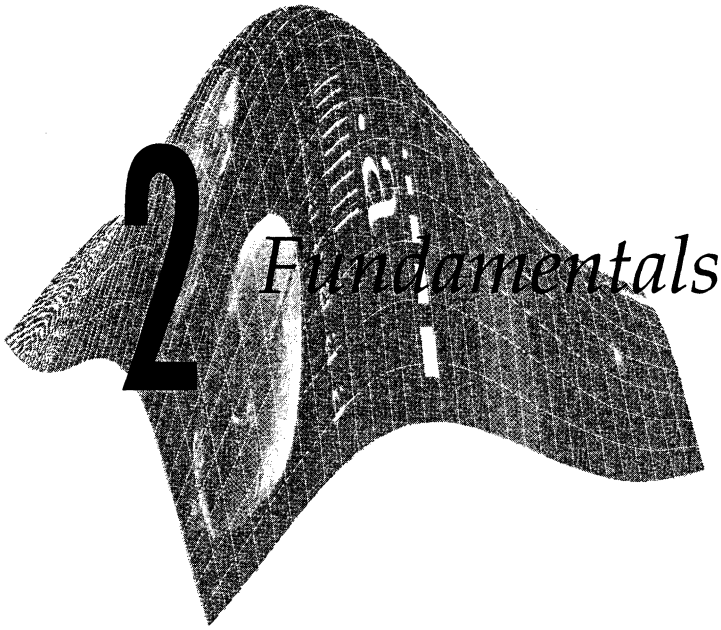


2 Fundamentals

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 (IPT) 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 many 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 IPT properties and functions, and introduce programming concepts that further enhance the power of IPT. Thus, the material in this chapter is the foundation for most of the material in the remainder of the book.

2.1 Digital Image Representation

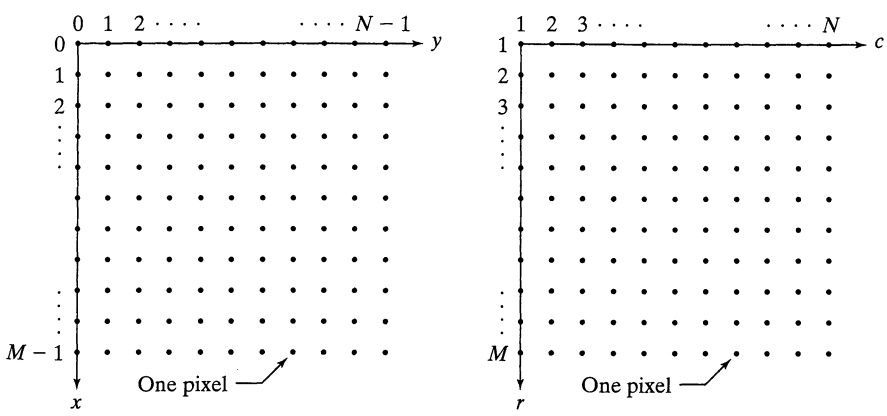
An image may be defined as a two-dimensional function, $f(x, y)$, where x and y are *spatial* (plane) *coordinates*, and the amplitude of f at any pair of coordinates (x, y) is called the *intensity* of the image at that point. The term *gray level* is used often to refer to the intensity of monochrome images. Color images are formed by a combination of individual 2-D images. For example, in the RGB color system, a color image consists of three (red, green, and blue) individual 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 treated in detail in Chapter 6.

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 (x, y) 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 $(x, y) = (0, 1)$. It is important to keep in mind that 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 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. This coordinate convention is shown in Fig. 2.1(b).



a b

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

IPT 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*, which uses x to refer to columns and y to refers to rows. This is the opposite of our use of variables x and y. With very few exceptions, we do not use IPT's spatial coordinate convention in this book, but the reader will definitely encounter the terminology in IPT 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 function:

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

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.

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

A digital image can be represented naturally as a MATLAB matrix:

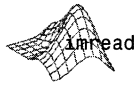
$$\mathtt{f} = \begin{bmatrix} \mathtt{f(1,1)} & \mathtt{f(1,2)} & \cdots & \mathtt{f(1,N)} \\ \mathtt{f(2,1)} & \mathtt{f(2,2)} & \cdots & \mathtt{f(2,N)} \\ \vdots & \vdots & & \vdots \\ \mathtt{f(M,1)} & \mathtt{f(M,2)} & \cdots & \mathtt{f(M,N)} \end{bmatrix}$$

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 the matrix `f`. Typically we use the letters `M` and `N`, respectively, to denote the number of rows and columns in a matrix. A `1 x N` matrix is called a *row vector*, whereas an `M x 1` matrix is called a *column vector*. A `1 x 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 syntax is




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

Format Name	Description	Recognized Extensions
TIFF	Tagged Image File Format	`.tif`, `.tiff`
JPEG	Joint Photographic Experts Group	`.jpg`, `.jpeg`
GIF	Graphics Interchange Format†	`.gif`
BMP	Windows Bitmap	`.bmp`
PNG	Portable Network Graphics	`.png`
XWD	X Window Dump	`.xwd`

†`GIF` is supported by `imread`, but not by `imwrite`.

TABLE 2.1 Some of the image/graphics formats supported by `imread` and `imwrite`, starting with MATLAB 6.5. Earlier versions support a subset of these formats. See online help for a complete list of supported formats.

