mn);
```

`prod(A)` *returns the product of the elements of* `A`. `prod(A, dim)` *returns the product of the elements of* `A` *along dimension* `dim`.

To reduce border effects, we pad the input image using, say, the `'replicate'` option in function `padarray`:

```
>> f = padarray(f, [m n], 'replicate');
```

Finally, we call `colfilt`:

```
>> g = colfilt(f, [m n], 'sliding', @gmean);
```

There are several important points at play here. First, note that, although matrix `A` is part of the argument in function `gmean`, it is not included in the parameters in `colfilt`. This matrix is passed automatically to `gmean` by `colfilt` using the function handle. Also, because matrix `A` is managed automatically by `colfilt`, the number of columns in `A` is variable (but, as noted earlier, the number of rows, that is, the column length, is always `mn`). Therefore, the size of `A` must be computed each time the function in the argument is called by `colfilt`. The filtering process in this case consists of computing the product of all pixels in the neighborhood and then raising the result to the power $1/mn$. For any value of (x, y), the filtered result at that point is contained in the appropriate column in `v`. The function identified by the handle, `@`, can be any function callable from where the function handle was created. The key requirement is that the function operate on the columns of `A` and return a row vector containing the result for all individual columns. Function `colfilt` then takes those results and rearranges them to produce the output image, `g`. ■

Some commonly used nonlinear filters can be implemented in terms of other MATLAB and IPT functions such as `imfilter` and `ordfilt2` (see Section 3.5.2). Function `spfilt` in Section 5.3, for example, implements the geometric mean filter in Example 3.8 in terms of `imfilter` and the MATLAB `log` and `exp` functions. When this is possible, performance usually is much faster, and memory usage is a fraction of the memory required by `colfilt`. Function `colfilt`, however, remains the best choice for nonlinear filtering operations that do not have such alternate implementations.

3.5 Image Processing Toolbox Standard Spatial Filters

In this section we discuss linear and nonlinear spatial filters supported by IPT. Additional nonlinear filters are implemented in Section 5.3.

3.5.1 Linear Spatial Filters

The toolbox supports a number of predefined 2-D linear spatial filters, obtained by using function `fspecial`, which generates a filter mask, `w`, using the syntax

