

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

© 2009 by Gatesmark, LLC

All rights reserved. No part of this book may be reproduced or transmitted in any form or by any means, without written permission from the publisher.

Gatesmark Publishing[®] is a registered trademark of Gatesmark, LLC, www.gatesmark.com.

Gatesmark[®] is a registered trademark of Gatesmark, LLC, www.gatesmark.com.

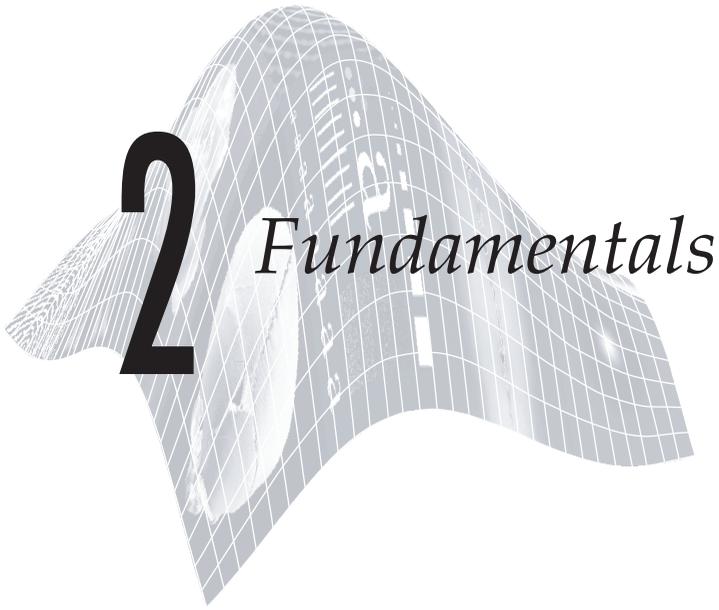
MATLAB[®] is a registered trademark of The MathWorks, Inc., 3 Apple Hill Drive, Natick, MA 01760-2098

The authors and publisher of this book have used their best efforts in preparing this book. These efforts include the development, research, and testing of the theories and programs to determine their effectiveness. The authors and publisher shall not be liable in any event for incidental or consequential damages with, or arising out of, the furnishing, performance, or use of these programs.

Printed in the United States of America

10 9 8 7 6 5 4 3 2 1

ISBN 978-0-9820854-0-0



2 Fundamentals

Preview

As mentioned in the previous chapter, the power that MATLAB brings to digital image processing is an extensive set of functions for processing multidimensional arrays of which images (two-dimensional numerical arrays) are a special case. The Image Processing Toolbox is a collection of functions that extend the capability of the MATLAB numeric computing environment. These functions, and the expressiveness of the MATLAB language, make image-processing operations easy to write in a compact, clear manner, thus providing an ideal software prototyping environment for the solution of image processing problems. In this chapter we introduce the basics of MATLAB notation, discuss a number of fundamental toolbox properties and functions, and begin a discussion of programming concepts. Thus, the material in this chapter is the foundation for most of the software-related discussions in the remainder of the book.

2.1 Digital Image Representation

An image may be defined as a two-dimensional function $f(x, y)$, where x and y are *spatial* (plane) *coordinates*, and the amplitude of f at any pair of coordinates is called the *intensity* of the image at that point. The term *gray level* is used often to refer to the intensity of monochrome images. Color images are formed by a combination of individual images. For example, in the RGB color system a color image consists of three individual monochrome images, referred to as the *red* (R), *green* (G), and *blue* (B) *primary* (or *component*) *images*. For this reason, many of the techniques developed for monochrome images can be extended to color images by processing the three component images individually. Color image processing is the topic of Chapter 7. An image may be continuous

with respect to the x - and y -coordinates, and also in amplitude. Converting such an image to digital form requires that the coordinates, as well as the amplitude, be digitized. Digitizing the coordinate values is called *sampling*; digitizing the amplitude values is called *quantization*. Thus, when x , y , and the amplitude values of f are all finite, discrete quantities, we call the image a *digital image*.

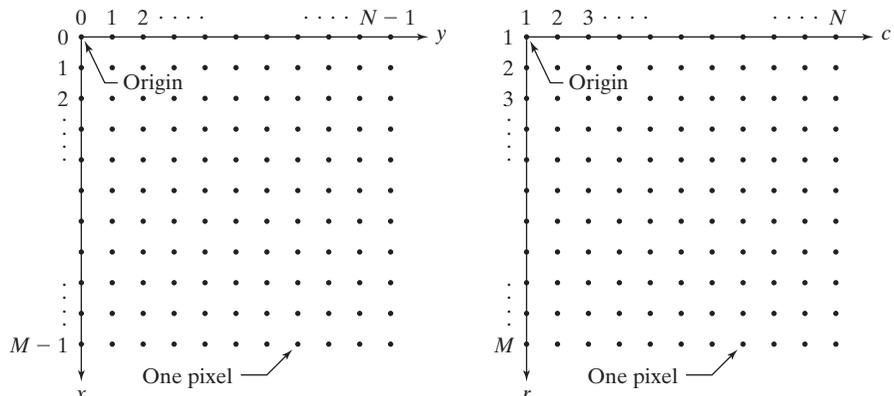
2.1.1 Coordinate Conventions

The result of sampling and quantization is a matrix of real numbers. We use two principal ways in this book to represent digital images. Assume that an image $f(x, y)$ is sampled so that the resulting image has M rows and N columns. We say that the image is of *size* $M \times N$. The values of the coordinates are discrete quantities. For notational clarity and convenience, we use integer values for these discrete coordinates. In many image processing books, the image origin is defined to be at $(x, y) = (0, 0)$. The next coordinate values along the first row of the image are $(x, y) = (0, 1)$. The notation $(0, 1)$ is used to signify the second sample along the first row. It *does not* mean that these are the *actual* values of physical coordinates when the image was sampled. Figure 2.1(a) shows this coordinate convention. Note that x ranges from 0 to $M - 1$ and y from 0 to $N - 1$ in integer increments.

The coordinate convention used in the Image Processing Toolbox to denote arrays is different from the preceding paragraph in two minor ways. First, instead of using (x, y) , the toolbox uses the notation (r, c) to indicate rows and columns. Note, however, that the order of coordinates is the same as the order discussed in the previous paragraph, in the sense that the first element of a coordinate tuple, (a, b) , refers to a row and the second to a column. The other difference is that the origin of the coordinate system is at $(r, c) = (1, 1)$; thus, r ranges from 1 to M , and c from 1 to N , in integer increments. Figure 2.1(b) illustrates this coordinate convention.

Image Processing Toolbox documentation refers to the coordinates in Fig. 2.1(b) as *pixel coordinates*. Less frequently, the toolbox also employs another coordinate convention, called *spatial coordinates*, that uses x to refer to columns and y to refer to rows. This is the opposite of our use of variables x and y . With

a b
FIGURE 2.1
 Coordinate conventions used (a) in many image processing books, and (b) in the Image Processing Toolbox.



a few exceptions, we do not use the toolbox's spatial coordinate convention in this book, but many MATLAB functions do, and you will definitely encounter it in toolbox and MATLAB documentation.

2.1.2 Images as Matrices

The coordinate system in Fig. 2.1(a) and the preceding discussion lead to the following representation for a digitized image:

$$f(x, y) = \begin{bmatrix} f(0,0) & f(0,1) & \cdots & f(0, N-1) \\ f(1,0) & f(1,1) & \cdots & f(1, N-1) \\ \vdots & \vdots & & \vdots \\ f(M-1,0) & f(M-1,1) & \cdots & f(M-1, N-1) \end{bmatrix}$$

The right side of this equation is a digital image by definition. Each element of this array is called an *image element*, *picture element*, *pixel*, or *pel*. The terms *image* and *pixel* are used throughout the rest of our discussions to denote a digital image and its elements.

A digital image can be represented as a MATLAB matrix:

$$f = \begin{bmatrix} f(1, 1) & f(1, 2) & \cdots & f(1, N) \\ f(2, 1) & f(2, 2) & \cdots & f(2, N) \\ \vdots & \vdots & & \vdots \\ f(M, 1) & f(M, 2) & \cdots & f(M, N) \end{bmatrix}$$

MATLAB documentation uses the terms *matrix* and *array* interchangeably. However, keep in mind that a matrix is two dimensional, whereas an array can have any finite dimension.

where $f(1, 1) = f(0,0)$ (note the use of a monospace font to denote MATLAB quantities). Clearly, the two representations are identical, except for the shift in origin. The notation $f(p, q)$ denotes the element located in row p and column q . For example, $f(6, 2)$ is the element in the sixth row and second column of matrix f . Typically, we use the letters M and N , respectively, to denote the number of rows and columns in a matrix. A $1 \times N$ matrix is called a *row vector*, whereas an $M \times 1$ matrix is called a *column vector*. A 1×1 matrix is a *scalar*.

Matrices in MATLAB are stored in variables with names such as A , a , RGB , $real_array$, and so on. Variables must begin with a letter and contain only letters, numerals, and underscores. As noted in the previous paragraph, all MATLAB quantities in this book are written using monospace characters. We use conventional Roman, italic notation, such as $f(x, y)$, for mathematical expressions.

2.2 Reading Images

Images are read into the MATLAB environment using function `imread`, whose basic syntax is

```
imread('filename')
```

Recall from Section 1.6 that we use margin icons to highlight the first use of a MATLAB or toolbox function.



Here, `filename` is a string containing the complete name of the image file (including any applicable extension). For example, the statement

```
>> f = imread('chestxray.jpg');
```

reads the image from the JPEG file `chestxray` into image array `f`. Note the use of single quotes (`'`) to delimit the string `filename`. The semicolon at the end of a statement is used by MATLAB for *suppressing* output. If a semicolon is not included, MATLAB displays on the screen the results of the operation(s) specified in that line. The prompt symbol (`>>`) designates the beginning of a command line, as it appears in the MATLAB Command Window (see Fig. 1.1).