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

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

reads the JPEG (Table 2.1) image `chestxray` into image array `f`. Note the use of single quotes (`'`) to delimit the string `filename`. The semicolon at the end of a command line is used by MATLAB for *suppressing* output. If a semicolon is not included, MATLAB displays 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* (see Section 1.7.1) and, if that fails, it tries to find the file in the MATLAB search path (see Section 1.7.1). 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,

In Windows, directories also are called folders.

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

reads the image from a folder called `myimages` on the `D:` drive, whereas

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

reads the image from the `myimages` subdirectory of the current working directory. The Current Directory Window on the MATLAB desktop toolbar displays MATLAB's current working directory and provides a simple, manual 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).

Function `size` gives the row and column dimensions of an image:


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

As in `size`, *many MATLAB and IPT functions can return more than one output argument. Multiple output arguments must be enclosed within square brackets,* [ ].

This function is particularly useful in programming when used in the following form to determine automatically the size of an image:

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

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

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


```
>> whos f
```

gives

```
  Name        Size              Bytes         Class
  f           1024x1024         1048576       uint8 array
  Grand total is 1048576 elements using 1048576 bytes
```

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:


```
imshow(f, G)
```

where `f` is an image array, and `G` is the number of intensity levels used to display it. If `G` is omitted, it defaults to 256 levels. 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 using the default number of levels. 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.

Function `pixval` is used frequently to display the intensity values of individual pixels interactively. This function displays a cursor overlaid on an image. As the cursor is moved over the image with the mouse, the coordinates of the cursor position and the corresponding intensity values are

shown on a display that appears below the figure window. When working with color images, the coordinates as well as the red, green, and blue components are displayed. If the left button on the mouse is clicked and then held pressed, `pixval` displays the Euclidean distance between the initial and current cursor locations.

The syntax form of interest here is

```
pixval
```

which shows the cursor on the last image displayed. Clicking the X button on the cursor window turns it off.

EXAMPLE 2.1: Image reading and displaying.

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

```
>> f = imread('rose_512.tif');
>> whos f
   Name        Size            Bytes        Class
   f           512x512         262144       uint8 array
   Grand total is 262144 elements using 262144 bytes
>> 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 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 results before they are printed or saved to disk.

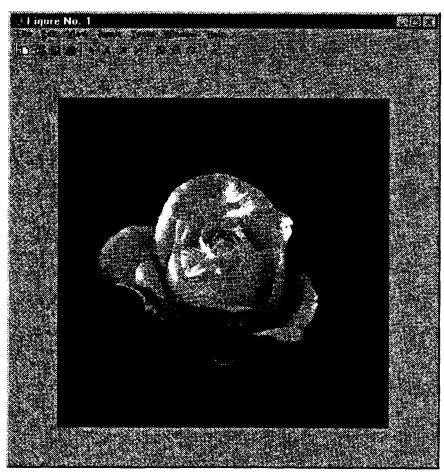


FIGURE 2.2 Screen capture showing how an image appears on the MATLAB desktop. However, in most of the examples throughout this book, only the images themselves are shown. Note the figure number on the top, left part of the window.

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

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.*

```
>> 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, as long as different commands are properly delimited by commas or semicolons. As mentioned earlier, a semicolon is used whenever it is desired to suppress screen outputs from a command line.

(b) Suppose that we have just read an image h and find that using imshow(h) produces the image in Fig. 2.3(a). It is clear that this image has a low dynamic range, which can be remedied for display purposes by using the statement

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

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

2.4 Writing Images

Images are written to disk 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). Alternatively, the desired format can be specified explicitly with a third input argument. For example, the following command writes f to a TIFF file named patient10_run1:

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

or, alternatively,

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

a b

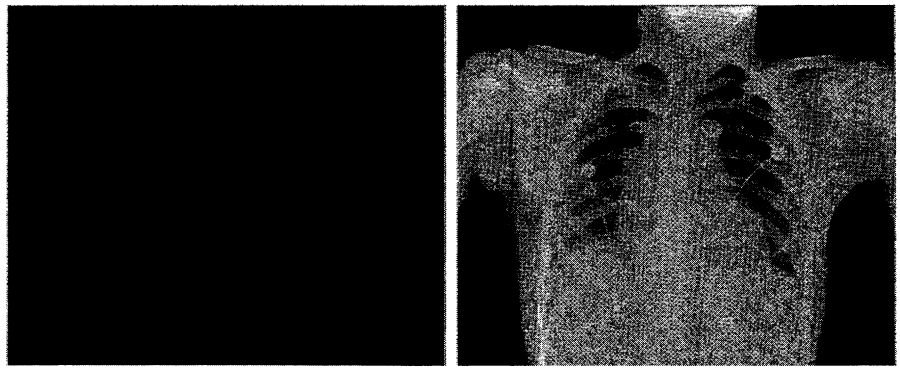


FIGURE 2.3 (a) An image, h, with low dynamic range. (b) Result of scaling by using imshow(h,[]). (Original image courtesy of Dr. David R. Pickens, Dept. of Radiology & Radiological Sciences, Vanderbilt University Medical Center.)

If `filename` contains no path information, then `imwrite` saves the file in the current working directory.