```
w = fspecial('type', parameters)
```

where `'type'` specifies the filter type, and `parameters` further define the specified filter. The spatial filters supported by `fspecial` are summarized in Table 3.4, including applicable parameters for each filter.

TABLE 3.4
Spatial filters supported by function `fspecial`.

Type	Syntax and Parameters
`'average'`	`fspecial('average', [r c])`. A rectangular averaging filter of size `r` × `c`. The default is 3 × 3. A single number instead of `[r c]` specifies a square filter.
`'disk'`	`fspecial('disk', r)`. A circular averaging filter (within a square of size `2r + 1`) with radius `r`. The default radius is 5.
`'gaussian'`	`fspecial('gaussian', [r c], sig)`. A Gaussian lowpass filter of size `r` × `c` and standard deviation `sig` (positive). The defaults are 3 × 3 and 0.5. A single number instead of `[r c]` specifies a square filter.
`'laplacian'`	`fspecial('laplacian', alpha)`. A 3 × 3 Laplacian filter whose shape is specified by `alpha`, a number in the range [0, 1]. The default value for `alpha` is 0.5.
`'log'`	`fspecial('log', [r c], sig)`. Laplacian of a Gaussian (LoG) filter of size `r` × `c` and standard deviation `sig` (positive). The defaults are 5 × 5 and 0.5. A single number instead of `[r c]` specifies a square filter.
`'motion'`	`fspecial('motion', len, theta)`. Outputs a filter that, when convolved with an image, approximates linear motion (of a camera with respect to the image) of `len` pixels. The direction of motion is `theta`, measured in degrees, counterclockwise from the horizontal. The defaults are 9 and 0, which represents a motion of 9 pixels in the horizontal direction.
`'prewitt'`	`fspecial('prewitt')`. Outputs a 3 × 3 Prewitt mask, `wv`, that approximates a vertical gradient. A mask for the horizontal gradient is obtained by transposing the result: `wh = wv'`.
`'sobel'`	`fspecial('sobel')`. Outputs a 3 × 3 Sobel mask, `sv`, that approximates a vertical gradient. A mask for the horizontal gradient is obtained by transposing the result: `sh = sv'`.
`'unsharp'`	`fspecial('unsharp', alpha)`. Outputs a 3 × 3 unsharp filter. Parameter `alpha` controls the shape; it must be greater than 0 and less than or equal to 1.0; the default is 0.2.

EXAMPLE 3.9: Using function `imfilter`.

■ We illustrate the use of `fspecial` and `imfilter` by enhancing an image with a Laplacian filter. The Laplacian of an image $f(x, y)$, denoted $\nabla^2 f(x, y)$, is defined as

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

Commonly used digital approximations of the second derivatives are

$$\frac{\partial^2 f}{\partial x^2} = f(x + 1, y) + f(x - 1, y) - 2f(x, y)$$

and

$$\frac{\partial^2 f}{\partial y^2} = f(x, y + 1) + f(x, y - 1) - 2f(x, y)$$

so that

$$\nabla^2 f = [f(x+1, y) + f(x-1, y) + f(x, y+1) + f(x, y-1)] - 4f(x, y)$$

This expression can be implemented at all points (x, y) in an image by convolving the image with the following spatial mask:

$$\begin{matrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{matrix}$$

An alternate definition of the digital second derivatives takes into account diagonal elements, and can be implemented using the mask

$$\begin{matrix} 1 & 1 & 1 \\ 1 & -8 & 1 \\ 1 & 1 & 1 \end{matrix}$$

Both derivatives sometimes are defined with the signs opposite to those shown here, resulting in masks that are the negatives of the preceding two masks.

Enhancement using the Laplacian is based on the equation

$$g(x, y) = f(x, y) + c[\nabla^2 f(x, y)]$$

where $f(x, y)$ is the input image, $g(x, y)$ is the enhanced image, and c is 1 if the center coefficient of the mask is positive, or -1 if it is negative (Gonzalez and Woods [2002]). Because the Laplacian is a derivative operator, it sharpens the image but drives constant areas to zero. Adding the original image back restores the gray-level tonality.

Function `fspecial('laplacian', alpha)` implements a more general Laplacian mask:

$$\begin{matrix} \frac{\alpha}{1+\alpha} & \frac{1-\alpha}{1+\alpha} & \frac{\alpha}{1+\alpha} \\ \frac{1-\alpha}{1+\alpha} & \frac{-4}{1+\alpha} & \frac{1-\alpha}{1+\alpha} \\ \frac{\alpha}{1+\alpha} & \frac{1-\alpha}{1+\alpha} & \frac{\alpha}{1+\alpha} \end{matrix}$$

which allows fine tuning of enhancement results. However, the predominant use of the Laplacian is based on the two masks just discussed.

We now proceed to enhance the image in Fig. 3.16(a) using the Laplacian. This image is a mildly blurred image of the North Pole of the moon. Enhancement in this case consists of sharpening the image, while preserving as much of its gray tonality as possible. First, we generate and display the Laplacian filter:

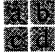


FIGURE 3.16 (a) Image of the North Pole of the moon. (b) Laplacian filtered image, using `uint8` formats. (c) Laplacian filtered image obtained using `double` formats. (d) Enhanced result, obtained by subtracting (c) from (a). (Original image courtesy of NASA.)