When, as in the preceding command line, no path information is included in `filename`, `imread` reads the file from the Current Directory and, if that fails, it tries to find the file in the MATLAB search path (see Section 1.7). The simplest way to read an image from a specified directory is to include a full or relative path to that directory in `filename`. For example,

```
>> f = imread('D:\myimages\chestxray.jpg');
```

reads the image from a directory called `myimages` in the D: drive, whereas

```
>> f = imread('..\myimages\chestxray.jpg');
```

reads the image from the `myimages` subdirectory of the current working directory. The MATLAB Desktop displays the path to the Current Directory on the toolbar, which provides an easy way to change it. Table 2.1 lists some of the most popular image/graphics formats supported by `imread` and `imwrite` (`imwrite` is discussed in Section 2.4).

Typing `size` at the prompt gives the row and column dimensions of an image:

```
>> size(f)
ans =
    1024    1024
```

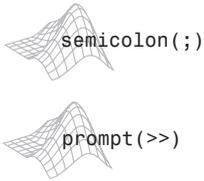
More generally, for an array `A` having an arbitrary number of dimensions, a statement of the form

$$[D1, D2, \dots, DK] = \text{size}(A)$$

returns the sizes of the first `K` dimensions of `A`. This function is particularly useful in programming to determine automatically the size of a 2-D image:

```
>> [M, N] = size(f);
```

This syntax returns the number of rows (`M`) and columns (`N`) in the image. Similarly, the command



In Windows, directories are called *folders*.



Format Name	Description	Recognized Extensions
BMP [†]	Windows Bitmap	.bmp
CUR	Windows Cursor Resources	.cur
FITS [†]	Flexible Image Transport System	.fts, .fits
GIF	Graphics Interchange Format	.gif
HDF	Hierarchical Data Format	.hdf
ICO [†]	Windows Icon Resources	.ico
JPEG	Joint Photographic Experts Group	.jpg, .jpeg
JPEG 2000 [†]	Joint Photographic Experts Group	.jp2, .jpf, .jpx, j2c, j2k
PBM	Portable Bitmap	.pbm
PGM	Portable Graymap	.pgm
PNG	Portable Network Graphics	.png
PNM	Portable Any Map	.pnm
RAS	Sun Raster	.ras
TIFF	Tagged Image File Format	.tif, .tiff
XWD	X Window Dump	.xwd

[†]Supported by `imread`, but not by `imwrite`

TABLE 2.1
Some of the image/graphics formats supported by `imread` and `imwrite`, starting with MATLAB 7.6. Earlier versions support a subset of these formats. See the MATLAB documentation for a complete list of supported formats.

```
>> M = size(f, 1);
```

gives the size of `f` along its first dimension, which is defined by MATLAB as the vertical dimension. That is, this command gives the number of rows in `f`. The second dimension of an array is in the horizontal direction, so the statement `size(f, 2)` gives the number of columns in `f`. A *singleton dimension* is any dimension, `dim`, for which `size(A, dim) = 1`.

The `whos` function displays additional information about an array. For instance, the statement

```
>> whos f
```

gives

Name	Size	Bytes	Class	Attributes
f	1024x1024	1048576	uint8	



Although not applicable in this example, attributes that might appear under `Attributes` include terms such as `global`, `complex`, and `sparse`.

The Workspace Browser in the MATLAB Desktop displays similar information. The `uint8` entry shown refers to one of several MATLAB data classes discussed in Section 2.5. A semicolon at the end of a `whos` line has no effect, so normally one is not used.

2.3 Displaying Images

Images are displayed on the MATLAB desktop using function `imshow`, which has the basic syntax:



Function `imshow` has a number of other syntax forms for performing tasks such as controlling image magnification. Consult the help page for `imshow` for additional details.

```
imshow(f)
```

where `f` is an image array. Using the syntax

```
imshow(f, [low high])
```

displays as black all values less than or equal to `low`, and as white all values greater than or equal to `high`. The values in between are displayed as intermediate intensity values. Finally, the syntax

```
imshow(f, [ ])
```

sets variable `low` to the minimum value of array `f` and `high` to its maximum value. This form of `imshow` is useful for displaying images that have a low dynamic range or that have positive and negative values.

EXAMPLE 2.1:
Reading and displaying images.

■ The following statements read from disk an image called `rose_512.tif`, extract information about the image, and display it using `imshow`:

```
>> f = imread('rose_512.tif');
>> whos f
```

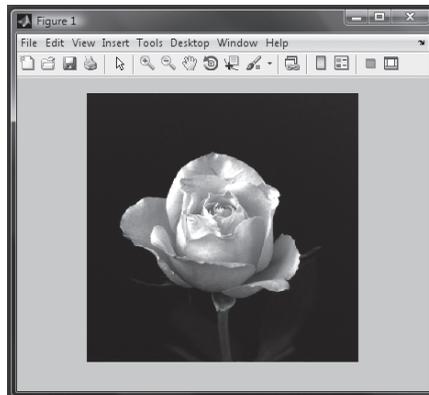
Name	Size	Bytes	Class	Attributes
f	512x512	262144	uint8 array	

```
>> imshow(f)
```

A semicolon at the end of an `imshow` line has no effect, so normally one is not used. Figure 2.2 shows what the output looks like on the screen. The figure

FIGURE 2.2

Screen capture showing how an image appears on the MATLAB desktop. Note the figure number on the top, left of the window. In most of the examples throughout the book, only the images themselves are shown.



number appears on the top, left of the window. Note the various pull-down menus and utility buttons. They are used for processes such as scaling, saving, and exporting the contents of the display window. In particular, the **Edit** menu has functions for editing and formatting the contents before they are printed or saved to disk.

If another image, *g*, is displayed using `imshow`, MATLAB replaces the image in the figure window with the new image. To keep the first image and output a second image, use function `figure`, as follows:

```
>> figure, imshow(g)
```

Using the statement

```
>> imshow(f), figure, imshow(g)
```

displays both images. Note that more than one command can be written on a line, provided that different commands are delimited by commas or semicolons. As mentioned earlier, a semicolon is used whenever it is desired to suppress screen outputs from a command line.

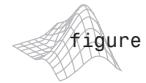
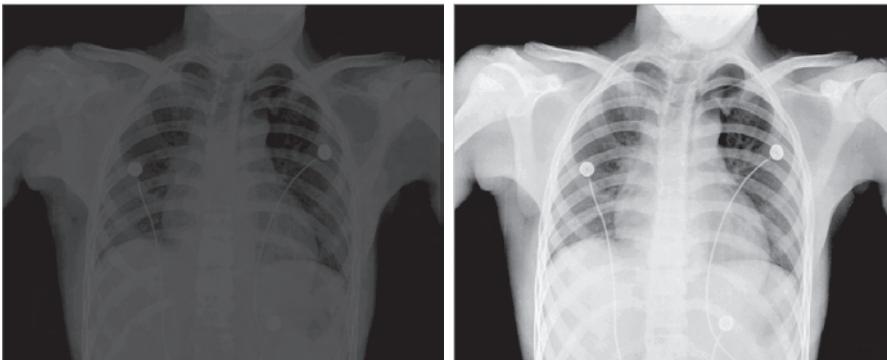
Finally, suppose that we have just read an image, *h*, and find that using `imshow(h)` produces the image in Fig. 2.3(a). This image has a low dynamic range, a condition that can be remedied for display purposes by using the statement

```
>> imshow(h, [ ])
```

Figure 2.3(b) shows the result. The improvement is apparent. ■

The *Image Tool* in the Image Processing Toolbox provides a more interactive environment for viewing and navigating within images, displaying detailed information about pixel values, measuring distances, and other useful operations. To start the Image Tool, use the `imtool` function. For example, the following statements read an image from a file and then display it using `imtool`:

```
>> f = imread('rose_1024.tif');
>> imtool(f)
```



Function `figure` creates a figure window. When used without an argument, as shown here, it simply creates a *new* figure window. Typing `figure(n)` forces figure number *n* to become visible.



a b
FIGURE 2.3 (a) An image, *h*, with low dynamic range. (b) Result of scaling by using `imshow(h, [])`. (Original image courtesy of Dr. David R. Pickens, Vanderbilt University Medical Center.)

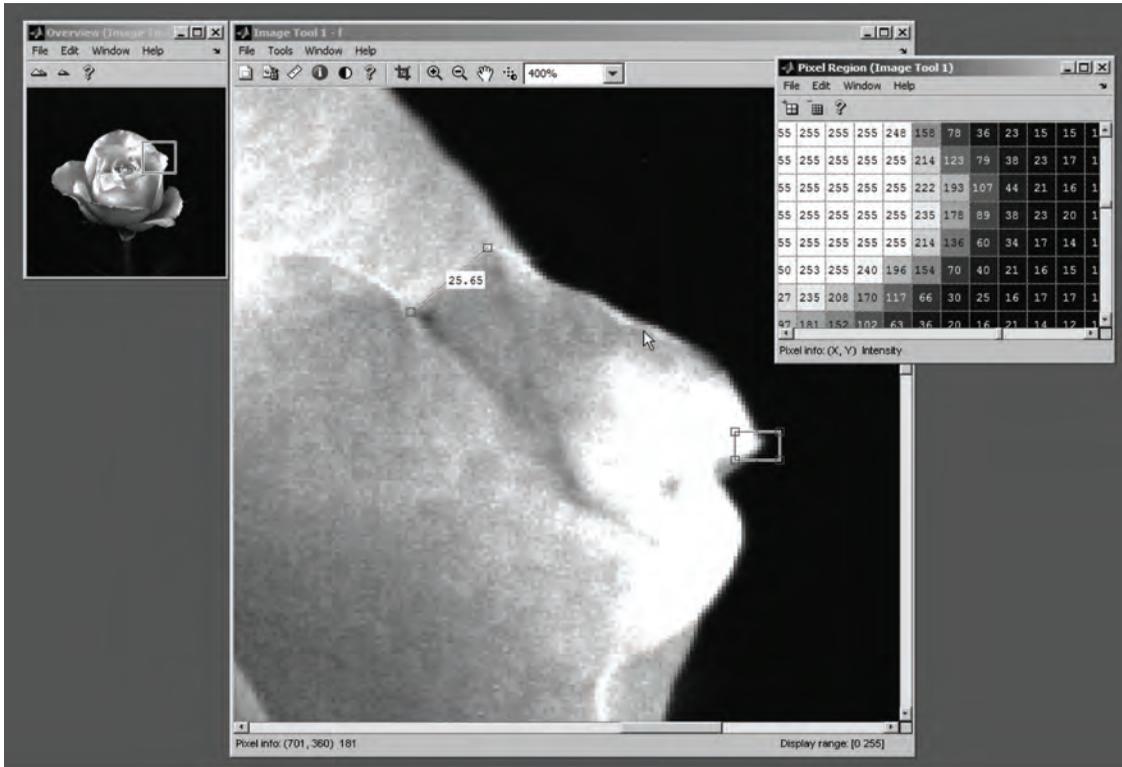


FIGURE 2.4 The Image Tool. The Overview Window, Main Window, and Pixel Region tools are shown.