The `imwrite` function 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 2.4(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 and transmission time, it is important that the images be compressed as much as possible while not degrading their visual appearance beyond a reasonable level. In this case "reasonable" means no perceptible false contouring. Figures 2.4(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, for `q` = 25 the applicable syntax is

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

The image for `q` = 15 [Fig. 2.4(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 *complete* file name of the image stored in 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:  {}
```

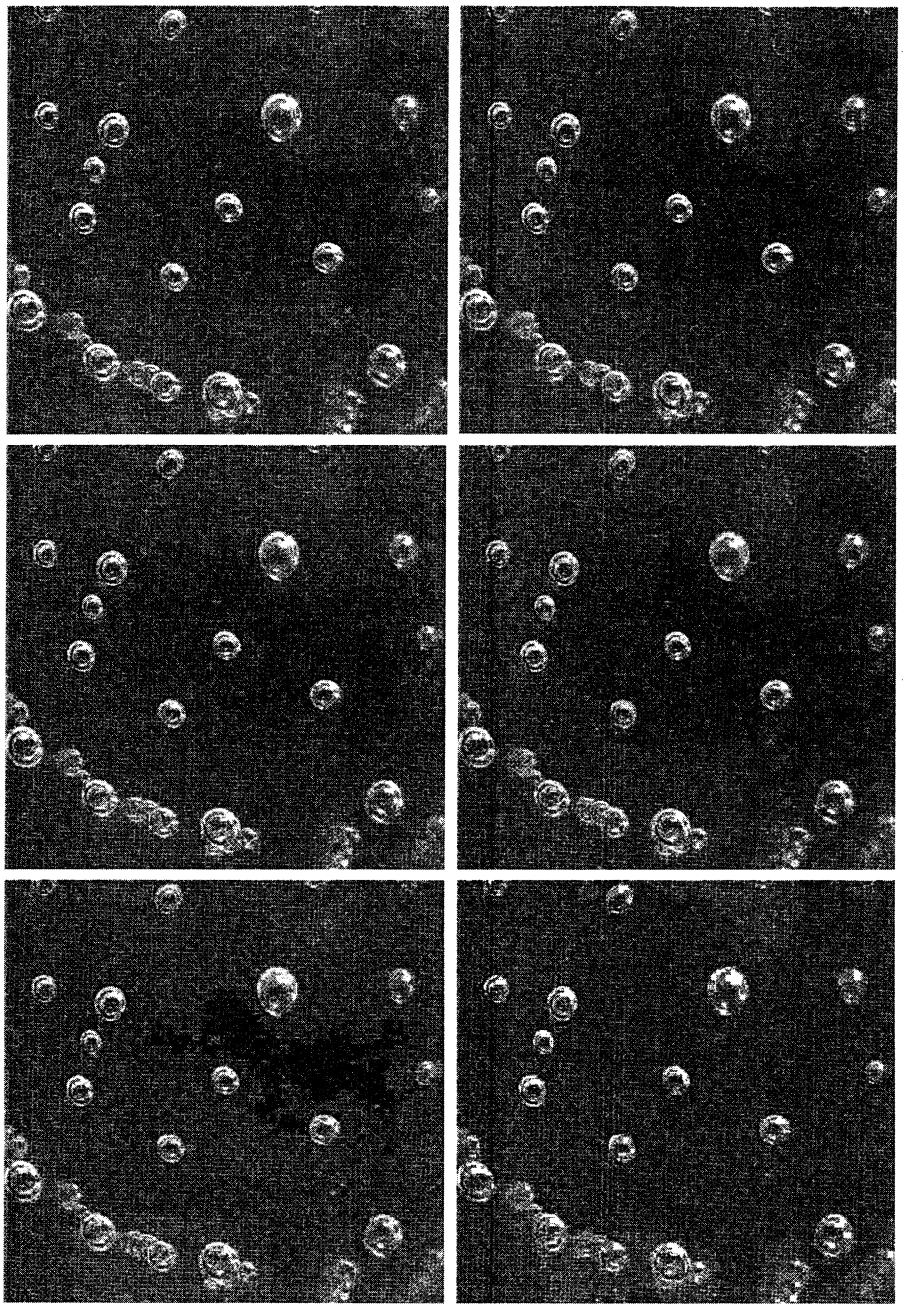


FIGURE 2.4
(a) Original image. (b) through (f) Results of using `jpg` quality values `q = 50, 25, 15, 5,` and `0`, respectively. False contouring begins to be barely noticeable for `q = 15` [image (d)] but is quite visible for `q = 5` and `q = 0`.

See Example 2.11 for a function that creates all the images in Fig. 2.4 using a simple `for` *loop.*

where `FileSize` is in bytes. The number of bytes in the original image is computed simply 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 appli-

cation. 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 assigning the name `K` to the structure variable, we use the syntax

Structures are discussed in Sections 2.10.6 and 11.1.1.

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

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

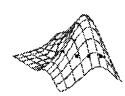
As an illustration, consider the following use of structure variable `K` to compute the compression ratio for `bubbles25.jpg`:

```
>> 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 online documentation. ■

To learn more about command function duality, consult the help page on this topic. See Section 1.7.3 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])
```

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.*

where `'parameter'` can have one of the following principal values: `'none'` indicates no compression; `'packbits'` indicates packbits compression (the default for nonbinary images); and `'ccitt'` indicates ccitt compression (the default for binary images). 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]`.

EXAMPLE 2.3: Using `imwrite` parameters.

■ Figure 2.5(a) is an 8-bit X-ray image 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 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 size of the image to 1.5 × 1.5 inches while keeping the pixel count at 450 × 450. The following statement yields the desired result:

```
>> 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 scale 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 is identical, except that its