```
>> w = fspecial('laplacian', 0)
w =

    0.0000    1.0000    0.0000
    1.0000   -4.0000    1.0000
    0.0000    1.0000    0.0000
```

Note that the filter is of class `double`, and that its shape with `alpha = 0` is the Laplacian filter discussed previously. We could just as easily have specified this shape manually as

```
>> w = [0 1 0; 1 -4 1; 0 1 0];
```

Next we apply w to the input image, f, which is of class uint8:

```
>> g1 = imfilter(f, w, 'replicate');
>> imshow(g1, [ ])
```

Figure 3.16(b) shows the resulting image. This result looks reasonable, but has a problem: all its pixels are positive. Because of the negative center filter coefficient, we know that we can expect in general to have a Laplacian image with negative values. However, f in this case is of class uint8 and, as discussed in the previous section, filtering with imfilter gives an output that is of the same class as the input image, so negative values are truncated. We get around this difficulty by converting f to class double before filtering it:

```
>> f2 = im2double(f);
>> g2 = imfilter(f2, w, 'replicate');
>> imshow(g2, [ ])
```

The result, shown in Fig. 3.15(c), is more what a properly processed Laplacian image should look like.

Finally, we restore the gray tones lost by using the Laplacian by subtracting (because the center coefficient is negative) the Laplacian image from the original image:

```
>> g = f2 - g2;
>> imshow(g)
```

The result, shown in Fig. 3.16(d), is sharper than the original image. ■

EXAMPLE 3.10: Manually specifying filters and comparing enhancement techniques.

■ Enhancement problems often require the specification of filters beyond those available in the toolbox. The Laplacian is a good example. The toolbox supports a 3×3 Laplacian filter with a -4 in the center. Usually, sharper enhancement is obtained by using the 3×3 Laplacian filter that has a -8 in the center and is surrounded by 1s, as discussed earlier. The purpose of this example is to implement this filter manually, and also to compare the results obtained by using the two Laplacian formulations. The sequence of commands is as follows:

```
>>  f = imread('moon.tif');
>>  w4 = fspecial('laplacian', 0);  % Same as w in Example 3.9.
>>  w8 = [1 1 1; 1 -8 1; 1 1 1];
>>  f = im2double(f);
>>  g4 = f - imfilter(f, w4, 'replicate');
>>  g8 = f - imfilter(f, w8, 'replicate');
>>  imshow(f)
>>  figure, imshow(g4)
>>  figure, imshow(g8)
```

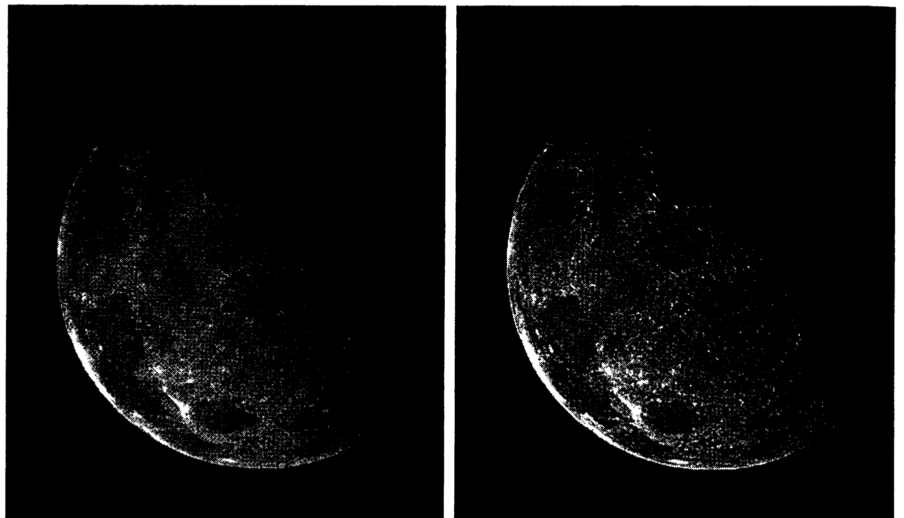


FIGURE 3.17 (a) Image of the North Pole of the moon. (b) Image enhanced using the Laplacian filter `'laplacian'`, which has a −4 in the center. (c) Image enhanced using a Laplacian filter with a −8 in the center.

Figure 3.17(a) shows the original moon image again for easy comparison. Fig. 3.17(b) is `g4`, which is the same as Fig. 3.16(d), and Fig. 3.17(c) shows `g8`. As expected, this result is significantly sharper than Fig. 3.17(b). ■

3.5.2 Nonlinear Spatial Filters

A commonly-used tool for generating nonlinear spatial filters in IPT is function `ordfilt2`, which generates *order-statistic filters* (also called *rank filters*). These are nonlinear spatial filters whose response is based on ordering (ranking) the pixels contained in an image neighborhood and then replacing the value of the center pixel in the neighborhood with the value determined by the

ranking result. Attention is focused in this section on nonlinear filters generated by `ordfilt2`. A number of additional nonlinear filters are developed and implemented in Section 5.3.

The syntax of function `ordfilt2` is