Figure 2.4 shows some of the windows that might appear when using the Image Tool. The large, central window is the main view. In the figure, it is showing the image pixels at 400% magnification, meaning that each image pixel is rendered on a 4×4 block of screen pixels. The status text at the bottom of the main window shows the column/row location (701, 360) and value (181) of the pixel lying under the mouse cursor (the origin of the image is at the top, left). The *Measure Distance* tool is in use, showing that the distance between the two pixels enclosed by the small boxes is 25.65 units.

The *Overview Window*, on the left side of Fig. 2.4, shows the entire image in a thumbnail view. The *Main Window* view can be adjusted by dragging the rectangle in the Overview Window. The *Pixel Region Window* shows individual pixels from the small square region on the upper right tip of the rose, zoomed large enough to see the actual pixel values.

Table 2.2 summarizes the various tools and capabilities associated with the Image Tool. In addition to these tools, the Main and Overview Window toolbars provide controls for tasks such as image zooming, panning, and scrolling.

Tool	Description
Pixel Information	Displays information about the pixel under the mouse pointer.
Pixel Region	Superimposes pixel values on a zoomed-in pixel view.
Distance	Measures the distance between two pixels.
Image Information	Displays information about images and image files.
Adjust Contrast	Adjusts the contrast of the displayed image.
Crop Image	Defines a crop region and crops the image.
Display Range	Shows the display range of the image data.
Overview	Shows the currently visible image.

TABLE 2.2 Tools associated with the Image Tool.

2.4 Writing Images

Images are written to the Current Directory using function `imwrite`, which has the following basic syntax:

```
imwrite(f, 'filename')
```



With this syntax, the string contained in `filename` must include a recognized file format extension (see Table 2.1). For example, the following command writes `f` to a file called `patient10_run1.tif`:

```
>> imwrite(f, 'patient10_run1.tif')
```

Function `imwrite` writes the image as a TIFF file because it recognizes the `.tif` extension in the filename.

Alternatively, the desired format can be specified explicitly with a third input argument. This syntax is useful when the desired file does not use one of the recognized file extensions. For example, the following command writes `f` to a TIFF file called `patient10.run1`:

```
>> imwrite(f, 'patient10.run1', 'tif')
```

Function `imwrite` can have other parameters, depending on the file format selected. Most of the work in the following chapters deals either with JPEG or TIFF images, so we focus attention here on these two formats. A more general `imwrite` syntax applicable only to JPEG images is

```
imwrite(f, 'filename.jpg', 'quality', q)
```

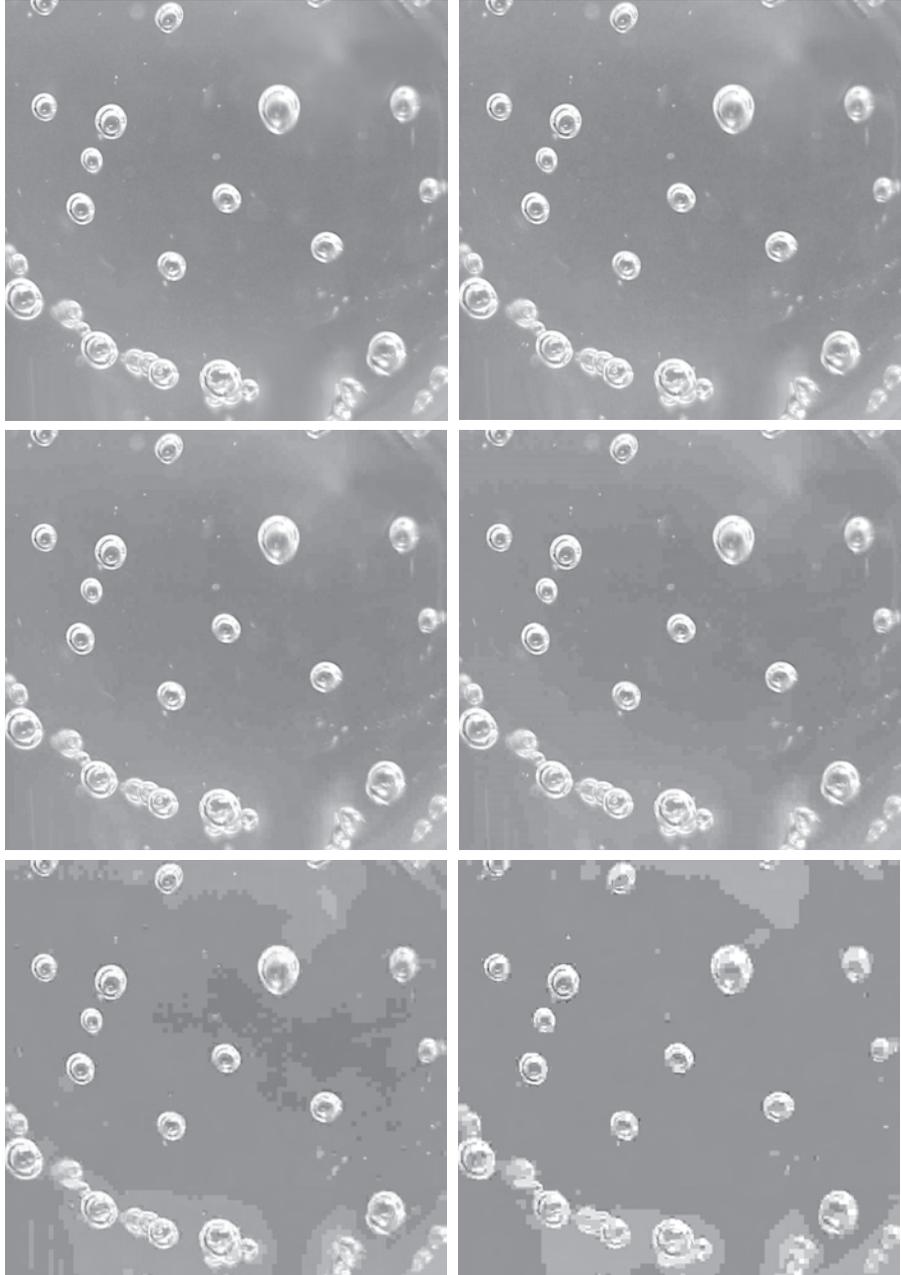
where `q` is an integer between 0 and 100 (the lower the number the higher the degradation due to JPEG compression).

EXAMPLE 2.2:
 Writing an image
 and using
 function `imfinfo`.

■ Figure 2.5(a) shows an image, f , typical of sequences of images resulting from a given chemical process. It is desired to transmit these images on a routine basis to a central site for visual and/or automated inspection. In order to reduce storage requirements and transmission time, it is important that the images be compressed as much as possible, while not degrading their visual

a	b
c	d
e	f

FIGURE 2.5
 (a) Original image.
 (b) through (f)
 Results of using
 jpg quality values
 $q = 50, 25, 15, 5,$
 and 0 , respectively.
 False contouring
 begins to be
 noticeable for
 $q = 15$ [image (d)]
 and is quite
 visible for $q = 5$
 and $q = 0$.



See Example 2.11 for
 a function that creates
 all the images in Fig. 2.5
 using a loop.

appearance beyond a reasonable level. In this case “reasonable” means no perceptible *false contouring*. Figures 2.5(b) through (f) show the results obtained by writing image `f` to disk (in JPEG format), with `q = 50, 25, 15, 5, and 0`, respectively. For example, the applicable syntax for `q = 25` is

```
>> imwrite(f, 'bubbles25.jpg', 'quality', 25)
```

The image for `q = 15` [Fig. 2.5(d)] has false contouring that is barely visible, but this effect becomes quite pronounced for `q = 5` and `q = 0`. Thus, an acceptable solution with some margin for error is to compress the images with `q = 25`. In order to get an idea of the compression achieved and to obtain other image file details, we can use function `imfinfo`, which has the syntax

```
imfinfo filename
```

where `filename` is the file name of the image stored on disk. For example,

```
>> imfinfo bubbles25.jpg
```

outputs the following information (note that some fields contain no information in this case):

```

      Filename: 'bubbles25.jpg'
      FileModDate: '04-Jan-2003 12:31:26'
      FileSize: 13849
      Format: 'jpg'
      FormatVersion: ''
      Width: 714
      Height: 682
      BitDepth: 8
      ColorType: 'grayscale'
      FormatSignature: ''
      Comment: {}

```

where `FileSize` is in bytes. The number of bytes in the original image is computed by multiplying `Width` by `Height` by `BitDepth` and dividing the result by 8. The result is 486948. Dividing this by `FileSize` gives the compression ratio: $(486948/13849) = 35.16$. This compression ratio was achieved while maintaining image quality consistent with the requirements of the application. In addition to the obvious advantages in storage space, this reduction allows the transmission of approximately 35 times the amount of uncompressed data per unit time.