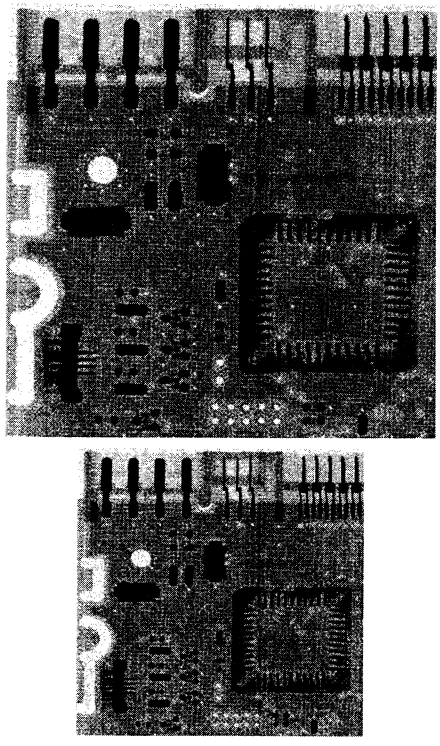


FIGURE 2.5 Effects of changing the dpi resolution while keeping the number of pixels constant. (a) A 450 × 450 image at 200 dpi (size = 2.25 × 2.25 inches). (b) The same 450 × 450 image, but at 300 dpi (size = 1.5 × 1.5 inches). (Original image courtesy of Lixi, Inc.)

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. ■

Often, it is necessary to export images to disk the way they appear on the MATLAB desktop. This is especially true with plots, as shown in the next chapter. 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 **Export**. 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 simply 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 Data Classes

Although we work with integer coordinates, the values of pixels themselves are not restricted to be integers in MATLAB. Table 2.2 lists the various *data classes*[†] supported by MATLAB and IPT for representing pixel values. The first eight entries in the table are referred to as *numeric* data classes. The ninth entry is the *char* class and, as shown, the last entry is referred to as the *logical* data class.

All *numeric* computations in MATLAB are done using `double` quantities, so this is also a frequent data class encountered in image processing applications. Class `uint8` also is encountered frequently, especially when reading data from storage devices, as 8-bit images are the most common representations found in practice. These two data classes, class `logical`, and, to a lesser degree, class `uint16`, constitute the primary data classes on which we focus in this book. Many IPT functions, however, support all the data classes listed in Table 2.2. Data class `double` requires 8 bytes to represent a number, `uint8` and `int8` require 1 byte each, `uint16` and `int16` require 2 bytes, and `uint32`,

[†]MATLAB documentation often uses the terms *data class* and *data type* interchangeably. In this book, we reserve use of the term *type* for images, as discussed in Section 2.6.

TABLE 2.2
Data classes. The first eight entries are referred to as *numeric* classes; the ninth entry is the *character* class, and the last entry is of class *logical*.

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

`int32`, and `single`, require 4 bytes each. The `char` data class holds characters in Unicode representation. A *character string* is merely a `1 x n` array of characters. A `logical` array contains only the values 0 and 1, with each element being stored in memory using one byte per element. Logical arrays are created by using function `logical` (see Section 2.6.2) or by using relational operators (Section 2.10.2).

2.6 Image Types

The toolbox supports four types of images:

- Intensity images
- Binary images
- Indexed images
- RGB images

Most monochrome image processing operations are carried out using binary or intensity images, so our initial focus is on these two image types. Indexed and RGB color images are discussed in Chapter 6.

2.6.1 Intensity Images

An *intensity image* is a data matrix whose values have been scaled to represent intensities. When the elements of an intensity image are of class `uint8`, or class `uint16`, they have integer values in the range [0, 255] and [0, 65535], respectively. If the image is of class `double`, the values are floating-point numbers. Values of scaled, class `double` intensity images are in the range [0, 1] by convention.

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 = logical(A)
```

If `A` contains elements other than 0s and 1s, use of 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 creates logical arrays.

To test if an array is 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 data class conversion functions discussed in Section 2.7.1.

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

2.6.3 A Note on Terminology

Considerable care was taken in the previous two sections to clarify the use of the terms *data class* and *image type*. In general, we refer to an image as being a "`data_class` image_type image," where `data_class` is one of the entries from Table 2.2, 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 "`unit8` intensity image" is simply referring to an intensity image whose pixels are of data class `unit8`. Some functions in the toolbox support all data classes, while others are very specific as to what constitutes a valid class. For example, the pixels in a binary image can only be of data class `logical`, as mentioned earlier.

2.7 Converting between Data Classes and Image Types

Converting between data classes and image types is a frequent operation in IPT applications. When converting between data classes, it is important to keep in mind the value ranges for each data class detailed in Table 2.2.

2.7.1 Converting between Data Classes

Converting between data classes is straightforward. The general syntax is

```
B = data_class_name(A)
```

where `data_class_name` is one of the names in the first column of Table 2.2. For example, suppose that `A` is an array of class `uint8`. A double-precision

array, B, is generated by the command B = double(A). This conversion is used routinely throughout the book because MATLAB expects operands in numerical computations to be double-precision, floating-point numbers. 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 converted to integers by discarding their fractional parts. 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 results in an array with logical 1s in locations where the input array had nonzero values, and logical 0s in places where the input array contained 0s.

2.7.2 Converting between Image Classes and Types

M-function changeclass, *discussed in Section 3.2.3, can be used for changing an input image to a specified class.*

The toolbox provides specific functions (Table 2.3) that perform the scaling necessary to convert between image classes and types. Function im2uint8 detects the data class of the input and performs all the necessary scaling for the toolbox to recognize the data as valid image data. For example, consider the following 2 × 2 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
```

TABLE 2.3 Functions in IPT for converting between image classes and types. See Table 6.3 for conversions that apply specifically to color images.