```
g = ordfilt2(f, order, domain)
```

This function creates the output image `g` by replacing each element of `f` by the `order`-th element in the sorted set of neighbors specified by the nonzero elements in `domain`. Here, `domain` is an $m \times n$ matrix of 1s and 0s that specify the pixel locations in the neighborhood that are to be used in the computation. In this sense, `domain` acts like a mask. The pixels in the neighborhood that correspond to 0 in the `domain` matrix are not used in the computation. For example, to implement a *min filter* (order 1) of size $m \times n$ we use the syntax

```
g = ordfilt2(f, 1, ones(m, n))
```

In this formulation the 1 denotes the 1st sample in the ordered set of mn samples, and `ones(m, n)` creates an $m \times n$ matrix of 1s, indicating that all samples in the neighborhood are to be used in the computation.

In the terminology of statistics, a min filter (the first sample of an ordered set) is referred to as the 0th percentile. Similarly, the 100th percentile is the last sample in the ordered set, which is the mnth sample. This corresponds to a *max filter*, which is implemented using the syntax

```
g = ordfilt2(f, m*n, ones(m, n))
```

The best-known order-statistic filter in digital image processing is the *median*† *filter*, which corresponds to the 50th percentile. We can use MATLAB function `median` in `ordfilt2` to create a median filter:

```
g = ordfilt2(f, median(1:m*n), ones(m, n))
```

where `median(1:m*n)` simply computes the median of the ordered sequence $1, 2, \ldots, mn$. Function `median` has the general syntax

```
v = median(A, dim)
```

where `v` is vector whose elements are the median of `A` along dimension `dim`. For example, if `dim = 1`, each element of `v` is the median of the elements along the corresponding column of `A`.

---

†Recall that the median, ξ, of a set of values is such that half the values in the set are less than or equal to ξ, and half are greater than or equal to ξ.

Because of its practical importance, the toolbox provides a specialized implementation of the 2-D median filter:

```
g = medfilt2(f, [m n], padopt)
```

where the tuple `[m n]` defines a neighborhood of size `m` × `n` over which the median is computed, and `padopt` specifies one of three possible border padding options: `'zeros'` (the default), `'symmetric'` in which `f` is extended symmetrically by mirror-reflecting it across its border, and `'indexed'`, in which `f` is padded with 1s if it is of class `double` and with 0s otherwise. The default form of this function is

```
g = medfilt2(f)
```

which uses a 3×3 neighborhood to compute the median, and pads the border of the input with 0s.

EXAMPLE 3.11: Median filtering with function `medfilt2`.

■ Median filtering is a useful tool for reducing salt-and-pepper noise in an image. Although we discuss noise reduction in much more detail in Chapter 5, it will be instructive at this point to illustrate briefly the implementation of median filtering.

The image in Fig. 3.18(a) is an X-ray image, `f`, of an industrial circuit board taken during automated inspection of the board. Figure 3.18(b) is the same image corrupted by salt-and-pepper noise in which both the black and white points have a probability of occurrence of 0.2. This image was generated using function `imnoise`, which is discussed in detail in Section 5.2.1:

```
>> fn = imnoise(f, 'salt & pepper', 0.2);
```

Figure 3.18(c) is the result of median filtering this noisy image, using the statement:

```
>> gm = medfilt2(fn);
```

Considering the level of noise in Fig. 3.18(b), median filtering using the default settings did a good job of noise reduction. Note, however, the black specks around the border. These were caused by the black points surrounding the image (recall that the default pads the border with 0s). This type of effect can often be reduced by using the `'symmetric'` option:

```
>> gms = medfilt2(fn, 'symmetric');
```

The result, shown in Fig. 3.18(d), is close to the result in Fig. 3.18(c), except that the black border effect is not as pronounced. ■

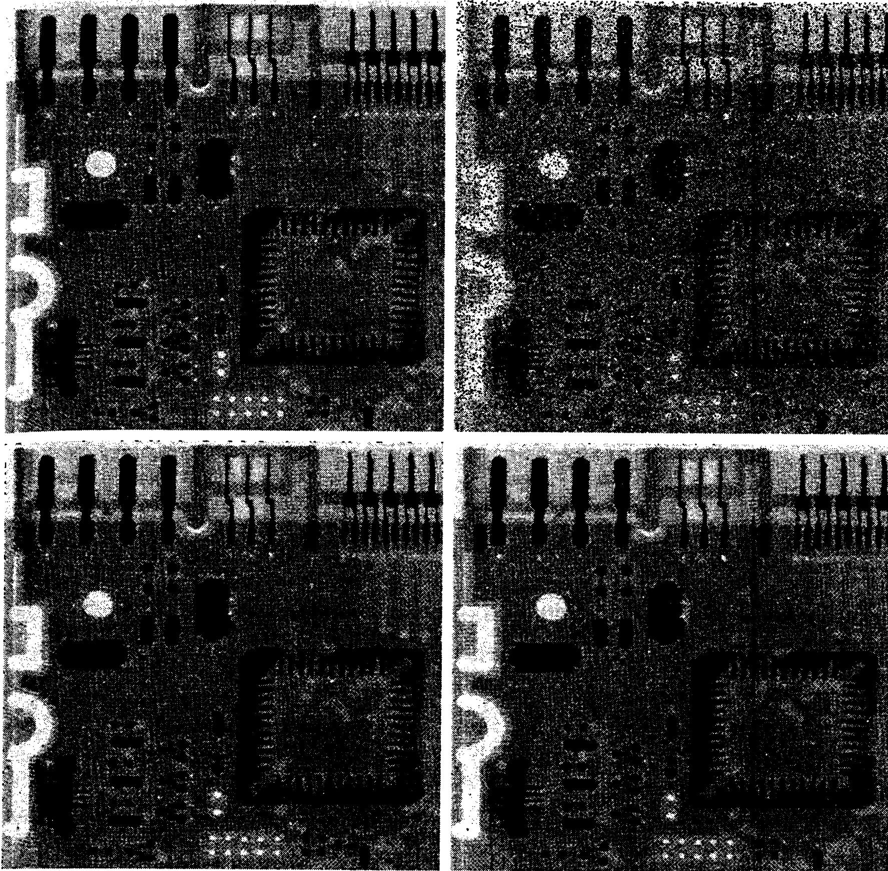


FIGURE 3.18 Median filtering, (a) X-ray image. (b) Image corrupted by salt-and-pepper noise. (c) Result of median filtering with `medfilt2` using the default settings. (d) Result of median filtering using the `'symmetric'` image extension option. Note the improvement in border behavior between (d) and (c). (Original image courtesy of Lixi, Inc.)

Summary

In addition to dealing with image enhancement, the material in this chapter is the foundation for numerous topics in subsequent chapters. For example, we will encounter spatial processing again in Chapter 5 in connection with image restoration, where we also take a closer look at noise reduction and noise-generating functions in MATLAB. Some of the spatial masks that were mentioned briefly here are used extensively in Chapter 10 for edge detection in segmentation applications. The concept of convolution and correlation is explained again in Chapter 4 from the perspective of the frequency domain. Conceptually, mask processing and the implementation of spatial filters will surface in various discussions throughout the book. In the process, we will extend the discussion begun here and introduce additional aspects of how spatial filters can be implemented efficiently in MATLAB.