The information fields displayed by `imfinfo` can be captured into a so-called *structure variable* that can be used for subsequent computations. Using the preceding image as an example, and letting `K` denote the structure variable, we use the syntax

```
>> K = imfinfo('bubbles25.jpg');
```

to store into variable `K` all the information generated by command `imfinfo`.



Recent versions of MATLAB may show more information in the output of `imfinfo`, particularly for images captured using digital cameras.

Structures are discussed in Section 2.10.7.

The information generated by `imfinfo` is appended to the structure variable by means of *fields*, separated from `K` by a dot. For example, the image height and width are now stored in structure fields `K.Height` and `K.Width`. As an illustration, consider the following use of structure variable `K` to compute the compression ratio for `bubbles25.jpg`:

```
>> K = imfinfo('bubbles25.jpg');
>> image_bytes = K.Width*K.Height*K.BitDepth/8;
>> compressed_bytes = K.FileSize;
>> compression_ratio = image_bytes/compressed_bytes

compression_ratio =
    35.1612
```

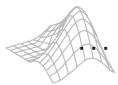
Note that `imfinfo` was used in two different ways. The first was to type `imfinfo bubbles25.jpg` at the prompt, which resulted in the information being displayed on the screen. The second was to type `K=imfinfo('bubbles25.jpg')`, which resulted in the information generated by `imfinfo` being stored in `K`. These two different ways of calling `imfinfo` are an example of *command-function duality*, an important concept that is explained in more detail in the MATLAB documentation. ■

To learn more about command function duality, consult the help page on this topic. (See Section 1.7.2 regarding help pages.)

A more general `imwrite` syntax applicable only to `tif` images has the form

```
imwrite(g, 'filename.tif', 'compression', 'parameter', ...
        'resolution', [colres rowres])
```

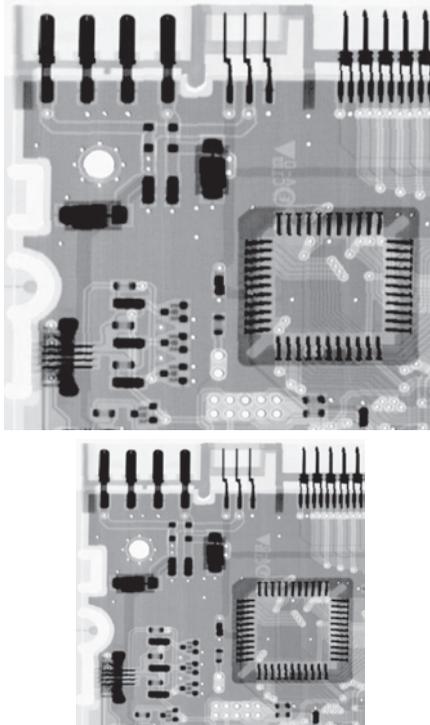
where 'parameter' can have one of the following principal values: 'none' indicates no compression; 'packbits' (the default for nonbinary images), 'lzw', 'deflate', 'jpeg', 'ccitt' (binary images only; the default), 'fax3' (binary images only), and 'fax4'. The 1×2 array `[colres rowres]` contains two integers that give the column resolution and row resolution in dots-per-unit (the default values are `[72 72]`). For example, if the image dimensions are in inches, `colres` is the number of dots (pixels) per inch (dpi) in the vertical direction, and similarly for `rowres` in the horizontal direction. Specifying the resolution by a single scalar, `res`, is equivalent to writing `[res res]`. As you will see in the following example, the TIFF resolution parameter can be used to modify the size of an image in printed documents.



If a statement does not fit on one line, use an ellipsis (three periods), followed by **Return** or **Enter**, to indicate that the statement continues on the next line. There are no spaces between the periods.

EXAMPLE 2.3: Using `imwrite` parameters.

■ Figure 2.6(a) is an 8-bit X-ray image, `f`, of a circuit board generated during quality inspection. It is in `jpg` format, at 200 dpi. The image is of size 450×450 pixels, so its printed dimensions are 2.25×2.25 inches. We want to store this image in `tif` format, with no compression, under the name `sf`. In addition, we want to reduce the printed size of the image to 1.5×1.5 inches while keeping the pixel count at 450×450 . The following statement gives the desired result:



a
b

FIGURE 2.6

Effects of changing the dpi resolution while keeping the number of pixels constant. (a) A 450×450 image at 200 dpi (size = 2.25×2.25 inches). (b) The same image, but at 300 dpi (size = 1.5×1.5 inches). (Original image courtesy of Lixi, Inc.)

```
>> imwrite(f, 'sf.tif', 'compression', 'none', 'resolution', [300 300])
```

The values of the vector `[colres rowres]` were determined by multiplying 200 dpi by the ratio $2.25/1.5$ which gives 300 dpi. Rather than do the computation manually, we could write

```
>> res = round(200*2.25/1.5);
>> imwrite(f, 'sf.tif', 'compression', 'none', 'resolution', res)
```



where function `round` rounds its argument to the nearest integer. It is important to note that the number of pixels was not changed by these commands. Only the printed size of the image changed. The original 450×450 image at 200 dpi is of size 2.25×2.25 inches. The new 300-dpi image [Fig. 2.6(b)] is identical, except that its 450×450 pixels are distributed over a 1.5×1.5 -inch area. Processes such as this are useful for controlling the size of an image in a printed document without sacrificing resolution. ■

Sometimes, it is necessary to export images and plots to disk the way they appear on the MATLAB desktop. The *contents* of a figure window can be exported to disk in two ways. The first is to use the **File** pull-down menu in the figure window (see Fig. 2.2) and then choose **Save As**. With this option, the



user can select a location, file name, and format. More control over export parameters is obtained by using the `print` command:

```
print -fno -dfileformat -rresno filename
```

where *no* refers to the figure number in the figure window of interest, *fileformat* refers to one of the file formats in Table 2.1, *resno* is the resolution in dpi, and *filename* is the name we wish to assign the file. For example, to export the contents of the figure window in Fig. 2.2 as a `tif` file at 300 dpi, and under the name `hi_res_rose`, we would type

```
>> print -f1 -dtiff -r300 hi_res_rose
```

This command sends the file `hi_res_rose.tif` to the Current Directory. If we type `print` at the prompt, MATLAB prints (to the default printer) the contents of the last figure window displayed. It is possible also to specify other options with `print`, such as a specific printing device.

2.5 Classes

Although we work with integer coordinates, the values (intensities) of pixels are not restricted to be integers in MATLAB. Table 2.3 lists the various *classes* supported by MATLAB and the Image Processing Toolbox[†] for representing pixel values. The first eight entries in the table are referred to as *numeric* class-

TABLE 2.3

Classes used for image processing in MATLAB. The first eight entries are referred to as *numeric* classes, the ninth entry is the *char* class, and the last entry is the *logical* class.

Name	Description
<code>double</code>	Double-precision, floating-point numbers in the approximate range $\pm 10^{308}$ (8 bytes per element).
<code>single</code>	Single-precision floating-point numbers with values in the approximate range $\pm 10^{38}$ (4 bytes per element).
<code>uint8</code>	Unsigned 8-bit integers in the range [0, 255] (1 byte per element).
<code>uint16</code>	Unsigned 16-bit integers in the range [0, 65535] (2 bytes per element).
<code>uint32</code>	Unsigned 32-bit integers in the range [0, 4294967295] (4 bytes per element).
<code>int8</code>	Signed 8-bit integers in the range [-128, 127] (1 byte per element).
<code>int16</code>	Signed 16-bit integers in the range [-32768, 32767] (2 bytes per element).
<code>int32</code>	Signed 32-bit integers in the range [-2147483648, 2147483647] (4 bytes per element).
<code>char</code>	Characters (2 bytes per element).
<code>logical</code>	Values are 0 or 1 (1 byte per element).

[†]MATLAB supports two other numeric classes not listed in Table 2.3, `uint64` and `int64`. The toolbox does not support these classes, and MATLAB arithmetic support for them is limited.

es. The ninth entry is the *char* (character) class and, as shown, the last entry is the *logical* class.

Classes `uint8` and `logical` are used extensively in image processing, and they are the usual classes encountered when reading images from image file formats such as TIFF or JPEG. These classes use 1 byte to represent each pixel. Some scientific data sources, such as medical imagery, require more dynamic range than is provided by `uint8`, so the `uint16` and `int16` classes are used often for such data. These classes use 2 bytes for each array element. The floating-point classes `double` and `single` are used for computationally intensive operations such as the Fourier transform (see Chapter 4). Double-precision floating-point uses 8 bytes per array element, whereas single-precision floating-point uses 4 bytes. The `int8`, `uint32`, and `int32` classes, although supported by the toolbox, are not used commonly for image processing.

2.6 Image Types

The toolbox supports four types of images:

- Gray-scale images
- Binary images
- Indexed images
- RGB images

Most monochrome image processing operations are carried out using binary or gray-scale images, so our initial focus is on these two image types. Indexed and RGB color images are discussed in Chapter 7.

Gray-scale images are referred to as *intensity images* in earlier versions of the toolbox. In the book, we use the two terms interchangeably when working with monochrome images.

2.6.1 Gray-scale Images

A *gray-scale image* is a data matrix whose values represent shades of gray. When the elements of a gray-scale image are of class `uint8` or `uint16`, they have integer values in the range $[0, 255]$ or $[0, 65535]$, respectively. If the image is of class `double` or `single`, the values are floating-point numbers (see the first two entries in Table 2.3). Values of `double` and `single` gray-scale images normally are scaled in the range $[0, 1]$, although other ranges can be used.