Name	Converts Input to:	Valid Input Image Data Classes
im2uint8	uint8	logical, uint8, uint16, and double
im2uint16	uint16	logical, uint8, uint16, and double
mat2gray	double (in range [0, 1])	double
im2double	double	logical, uint8, uint16, and double
im2bw	logical	uint8, uint16, and double

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. Note that the rounding behavior of `im2uint8` is different from the data-class conversion function `uint8` discussed in the previous section, which simply discards fractional parts.

Converting an arbitrary array of class `double` to an array of class `double` *scaled* to the range [0, 1] can be accomplished by using function `mat2gray` whose basic syntax is

```
g = mat2gray(A, [Amin, Amax])
```

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`. Writing

```
>> g = mat2gray(A);
```

sets the values of `Amin` and `Amax` to the actual minimum and maximum values in `A`. The input is assumed to be of class `double`. The output also is of class `double`.

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 already of class `double`, `im2double` returns an array that is 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 mentioned in the preceding paragraph, a `double` array having arbitrary values can be converted to a `double` array with values in the range [0, 1] by using function `mat2gray`.

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.

[†]Section 2.8.2 explains the use of square brackets and semicolons to specify a matrix.

Finally, we consider conversion between binary and intensity image types. Function `im2bw`, which has the syntax

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

produces a binary image, `g`, from an intensity image, `f`, by thresholding. The output binary image `g` has values of 0 for all pixels in the input image with intensity values less than threshold `T`, and 1 for all other pixels. The value specified for `T` has to be in the range [0, 1], regardless of the class of the input. The output binary image is automatically declared as a `logical` array by `im2bw`. If we write `g = im2bw(f)`, IPT uses a default value of 0.5 for `T`. If the input is an `uint8` image, `im2bw` divides all its pixels by 255 and then applies either the default or a specified threshold. If the input is of class `uint16`, the division is by 65535. If the input is a `double` image, `im2bw` applies either the default or a specified threshold directly. If the input is a `logical` array, the output is identical to the input. A logical (binary) array can be converted to a numerical array by using any of the four functions in the first column of Table 2.3.

EXAMPLE 2.4: Converting between image classes and types.

■ (a) We wish to convert the following `double` image

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

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 in Section 2.5, we can generate a binary array directly using relational operators (Section 2.10.2). Thus we get the same result by writing

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

We could store in a variable (say, gbv) the fact that gb is a logical array by using the `islogical` function, as follows:

```
>> gbv = islogical(gb)
gbv =
    1
```

(b) 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 a numeric array 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 an `uint8` numeric array 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 MATLAB supports nested statements, 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);
```

again demonstrating the compactness of the MATLAB language. ■

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). More sophisticated techniques are introduced as needed in subsequent discussions.

2.8.1 Vector Indexing

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 are accessed using one-dimensional indexing. Thus, `v(1)` is the first element of vector `v`, `v(2)` its second element, and so forth. The elements of vectors in MATLAB are enclosed by square brackets and are separated by spaces or by commas. 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 using the *transpose operator* (`.'`):

Using a single quote without the period computes the conjugate transpose. When the data are real, both transposes can be used interchangeably. See Table 2.4.

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

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. If v is a vector, writing

```
>> v(:)
```

produces a column vector, whereas writing

```
>> v(1:end)
```

produces a row 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 pick the first, fourth, and fifth elements of `v` using the command

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

As shown 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 Matrix Indexing

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]
```

displays the 3×3 matrix

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

Note that the use of semicolons here 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, we write

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

The colon operator is used in matrix indexing to select a two-dimensional block of elements out of a matrix. For example,

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

Here, use of the colon by itself is analogous to writing `A(1:3,3)`, which simply picks the third column of the matrix. Similarly, we extract the second row as follows:

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

The following statement extracts the top two rows:

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

To create a matrix `B` equal to `A` but with its last column set to 0s, we write

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

Operations using `end` are carried out in a manner similar to the examples given in the previous section for vector indexing. The following examples illustrate this.

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

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

Using vectors to index into a matrix provides a powerful approach for element selection. For example,

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

The notation `A([a b],[c d])` picks out the elements in `A` with coordinates (row `a`, column `c`), (row `a`, column `d`), (row `b`, column `c`), and (row `b`, column `d`). Thus, when we let `E = A([1 3], [2 3])` we are selecting the following elements in `A`: the element in row 1 column 2, the element in row 1 column 3, the element in row 3 column 2, and the element in row 3 column 3.