2.6.2 Binary Images

Binary images have a very specific meaning in MATLAB. A *binary image* is a *logical* array of 0s and 1s. Thus, an array of 0s and 1s whose values are of data class, say, `uint8`, is not considered a binary image in MATLAB. A numeric array is converted to binary using function `logical`. Thus, if `A` is a numeric array consisting of 0s and 1s, we create a logical array `B` using the statement

$$B = \text{logical}(A)$$


If `A` contains elements other than 0s and 1s, the `logical` function converts all nonzero quantities to logical 1s and all entries with value 0 to logical 0s. Using relational and logical operators (see Section 2.10.2) also results in logical arrays.



See Table 2.9 for a list of other functions based on the `is...` construct.

To test if an array is of class `logical` we use the `islogical` function:

```
islogical(C)
```

If `C` is a logical array, this function returns a 1. Otherwise it returns a 0. Logical arrays can be converted to numeric arrays using the class conversion functions discussed in Section 2.7.

2.6.3 A Note on Terminology

Considerable care was taken in the previous two sections to clarify the use of the terms *class* and *image type*. In general, we refer to an image as being a “class `image_type` image,” where `class` is one of the entries from Table 2.3, and `image_type` is one of the image types defined at the beginning of this section. Thus, an image is characterized by *both* a class *and* a type. For instance, a statement discussing an “`uint8` gray-scale image” is simply referring to a gray-scale image whose pixels are of class `uint8`. Some functions in the toolbox support all the data classes listed in Table 2.3, while others are very specific as to what constitutes a valid class.

2.7 Converting between Classes

Converting images from one class to another is a common operation. When converting between classes, keep in mind the value ranges of the classes being converted (see Table 2.3).

The general syntax for class conversion is

```
B = class_name(A)
```

where `class_name` is one of the names in the first column of Table 2.3. For example, suppose that `A` is an array of class `uint8`. A double-precision array, `B`, is generated by the command `B = double(A)`. If `C` is an array of class `double` in which all values are in the range `[0, 255]` (but possibly containing fractional values), it can be converted to an `uint8` array with the command `D = uint8(C)`. If an array of class `double` has any values outside the range `[0, 255]` and it is converted to class `uint8` in the manner just described, MATLAB converts to 0 all values that are less than 0, and converts to 255 all values that are greater than 255. Numbers in between are rounded to the nearest integer. Thus, proper scaling of a `double` array so that its elements are in the range `[0, 255]` is necessary before converting it to `uint8`. As indicated in Section 2.6.2, converting any of the numeric data classes to `logical` creates an array with logical 1s in locations where the input array has nonzero values, and logical 0s in places where the input array contains 0s.

The toolbox provides specific functions (Table 2.4) that perform the scaling and other bookkeeping necessary to convert images from one class to another. Function `im2uint8`, for example, creates a `uint8` image after detecting the

To simplify terminology, statements referring to values of class `double` are applicable also to the `single` class, unless stated otherwise. Both refer to floating point numbers, the only difference between them being precision and the number of bytes needed for storage.

Name	Converts Input to:	Valid Input Image Data Classes
im2uint8	uint8	logical, uint8, uint16, int16, single, and double
im2uint16	uint16	logical, uint8, uint16, int16, single, and double
im2double	double	logical, uint8, uint16, int16, single, and double
im2single	single	logical, uint8, uint16, int16, single, and double
mat2gray	double in the range [0, 1]	logical, uint8, int8, uint16, int16, uint32, int32, single, and double
im2bw	logical	uint8, uint16, int16, single, and double

TABLE 2.4

Toolbox functions for converting images from one class to another.

data class of the input and performing all the necessary scaling for the toolbox to recognize the data as valid image data. For example, consider the following image `f` of class `double`, which could be the result of an intermediate computation:

```
f =
    -0.5    0.5
     0.75   1.5
```

Performing the conversion

```
>> g = im2uint8(f)
```

yields the result

```
g =
     0   128
    191  255
```

from which we see that function `im2uint8` sets to 0 all values in the input that are less than 0, sets to 255 all values in the input that are greater than 1, and multiplies all other values by 255. Rounding the results of the multiplication to the nearest integer completes the conversion.

Function `im2double` converts an input to class `double`. If the input is of class `uint8`, `uint16`, or `logical`, function `im2double` converts it to class `double` with values in the range [0, 1]. If the input is of class `single`, or is already of class `double`, `im2double` returns an array that is of class `double`, but is numerically equal to the input. For example, if an array of class `double` results from computations that yield values outside the range [0, 1], inputting this array into



`im2double` will have no effect. As explained below, function `mat2gray` can be used to convert an array of any of the classes in Table 2.4 to a double array with values in the range $[0, 1]$.

As an illustration, consider the class `uint8` image

```
>> h = uint8([25 50; 128 200]);
```

Performing the conversion

```
>> g = im2double(h)
```

yields the result

```
g =
    0.0980    0.1961
    0.4706    0.7843
```

from which we infer that the conversion when the input is of class `uint8` is done simply by dividing each value of the input array by 255. If the input is of class `uint16` the division is by 65535.

Toolbox function `mat2gray` converts an image of any of the classes in Table 2.4 to an array of class `double` *scaled* to the range $[0, 1]$. The calling syntax is

$$g = \text{mat2gray}(A, [A_{\min}, A_{\max}])$$

where image `g` has values in the range 0 (black) to 1 (white). The specified parameters, `Amin` and `Amax`, are such that values less than `Amin` in `A` become 0 in `g`, and values greater than `Amax` in `A` correspond to 1 in `g`. The syntax

$$g = \text{mat2gray}(A)$$

sets the values of `Amin` and `Amax` to the actual minimum and maximum values in `A`. The second syntax of `mat2gray` is a very useful tool because it scales the entire range of values in the input to the range $[0, 1]$, independently of the class of the input, thus eliminating clipping.

Finally, we consider conversion to class `logical`. (Recall that the Image Processing Toolbox treats logical matrices as binary images.) Function `logical` converts an input array to a logical array. In the process, nonzero elements in the input are converted to 1s, and 0s are converted to 0s in the output. An alternative conversion procedure that often is more useful is to use a relational operator, such as `>`, with a threshold value. For example, the syntax

$$g = f > T$$

produces a logical matrix containing 1s wherever the elements of `f` are greater than `T` and 0s elsewhere.

Toolbox function `im2bw` performs this thresholding operation in a way that automatically scales the specified threshold in different ways, depending on the class of the input image. The syntax is

Section 2.8.2 explains the use of square brackets and semicolons to specify matrices.



See Section 2.10.2 regarding logical and relational operators.

```
g = im2bw(f, T)
```



Values specified for the threshold T must be in the range $[0, 1]$, regardless of the class of the input. The function automatically scales the threshold value according to the input image class. For example, if f is `uint8` and T is `0.4`, then `im2bw` thresholds the pixels in f by comparing them to $255 * 0.4 = 102$.

■ We wish to convert the following small, `double` image

```
>> f = [1 2; 3 4]
f =
     1     2
     3     4
```

EXAMPLE 2.4:
Converting
between image
classes.

to binary, such that values 1 and 2 become 0 and the other two values become 1. First we convert it to the range $[0, 1]$:

```
>> g = mat2gray(f)
g =
     0    0.3333
    0.6667    1.0000
```

Then we convert it to binary using a threshold, say, of value 0.6:

```
>> gb = im2bw(g, 0.6)
gb =
     0     0
     1     1
```

As mentioned earlier, we can generate a binary array directly using relational operators. Thus we get the same result by writing

```
>> gb = f > 2
gb =
     0     0
     1     1
```

Suppose now that we want to convert `gb` to a numerical array of 0s and 1s of class `double`. This is done directly:

```
>> gbd = im2double(gb)
gbd =
     0     0
     1     1
```

If `gb` had been of class `uint8`, applying `im2double` to it would have resulted in an array with values

```

        0        0
    0.0039  0.0039

```

because `im2double` would have divided all the elements by 255. This did not happen in the preceding conversion because `im2double` detected that the input was a `logical` array, whose only possible values are 0 and 1. If the input in fact had been of class `uint8` and we wanted to convert it to class `double` while keeping the 0 and 1 values, we would have converted the array by writing

```

>> gbd = double(gb)
gbd =
     0     0
     1     1

```

Finally, we point out that the output of one function can be passed directly as the input to another, so we could have started with image `f` and arrived at the same result by using the one-line statement

```
>> gbd = im2double(im2bw(mat2gray(f), 0.6));
```

or by using partial groupings of these functions. Of course, the entire process could have been done in this case with a simpler command:

```
>> gbd = double(f > 2);
```

demonstrating again the compactness of the MATLAB language. ■

As the first two entries in Table 2.3 show class numeric data of class `double` requires twice as much storage as data of class `single`. In most image processing applications in which numeric processing is used, `single` precision is perfectly adequate. Therefore, unless a specific application or a MATLAB or toolbox function requires class `double`, it is good practice to work with `single` data to conserve memory. A consistent programming pattern that you will see used throughout the book to change inputs to class `single` is as follows:

```

[fout, revertclass] = tofloat(f);
g = some_operation(fout)
g = revertclass(g);

```

Function `tofloat` (see Appendix C for the code) converts an input image `f` to floating-point. If `f` is a `double` or `single` image, then `fout` equals `f`. Otherwise, `fout` equals `im2single(f)`. Output `revertclass` can be used to convert back to the same class as `f`. In other words, the idea is to convert the input

Recall from Section 1.6 that we use the margin icon to denote the first use of a function developed in the book.

tofloat

See function `intrans` in Section 3.2.3 for an example of how `tofloat` is used.

image to `single`, perform operations using `single` precision, and then, if so desired, convert the final output image to the same class as the input. The valid image classes for `f` are those listed in the third column of the first four entries in Table 2.4: `logical`, `uint8`, `uint16`, `int16`, `double`, and `single`.

2.8 Array Indexing

MATLAB supports a number of powerful indexing schemes that simplify array manipulation and improve the efficiency of programs. In this section we discuss and illustrate basic indexing in one and two dimensions (i.e., vectors and matrices), as well as indexing techniques useful with binary images.

2.8.1 Indexing Vectors

As discussed in Section 2.1.2, an array of dimension $1 \times N$ is called a *row vector*. The elements of such a vector can be accessed using a single index value (also called a *subscript*). Thus, `v(1)` is the first element of vector `v`, `v(2)` is its second element, and so forth. Vectors can be formed in MATLAB by enclosing the elements, separated by spaces or commas, within square brackets. For example,

```
>> v = [1 3 5 7 9]
v =
    1  3  5  7  9
>> v(2)
ans =
    3
```

A row vector is converted to a column vector (and vice versa) using the *transpose operator* (`'`):

```
>> w = v.'
w =
    1
    3
    5
    7
    9
```



Using a single quote without the period computes the conjugate transpose. When the data are real, both transposes can be used interchangeably. See Table 2.5.

To access *blocks* of elements, we use MATLAB's *colon* notation. For example, to access the first three elements of `v` we write

```
>> v(1:3)
ans =
    1    3    5
```



Similarly, we can access the second through the fourth elements

```
>> v(2:4)
ans =
     3     5     7
```

or all the elements from, say, the third through the last element:

```
>> v(3:end)
ans =
     5     7     9
```

where end signifies the last element in the vector.

Indexing is not restricted to contiguous elements. For example,

```
>> v(1:2:end)
ans =
     1     5     9
```

The notation 1:2:end says to start at 1, count up by 2, and stop when the count reaches the last element. The steps can be negative:

```
>> v(end:-2:1)
ans =
     9     5     1
```

Here, the index count started at the last element, decreased by 2, and stopped when it reached the first element.

Function linspace, with syntax

```
x = linspace(a, b, n)
```

generates a row vector x of n elements linearly-spaced between, and including, a and b . We use this function in several places in later chapters. A vector can even be used as an index into another vector. For example, we can select the first, fourth, and fifth elements of v using the command

```
>> v([1 4 5])
ans =
     1     7     9
```

As we show in the following section, the ability to use a vector as an index into another vector also plays a key role in matrix indexing.

2.8.2 Indexing Matrices

Matrices can be represented conveniently in MATLAB as a sequence of row vectors enclosed by square brackets and separated by semicolons. For example, typing

```
>> A = [1 2 3; 4 5 6; 7 8 9]
```

gives the 3×3 matrix

```
A =
     1     2     3
     4     5     6
     7     8     9
```

Note that the use of semicolons inside square brackets is different from their use mentioned earlier to suppress output or to write multiple commands in a single line. We select elements in a matrix just as we did for vectors, but now we need two indices: one to establish a row location, and the other for the corresponding column. For example, to extract the element in the second row, third column of matrix A, we write

```
>> A(2, 3)
ans =
     6
```

A submatrix of A can be extracted by specifying a vector of values for both the row and the column indices. For example, the following statement extracts the submatrix of A containing rows 1 and 2 and columns 1, 2, and 3:

```
>> T2 = A([1 2], [1 2 3])
T2 =
     1     2     3
     4     5     6
```

Because the expression `1:K` creates a vector of integer values from 1 through K, the preceding statement could be written also as:

```
>> T2 = A(1:2, 1:3)
T2 =
     1     2     3
     4     5     6
```

The row and column indices do not have to be contiguous, nor do they have to be in ascending order. For example,

```
>> E = A([1 3], [3 2])
E =
     3     2
     9     8
```

The notation $A([a \ b], [c \ d])$ selects the elements in A with coordinates (a, c) , (a, d) , (b, c) , and (b, d) . Thus, when we let $E = A([1 \ 3], [3 \ 2])$, we are selecting the following elements in A : $A(1, 3)$, $A(1, 2)$, $A(3, 3)$, and $A(3, 2)$.

The row or column index can also be a single colon. A colon in the row index position is shorthand notation for selecting all rows. Similarly, a colon in the column index position selects all columns. For example, the following statement selects the entire 3rd column of A :

```
>> C3 = A(:, 3)
C3 =
     3
     6
     9
```

Similarly, this statement extracts the second row:

```
>> R2 = A(2, :)
R2 =
     4     5     6
```

Any of the preceding forms of indexing can be used on the left-hand side of an assignment statement. The next two statements create a copy, B , of matrix A , and then assign the value 0 to all elements in the 3rd column of B .

```
>> B = A;
>> B(:, 3) = 0
B =
     1     2     0
     4     5     0
     7     8     0
```

The keyword `end`, when it appears in the row index position, is shorthand notation for the last row. When `end` appears in the column index position, it indicates the last column. For example, the following statement finds the element in the last row and last column of A :

```
>> A(end, end)
ans =
     9
```

When used for indexing, the `end` keyword can be mixed with arithmetic operations, as well as with the colon operator. For example:

```
>> A(end, end - 2)
ans =
     7

>> A(2:end, end:-2:1)
ans =
     6     4
     9     7
```

2.8.3 Indexing with a Single Colon

The use of a single colon as an index into a matrix selects all the elements of the array and arranges them (in column order) into a single column vector. For example, with reference to matrix `T2` in the previous section,

```
>> v = T2(:)
v =
     1
     4
     2
     5
     3
     6
```

This use of the colon is helpful when, for example, we want to find the sum of all the elements of a matrix. One approach is to call function `sum` twice:

```
>> col_sums = sum(A)
col_sums =
    111    15    112
```



Function `sum` computes the sum of each column of `A`, storing the results into a row vector. Then we call `sum` again, passing it the vector of column sums:

```
>> total_sum = sum(col_sums)
total_sum =
    238
```

An easier procedure is to use single-colon indexing to convert *A* to a column vector, and pass the result to `sum`:

```
>> total_sum = sum(A(:))
total_sum =
    238
```

2.8.4 Logical Indexing

Another form of indexing that you will find quite useful is *logical indexing*. A logical indexing expression has the form `A(D)`, where *A* is an array and *D* is a logical array of the same size as *A*. The expression `A(D)` extracts all the elements of *A* corresponding to the 1-valued elements of *D*. For example,

```
>> D = logical([1 0 0; 0 0 1; 0 0 0])
D =
     1     0     0
     0     0     1
     0     0     0

>> A(D)
ans =
     1
     6
```

where *A* is as defined at the beginning of Section 2.8.2. The output of this method of logical indexing always is a column vector.

Logical indexing can be used also on the left-hand side of an assignment statement. For example, using the same *D* as above,

```
>> A(D) = [30 40]
A =
    30     2     3
     4     5    40
     7     8     9
```

In the preceding assignment, the number of elements on the right-hand side matched the number of 1-valued elements of *D*. Alternatively, the right-hand side can be a scalar, like this:

```
>> A(D) = 100
A =
```

```

100    2    3
   4    5   100
   7    8    9

```

Because binary images are represented as logical arrays, they can be used directly in logical indexing expressions to extract pixel values in an image that correspond to 1-valued pixels in a binary image. You will see numerous examples later in the book that use binary images and logical indexing.

2.8.5 Linear Indexing

The final category of indexing useful for image processing is *linear indexing*. A linear indexing expression is one that uses a *single* subscript to index a matrix or higher-dimensional array. To illustrate the concept we will use a 4×4 *Hilbert matrix* as an example:

```

>> H = hilb(4)
H =
    1.0000    0.5000    0.3333    0.2500
    0.5000    0.3333    0.2500    0.2000
    0.3333    0.2500    0.2000    0.1667
    0.2500    0.2000    0.1667    0.1429

```



`H([2 11])` is an example of a linear indexing expression:

```

>> H([2 11])
ans =
    0.5000    0.2000

```

To see how this type of indexing works, number the elements of `H` from the first to the last column in the order shown:

1.0000 ¹	0.5000 ⁵	0.3333 ⁹	0.2500 ¹³
0.5000 ²	0.3333 ⁶	0.2500 ¹⁰	0.2000 ¹⁴
0.3333 ³	0.2500 ⁷	0.2000 ¹¹	0.1667 ¹⁵
0.2500 ⁴	0.2000 ⁸	0.1667 ¹²	0.1429 ¹⁶

Here you can see that `H([2 11])` extracts the 2nd and 11th elements of `H`, based on the preceding numbering scheme.

In image processing, linear indexing is useful for extracting a set of pixel values from arbitrary locations. For example, suppose we want an expression that extracts the values of `H` at row-column coordinates (1, 3), (2, 4), and (4, 3):

```
>> r = [1 2 4];
>> c = [3 4 3];
```

Expression $H(r, c)$ does not do what we want, as you can see:

```
>> H(r, c)
ans =
    0.3333    0.2500    0.3333
    0.2500    0.2000    0.2500
    0.1667    0.1429    0.1667
```

Instead, we convert the *row-column* coordinates to linear index values, as follows:

```
>> M = size(H, 1);
>> linear_indices = M*(c - 1) + r
linear_indices =
     9    14    12

>> H(linear_indices)
ans =
    0.3333    0.2000    0.1667
```