More complex schemes can be implemented using matrix addressing. A particularly useful addressing approach using matrices for indexing is of the form `A(D)`, where `D` is a logical array. For example, if

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

then

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

Finally, we point out that use of a single colon as an index into a matrix selects all the elements of the array (on a column-by-column basis) and arranges them in the form of a column vector. For example, with reference to matrix `T2`,

```
>> 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:


```
>> s = sum(A(:))
s =
    45
```

In general, `sum(v)` adds the values of all the elements of input vector `v`. If a matrix is input into `sum` [as in `sum(A)`], the output is a row vector containing the sums of each individual column of the input array (this behavior is typical of many MATLAB functions encountered in later chapters). By using a single colon in the manner just illustrated, we are in reality implementing the command

```
>> sum(sum(A));
```

because use of a single colon converts the matrix into a vector.

Using the colon notation is actually a form of *linear indexing* into a matrix or higher-dimensional array. In fact, MATLAB stores each array as a column of values regardless of the actual dimensions. This column consists of the array columns, appended end to end. For example, matrix `A` is stored in MATLAB as

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

Accessing `A` with a single subscript indexes directly into this column. For example, `A(3)` accesses the third value in the column, the number 7; `A(8)` accesses the eighth value, 6, and so on. When we use the column notation, we are simply

addressing all the elements, `A(1:end)`. This type of indexing is a basic staple in *vectorizing* loops for program optimization, as discussed in Section 2.10.4.

EXAMPLE 2.5: Some simple image operations using array indexing.

■ The image in Fig. 2.6(a) is a 1024 × 1024 intensity image, `f`, of class `uint8`. The image in Fig. 2.6(b) was flipped vertically using the statement

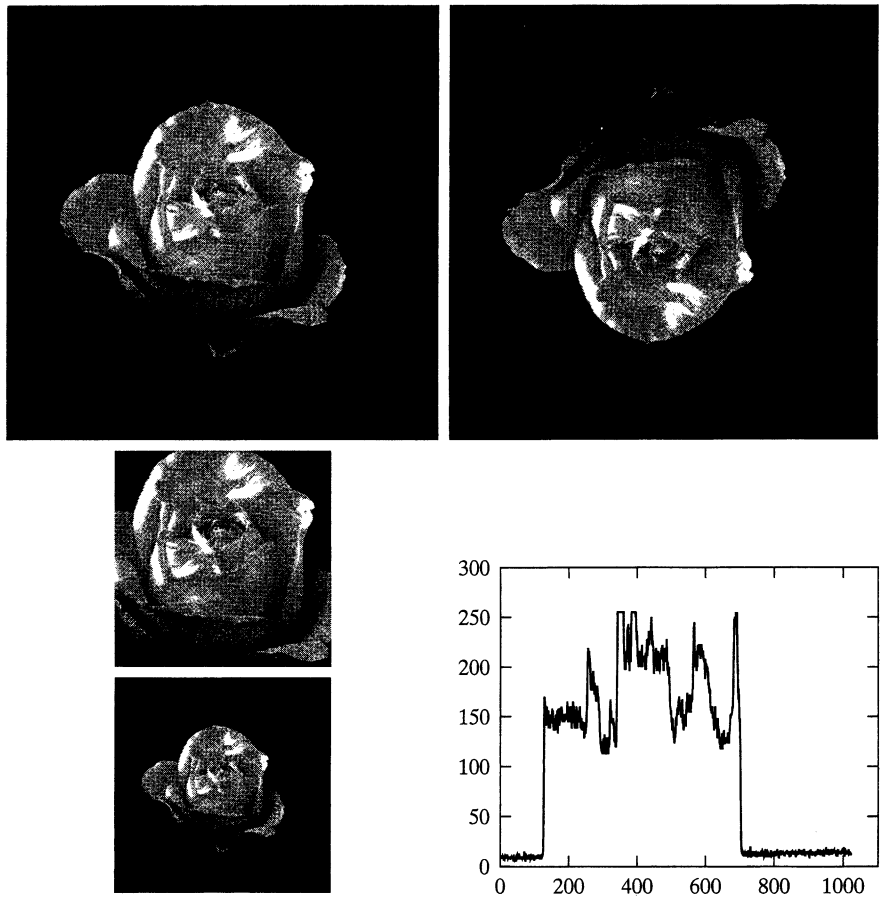
```
>> fp = f(end:-1:1, :);
```

The image shown in Fig. 2.6(c) is a section out of image (a), obtained using the command

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

Similarly, Fig. 2.6(d) shows a subsampled image obtained using the statement

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

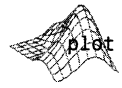


a b
c
d e

FIGURE 2.6 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).

Finally, Fig. 2.6(e) shows a horizontal scan line through the middle of Fig. 2.6(a), obtained using the command

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



The `plot` function is discussed in detail in Section 3.3.1. ■

2.8.3 Selecting Array Dimensions

Operations of the form

```
operation(A, 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, suppose that `A` is an array of size $M \times N$. The command

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

gives the size of `A` along its first dimension, which is defined by MATLAB as the vertical dimension. That is, this command 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`. A *singleton dimension* is any dimension, `dim`, for which `size(A, dim) = 1`. 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 6, 11, and 12. Function `ndims`, with syntax

```
d = 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.

2.9 Some Important Standard Arrays

Often, it is useful to be able to generate simple image arrays to try out ideas and to test the syntax of functions during development. In this section we introduce seven 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 × N matrix of 0s of class `double`.
- `ones(M, N)` generates an M × N matrix of 1s of class `double`.
- `true(M, N)` generates an M × N `logical` matrix of 1s.
- `false(M, N)` generates an M × N `logical` matrix of 0s.
- `magic(M)` generates an M × 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.
- `rand(M, N)` generates an M × N matrix whose entries are uniformly distributed random numbers in the interval [0, 1].
- `randn(M, N)` generates an M × 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 the Image Processing Toolbox is its transparent access to the MATLAB programming environment. As will become evident shortly, MATLAB function programming is flexible and particularly easy to learn.

2.10.1 M-Files

So-called *M-files* in MATLAB 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 in on M-file functions. These functions extend the capabilities of both MATLAB and IPT 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 arbitrarily defined, 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 distinguishes 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.
```

As indicated in Section 1.7.3, the H1 line is the first text that appears when a user types

```
>> help function_name
```

at the MATLAB prompt. Also, as mentioned in that section, 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 online 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. As noted in Section 1.7.2, 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, this 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 2.4 lists the MATLAB arithmetic operators, where A

Operator	Name	MATLAB Function	Comments and Examples
+	Array and matrix addition	plus(A, B)	a + b, A + B, or a + A.
–	Array and matrix subtraction	minus(A, B)	a – b, A – B, A – a, or a – A.
.*	Array multiplication	times(A, B)	C = A.*B, C(I, J) = A(I, J)*B(I, J).
*	Matrix multiplication	mtimes(A, B)	A*B, standard matrix multiplication, or a*A, multiplication of a scalar times all elements of A.
./	Array right division	rdivide(A, B)	C = A./B, C(I, J) = A(I, J)/B(I, J).
.\	Array left division	ldivide(A, B)	C = A.\B, C(I, J) = B(I, J)/A(I, J).
/	Matrix right division	mrdivide(A, B)	A/B is roughly the same as A*inv(B), depending on computational accuracy.
\	Matrix left division	mldivide(A, B)	A\B is roughly the same as inv(A)*B, depending on computational accuracy.
.^	Array power	power(A, B)	If C = A.^B, then C(I, J) = A(I, J)^B(I, J).
^	Matrix power	mpower(A, B)	See online help for a discussion of this operator.
.'	Vector and matrix transpose	transpose(A)	A.'. Standard vector and matrix transpose.
'	Vector and matrix complex conjugate transpose	ctranspose(A)	A'. Standard vector and matrix conjugate transpose. When A is real A.' = A'.
+	Unary plus	uplus (A)	+A is the same as 0 + A.
–	Unary minus	uminus (A)	–A is the same as 0 – A or –1*A.
:	Colon		Discussed in Section 2.8.

TABLE 2.4 Array and matrix arithmetic operators. Computations involving these operators can be implemented using the operators themselves, as in A + B, or using the MATLAB functions shown, as in plus (A, B). The examples shown for arrays use matrices to simplify the notation, but they are easily extendable to higher dimensions.

TABLE 2.5 The image arithmetic functions supported by IPT.

Function	Description
`imadd`	Adds two images; or adds a constant to an image.
`imsubtract`	Subtracts two images; or subtracts a constant from an image.
`immultiply`	Multiplies two images, where the multiplication is carried out between pairs of corresponding image elements; or multiplies a constant times an image.
`imdivide`	Divides two images, where the division is carried out between pairs of corresponding image elements; or divides an image by a constant.
`imabsdiff`	Computes the absolute difference between two images.
`imcomplement`	Complements an image. See Section 3.2.1.
`imlincomb`	Computes a linear combination of two or more images. See Section 5.3.1 for an example.

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. Keep in mind that images are 2-D arrays, which are equivalent to matrices, so all the operators in the table are applicable to images.

The toolbox supports the *image* arithmetic functions listed in Table 2.5. Although these functions could be implemented using MATLAB arithmetic operators directly, the advantage of using the IPT functions is that they support the integer data classes whereas the equivalent MATLAB math operators require inputs of class `double`.

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


```
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 several identical 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.

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

■ Suppose that we want to write an M-function, call it fgprod, that multiplies two input images and outputs the product of the images, the maximum and minimum values of the product, and a normalized product image whose values are in the range [0, 1]. Using the text editor we write the desired function as follows:

```
function [p, pmax, pmin, pn] = improd(f, g)
%IMPROD Computes the product of two images.
%  [P, PMAX, PMIN, PN] = IMPROD(F, G)† outputs the element-by-
%  element product of two input images, F and G, the product
%  maximum and minimum values, and a normalized product array with
%  values in the range [0, 1]. The input images must be of the same
%  size.  They can be of class uint8, unit16, or double. The outputs
%  are of class double.
fd = double(f);
gd = double(g);
p = fd.*gd;
pmax = max(p(:));
pmin = min(p(:));
pn = mat2gray(p);
```

Note that the input images were converted to double using the function double instead of im2double because, if the inputs were of type uint8, im2double would convert them to the range [0, 1]. Presumably, we want p to contain the product of the original values. To obtain a normalized array, pn, in the range [0, 1] we used function mat2gray. Note also the use of single-colon indexing, as discussed in Section 2.8.

Suppose that f = [1 2; 3 4] and g = [1 2; 2 1]. Typing the preceding function at the prompt results in the following output:

```
>> [p, pmax, pmin, pn] = improd(f, g)
p =
     1     4
     6     4
pmax =
     6
pmin =
     1
```

†In MATLAB documentation, it is customary to use uppercase characters in the H1 line and in Help text when referring to function names and arguments. This is done to avoid confusion between program names/variables and normal explanatory text.