MATLAB functions `sub2ind` and `ind2sub` convert back and forth between row-column subscripts and linear indices. For example,



sub2ind

```
>> linear_indices = sub2ind(size(H), r, c)
linear_indices =
     9    14    12
```



ind2sub

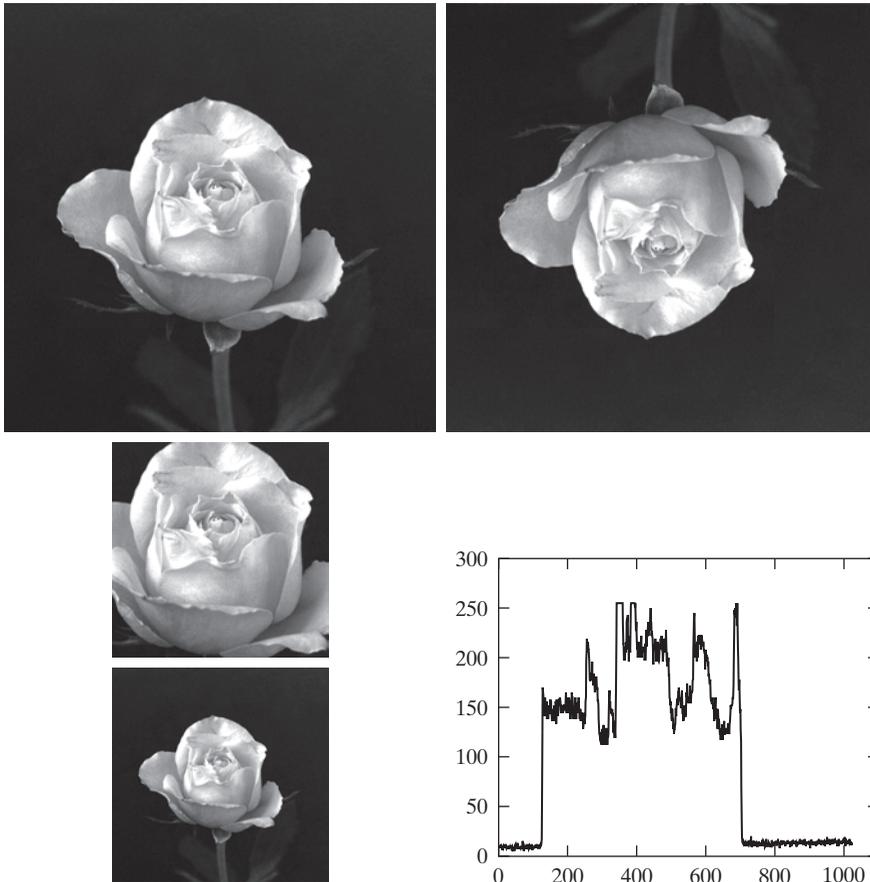
```
>> [r, c] = ind2sub(size(H), linear_indices)
r =
     1     2     4
c =
     3     4     3
```

Linear indexing is a basic staple in vectorizing loops for program optimization, as discussed in Section 2.10.5.

EXAMPLE 2.5: Some simple image operations using array indexing.

■ The image in Fig. 2.7(a) is a 1024×1024 gray-scale image, `f`, of class `uint8`. The image in Fig. 2.7(b) was flipped vertically using the statement

```
>> fp = f(end:-1:1, :);
```



a b
c
d e

FIGURE 2.7

Results obtained using array indexing. (a) Original image. (b) Image flipped vertically. (c) Cropped image. (d) Subsampled image. (e) A horizontal scan line through the middle of the image in (a).

The image in Fig. 2.7(c) is a section out of image (a), obtained using the command

```
>> fc = f(257:768, 257:768);
```

Similarly, Fig. 2.7(d) shows a subsampled image obtained using the statement

```
>> fs = f(1:2:end, 1:2:end);
```

Finally, Fig. 2.7(e) shows a horizontal scan line through the middle of Fig. 2.7(a), obtained using the command

```
>> plot(f(512, :))
```

Function `plot` is discussed in Section 3.3.1.



2.8.6 Selecting Array Dimensions

Operations of the form

$$\text{operation}(A, \text{dim})$$

where *operation* denotes an applicable MATLAB operation, *A* is an array, and *dim* is a scalar, are used frequently in this book. For example, if *A* is a 2-D array, the statement

```
>> k = size(A, 1);
```

gives the size of *A* along its first dimension (i.e., it gives the number of rows in *A*). Similarly, the second dimension of an array is in the horizontal direction, so the statement `size(A, 2)` gives the number of columns in *A*. Using these concepts, we could have written the last command in Example 2.5 as

```
>> plot(f(size(f, 1)/2, :))
```

MATLAB does not restrict the number of dimensions of an array, so being able to extract the components of an array in any dimension is an important feature. For the most part, we deal with 2-D arrays, but there are several instances (as when working with color or multispectral images) when it is necessary to be able to “stack” images along a third or higher dimension. We deal with this in Chapters 7, 8, 12, and 13. Function `ndims`, with syntax

$$d = \text{ndims}(A)$$

gives the number of dimensions of array *A*. Function `ndims` never returns a value less than 2 because even scalars are considered two dimensional, in the sense that they are arrays of size 1×1 .

2.8.7 Sparse Matrices

When a matrix has a large number of 0s, it is advantageous to express it in *sparse* form to reduce storage requirements. Function `sparse` converts a matrix to sparse form by “squeezing out” all zero elements. The basic syntax for this function is

$$S = \text{sparse}(A)$$

For example, if

```
>> A = [1 0 0; 0 3 4; 0 2 0]
```

A =

```

1   0   0
0   3   4
0   2   0
```



Then

```
>> S = sparse(A)
S =
    (1,1)    1
    (2,2)    3
    (3,2)    2
    (2,3)    4
```

from which we see that `S` contains only the (row, col) locations of nonzero elements (note that the elements are sorted by columns). To recover the original (full) matrix, we use function `full`:

```
>> Original = full(S)
Original =
    1    0    0
    0    3    4
    0    2    0
```



A syntax used sometimes with function `sparse` has five inputs:

$$S = \text{sparse}(r, c, s, m, n)$$

where `r` and `c` are vectors containing, respectively, the row and column indices of the nonzero elements of the matrix we wish to express in sparse form. Parameter `s` is a vector containing the values corresponding to index pairs `(r, c)`, and `m` and `n` are the row and column dimensions of the matrix. For instance, the preceding matrix `S` can be generated directly using the command

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

The syntax `sparse(A)` requires that there be enough memory to hold the entire matrix. When that is not the case, and the location and values of all nonzero elements are known, the alternate syntax shown here provides a solution for generating a sparse matrix.

Arithmetic and other operations (Section 2.10.2) on sparse matrices are carried out in exactly the same way as with full matrices. There are a number of other syntax forms for function `sparse`, as detailed in the help page for this function.

2.9 Some Important Standard Arrays

Sometimes, it is useful to be able to generate image arrays with known characteristics to try out ideas and to test the syntax of functions during development. In this section we introduce eight array-generating functions that are used in

later chapters. If only one argument is included in any of the following functions, the result is a square array.

- `zeros(M, N)` generates an $M \times N$ matrix of 0s of class `double`.
- `ones(M, N)` generates an $M \times N$ matrix of 1s of class `double`.
- `true(M, N)` generates an $M \times N$ logical matrix of 1s.
- `false(M, N)` generates an $M \times N$ logical matrix of 0s.
- `magic(M)` generates an $M \times M$ “magic square.” This is a square array in which the sum along any row, column, or main diagonal, is the same. Magic squares are useful arrays for testing purposes because they are easy to generate and their numbers are integers.
- `eye(M)` generates an $M \times M$ identity matrix.
- `rand(M, N)` generates an $M \times N$ matrix whose entries are uniformly distributed random numbers in the interval $[0, 1]$.
- `randn(M, N)` generates an $M \times N$ matrix whose numbers are normally distributed (i.e., Gaussian) random numbers with mean 0 and variance 1.

For example,

```
>> A = 5*ones(3, 3)
A =
     5     5     5
     5     5     5
     5     5     5

>> magic(3)
ans =
     8     1     6
     3     5     7
     4     9     2

>> B = rand(2, 4)
B =
    0.2311    0.4860    0.7621    0.0185
    0.6068    0.8913    0.4565    0.8214
```

2.10 Introduction to M-Function Programming

One of the most powerful features of MATLAB is the capability it provides users to program their own new functions. As you will learn shortly, MATLAB function programming is flexible and particularly easy to learn.

2.10.1 M-Files

M-files in MATLAB (see Section 1.3) can be scripts that simply execute a series of MATLAB statements, or they can be functions that can accept arguments and can produce one or more outputs. The focus of this section is on M-

file functions. These functions extend the capabilities of both MATLAB and the Image Processing Toolbox to address specific, user-defined applications.

M-files are created using a text editor and are stored with a name of the form `filename.m`, such as `average.m` and `filter.m`. The components of a function M-file are

- The function definition line
- The H1 line
- Help text
- The function body
- Comments

The *function definition line* has the form

```
function [outputs] = name(inputs)
```

For example, a function to compute the sum and product (two different outputs) of two images would have the form

```
function [s, p] = sumprod(f, g)
```

where `f` and `g` are the input images, `s` is the sum image, and `p` is the product image. The name `sumprod` is chosen arbitrarily (subject to the constraints at the end of this paragraph), but the word `function` always appears on the left, in the form shown. Note that the output arguments are enclosed by square brackets and the inputs are enclosed by parentheses. If the function has a single output argument, it is acceptable to list the argument without brackets. If the function has no output, only the word `function` is used, without brackets or equal sign. Function names must begin with a letter, and the remaining characters can be any combination of letters, numbers, and underscores. No spaces are allowed. MATLAB recognizes function names up to 63 characters long. Additional characters are ignored.

Functions can be called at the command prompt. For example,

```
>> [s, p] = sumprod(f, g);
```

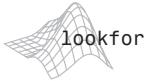
or they can be used as elements of other functions, in which case they become *subfunctions*. As noted in the previous paragraph, if the output has a single argument, it is acceptable to write it without the brackets, as in

```
>> y = sum(x);
```

The *H1 line* is the first text line. It is a *single* comment line that follows the function definition line. There can be no blank lines or leading spaces between the H1 line and the function definition line. An example of an H1 line is

```
%SUMPROD Computes the sum and product of two images.
```

It is customary to omit the space between % and the first word in the H1 line.



The H1 line is the first text that appears when a user types

```
>> help function_name
```

at the MATLAB prompt. Typing `lookfor` keyword displays all the H1 lines containing the string keyword. This line provides important summary information about the M-file, so it should be as descriptive as possible.

Help text is a text block that follows the H1 line, without any blank lines in between the two. Help text is used to provide comments and on-screen help for the function. When a user types `help function_name` at the prompt, MATLAB displays all comment lines that appear between the function definition line and the first noncomment (executable or blank) line. The help system ignores any comment lines that appear after the Help text block.

The *function body* contains all the MATLAB code that performs computations and assigns values to output arguments. Several examples of MATLAB code are given later in this chapter.

All lines preceded by the symbol “%” that are not the H1 line or Help text are considered function *comment* lines and are not considered part of the Help text block. It is permissible to append comments to the end of a line of code.

M-files can be created and edited using any text editor and saved with the extension `.m` in a specified directory, typically in the MATLAB search path. Another way to create or edit an M-file is to use the `edit` function at the prompt. For example,



```
>> edit sumprod
```

opens for editing the file `sumprod.m` if the file exists in a directory that is in the MATLAB path or in the Current Directory. If the file cannot be found, MATLAB gives the user the option to create it. The MATLAB editor window has numerous pull-down menus for tasks such as saving, viewing, and debugging files. Because it performs some simple checks and uses color to differentiate between various elements of code, the MATLAB text editor is recommended as the tool of choice for writing and editing M-functions.

2.10.2 Operators

MATLAB operators are grouped into three main categories:

- Arithmetic operators that perform numeric computations
- Relational operators that compare operands quantitatively
- Logical operators that perform the functions AND, OR, and NOT

These are discussed in the remainder of this section.

Arithmetic Operators

MATLAB has two different types of arithmetic operations. *Matrix arithmetic operations* are defined by the rules of linear algebra. *Array arithmetic operations* are carried out element by element and can be used with multidimensional arrays. The period (dot) character (`.`) distinguishes array operations



from matrix operations. For example, $A*B$ indicates matrix multiplication in the traditional sense, whereas $A.*B$ indicates array multiplication, in the sense that the result is an array, the same size as A and B , in which each element is the product of corresponding elements of A and B . In other words, if $C = A.*B$, then $C(I, J) = A(I, J) * B(I, J)$. Because matrix and array operations are the same for addition and subtraction, the character pairs $.+$ and $.-$ are not used.

When writing an expression such as $B = A$, MATLAB makes a “note” that B is equal to A , but does not actually copy the data into B unless the contents of A change later in the program. This is an important point because using different variables to “store” the same information sometimes can enhance code clarity and readability. Thus, the fact that MATLAB does not duplicate information unless it is absolutely necessary is worth remembering when writing MATLAB code. Table 2.5 lists the MATLAB arithmetic operators, where A and B are matrices or arrays and a and b are scalars. All operands can be real or complex. The dot shown in the array operators is not necessary if the operands are scalars. Because images are 2-D arrays, which are equivalent to matrices, all the operators in the table are applicable to images.

The difference between *array* and *matrix operations* is important. For example, consider the following:

Throughout the book, we use the term *array operations* interchangeably with the terminology *operations between pairs of corresponding elements*, and also *elementwise operations*.

TABLE 2.5 Array and matrix arithmetic operators. Characters a and b are scalars.

Operator	Name	Comments and Examples
$+$	Array and matrix addition	$a + b, A + B$, or $a + A$.
$-$	Array and matrix subtraction	$a - b, A - B, A - a$, or $a - A$.
$.*$	Array multiplication	$Cv = A.*B, C(I, J) = A(I, J) * B(I, J)$.
$*$	Matrix multiplication	$A*B$, standard matrix multiplication, or $a*A$, multiplication of a scalar times all elements of A .
$./$	Array right division [†]	$C = A ./ B, C(I, J) = A(I, J) / B(I, J)$.
$.\backslash$	Array left division [†]	$C = A .\backslash B, C(I, J) = B(I, J) / A(I, J)$.
$/$	Matrix right division	A/B is the preferred way to compute $A * \text{inv}(B)$.
\backslash	Matrix left division	$A \backslash B$ is the preferred way to compute $\text{inv}(A) * B$.
$.^$	Array power	If $C = A.^B$, then $C(I, J) = A(I, J) ^ B(I, J)$.
$^$	Matrix power	See <code>help</code> for a discussion of this operator.
$.'$	Vector and matrix transpose	$A.'$, standard vector and matrix transpose.
$'$	Vector and matrix complex conjugate transpose	A' , standard vector and matrix conjugate transpose. When A is real $A.' = A'$.
$+$	Unary plus	$+A$ is the same as $0 + A$.
$-$	Unary minus	$-A$ is the same as $0 - A$ or $-1 * A$.
$:$	Colon	Discussed in Section 2.8.1.

[†]In division, if the denominator is 0, MATLAB reports the result as `Inf` (denoting infinity). If both the numerator and denominator are 0, the result is reported as `NaN` (Not a Number).

$$A = \begin{bmatrix} a1 & a2 \\ a3 & a4 \end{bmatrix} \quad \text{and} \quad B = \begin{bmatrix} b1 & b2 \\ b3 & b4 \end{bmatrix}$$

The *array product* of A and B gives the result

$$A.*B = \begin{bmatrix} a1b1 & a2b2 \\ a3b3 & a4b4 \end{bmatrix}$$

whereas the *matrix product* yields the familiar result:

$$A*B = \begin{bmatrix} a1b1 + a2b3 & a1b2 + a2b4 \\ a3b1 + a4b3 & a3b2 + a4b4 \end{bmatrix}$$

Most of the arithmetic, relational, and logical operations involving images are array operations.

Example 2.6, to follow, uses functions `max` and `min`. The former function has the syntax forms



The syntax forms shown for `max` apply also to function `min`.

```
C = max(A)
C = max(A, B)
C = max(A, [ ], dim)
[C, I] = max(...)
```

In the first form, if A is a vector, `max(A)` returns its largest element; if A is a matrix, then `max(A)` treats the columns of A as vectors and returns a row vector containing the maximum element from each column. In the second form, `max(A, B)` returns an array the same size as A and B with the largest elements taken from A or B. In the third form, `max(A, [], dim)` returns the largest elements along the dimension of A specified by scalar `dim`. For example, `max(A, [], 1)` produces the maximum values along the first dimension (the rows) of A. Finally, `[C, I] = max(...)` also finds the indices of the maximum values of A, and returns them in output vector I. If there are duplicate maximum values, the index of the first one found is returned. The dots indicate the syntax used on the right of any of the previous three forms. Function `min` has the same syntax forms just described for `max`.

EXAMPLE 2.6: Illustration of arithmetic operators and functions `max` and `min`.

■ Suppose that we want to write an M-function, call it `imblend`, that forms a new image as an equally-weighted sum of two input images. The function should output the new image, as well as the maximum and minimum values of the new image. Using the MATLAB editor we write the desired function as follows:

```
function [w, wmax, wmin] = imblend(f, g)
%IMBLEND Weighted sum of two images.
% [W, WMAX, WMIN] = IMBLEND(F, G) computes a weighted sum (W) of
% two input images, F and G. IMBLEND also computes the maximum
% (WMAX) and minimum (WMIN) values of W. F and G must be of
% the same size and numeric class. The output image is of the
% same class as the input images.
```

```
w1 = 0.5 * f;
w2 = 0.5 * g;
w = w1 + w2;

wmax = max(w(:));
wmin = min(w(:));
```

Observe the use of single-colon indexing, as discussed in Section 2.8.1, to compute the minimum and maximum values. Suppose that $f = [1 \ 2; 3 \ 4]$ and $g = [1 \ 2; 2 \ 1]$. Calling `imblend` with these inputs results in the following output:

```
>> [w, wmax, wmin] = imblend(f, g)
w =
    1.0000    2.0000
    2.5000    2.5000
wmax =
    2.5000
wmin =
    1
```

Note in the code for `imblend` that the input images, f and g , were multiplied by the weights (0.5) first before being added together. Instead, we could have used the statement

```
>> w = 0.5 * (f + g);
```

However, this expression does not work well for integer classes because when MATLAB evaluates the subexpression $(f + g)$, it saturates any values that overflow the range of the class of f and g . For example, consider the following scalars:

```
>> f = uint8(100);
>> g = uint8(200);
>> t = f + g
t =
    255
```

Instead of getting a sum of 300, the computed sum saturated to the maximum value for the `uint8` class. So, when we multiply the sum by 0.5, we get an incorrect result:

```
>> d = 0.5 * t
d =
    128
```

Compare this with the result when we multiply by the weights first before adding:

```
>> e1 = 0.5 * f
e1 =
    50

>> e2 = 0.5 * g
e2 =
   100

>> e = w1 + w2
e =
   150
```

A good alternative is to use the image arithmetic function `imlincomb`, which computes a weighted sum of images, for any set of weights and any number of images. The calling syntax for this function is



```
g = imlincomb(k1, f1, k2, f2, ...)
```

For example, using the previous scalar values,

```
>> w = imlincomb(0.5, f, 0.5, g)
w =
   150
```

Typing `help imblend` at the command prompt results in the following output:

```
%IMBLEND Weighted sum of two images.
% [W, WMAX, WMIN] = IMBLEND(F, G) computes a weighted sum (W) of
% two input images, F and G. IMBLEND also computes the maximum
% (WMAX) and minimum (WMIN) values of W. F and G must be of
% the same size and numeric class. The output image is of the
% same class as the input images. ■
```

Relational Operators

MATLAB's relational operators are listed in Table 2.6. These are array operators; that is, they compare corresponding pairs of elements in arrays of equal dimensions.

EXAMPLE 2.7: Relational operators.

■ Although the key use of relational operators is in flow control (e.g., in `if` statements), which is discussed in Section 2.10.3, we illustrate briefly how these operators can be used directly on arrays. Consider the following:

```
>> A = [1 2 3; 4 5 6; 7 8 9]
```