```
pn =
         0    0.6000
    1.0000    0.6000
```

Typing `help improd` at the prompt results in the following output:

```
>> help improd
IMPROD Computes the product of two images.
  [P, PMAX, PMIN, PN] = IMPROD(F, G) outputs the element-by-
  element product of two input images, F and G, the product
  maximum and minimum values, and a normalized product array with
  values in the range [0, 1]. The input images must be of the same
  size.  They can be of class uint8, unit16, or double. The outputs
  are of class double.                                          ■
```

Relational Operators

MATLAB's relational operators are listed in Table 2.6. These operators compare corresponding elements of arrays of equal dimensions, on an element-by-element basis.

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 sequence of inputs and outputs:

```
>> A = [1 2 3; 4 5 6; 7 8 9]
A =
     1     2     3
     4     5     6
     7     8     9
>> B = [0 2 4; 3 5 6; 3 4 9]
B =
     0     2     4
     3     5     6
     3     4     9
>> A == B
```

TABLE 2.6 Relational operators.

Operator	Name
<	Less than
<=	Less than or equal to
>	Greater than
>=	Greater than or equal to
==	Equal to
~=	Not equal to

```
ans =
     0     1     0
     0     1     1
     0     0     1
```

Thus, we see that the operation A == B produces a logical array of the same dimensions as A and B, with 1s in locations where the corresponding elements of A and B match, and 0s elsewhere. As another illustration, the statement,

```
>> A >= B
ans =
     1     1     0
     1     1     1
     1     1     1
```

produces a logical array with 1s where the elements of A are greater than or equal to the corresponding elements of B and 0s elsewhere. ■

For vectors and rectangular arrays, both operands must have the same dimensions unless one operand is a scalar. In this case, MATLAB tests the scalar against every element of the other operand, yielding a logical array of the same size as the operand, with 1s in locations where the specified relation is satisfied and 0s elsewhere. If both operands are scalars, the result is a 1 if the specified relation is satisfied and 0 otherwise.

Logical Operators and Functions

Table 2.7 lists MATLAB's logical operators, and the following example illustrates some of their properties. Unlike most common interpretations of logical operators, the operators in Table 2.7 can operate on both logical *and* numeric data. MATLAB treats a logical 1 or nonzero numeric quantity as `true`, and a logical 0 or numeric 0 as `false` in all logical tests. For instance, the AND of two operands is 1 if both operands are logical 1s or nonzero numbers. The AND operation is 0 if either of its operands is logically or numerically 0, or if they both are logically or numerically 0.

Operator	Name
&	AND
|	OR
~	NOT

TABLE 2.7
Logical operators.

EXAMPLE 2.8: Logical operators.

■ Consider the AND operation on the following numeric arrays:

```
>> A = [1 2 0; 0 4 5];
>> B = [1 -2 3; 0 1 1];
>> A & B

ans =

    1    1    0
    0    1    1
```

We see that the AND operator produces a logical array that is of the same size as the input arrays and has a 1 at locations where both operands are nonzero and 0s elsewhere. Note that all operations are done on pairs of corresponding elements of the arrays, as before.

The OR operator works in a similar manner. An OR expression is `true` if either operand is a logical 1 or nonzero numerical quantity, or if they both are logical 1s or nonzero numbers; otherwise it is `false`. The NOT operator works with a single input. Logically, if the operand is `true`, the NOT operator converts it to `false`. When using NOT with numeric data, any nonzero operand becomes 0, and any zero operand becomes 1. ■

MATLAB also supports the logical functions summarized in Table 2.8. The `all` and `any` functions are particularly useful in programming.

EXAMPLE 2.9: Logical functions.

■ Consider the simple arrays `A = [1 2 3; 4 5 6]` and `B = [0 -1 1; 0 0 2]`. Substituting these arrays into the functions in Table 2.8 yield the following results:

```
>> xor(A, B)

ans =

    1    0    0
    1    1    0
```

TABLE 2.8 Logical functions.

Function	Comments
`xor` (exclusive OR)	The `xor` function returns a 1 only if both operands are logically different; otherwise `xor` returns a 0.
`all`	The `all` function returns a 1 if all the elements in a vector are nonzero; otherwise `all` returns a 0. This function operates columnwise on matrices.
`any`	The `any` function returns a 1 if any of the elements in a vector is nonzero; otherwise `any` returns a 0. This function operates columnwise on matrices.

```
>> all(A)
ans =
     1     1     1
>> any(A)
ans =
     1     1     1
>> all(B)
ans =
     0     0     1
>> any(B)
ans =
     0     1     1
```

Note how functions `all` and `any` operate on columns of `A` and `B`. For instance, the first two elements of the vector produced by `all(B)` are `0` because each of the first two columns of `B` contains at least one `0`; the last element is `1` because all elements in the last column of `B` are nonzero. ■

In addition to the functions listed in Table 2.8, MATLAB provides a number of other functions that test for the existence of specific conditions or values and return logical results. Some of these functions are listed in Table 2.9. A few of them deal with terms and concepts discussed earlier in this chapter (for example, see function `islogical` in Section 2.6.2); others are used in subsequent discussions. Keep in mind that the functions listed in Table 2.9 return a logical 1 when the condition being tested is true; otherwise they return a logical 0. When the argument is an array, some of the functions in Table 2.9 yield an array the same size as the argument containing logical 1s in the locations that satisfy the test performed by the function, and logical 0s elsewhere. For example, if `A = [1 2; 3 1/0]`, the function `isfinite(A)` returns the matrix `[1 1; 1 0]`, where the `0` (false) entry indicates that the last element of `A` is not finite.

Some Important Variables and Constants

The entries in Table 2.10 are used extensively in MATLAB programming. For example, `eps` typically is added to denominators in expressions to prevent overflow in the event that a denominator becomes zero.

TABLE 2.9
Some functions that return a logical 1 or a logical 0 depending on whether the value or condition in their arguments are `true` or `false`. See online help for a complete list.

Function	Description
`iscell(C)`	True if `C` is a cell array.
`iscellstr(s)`	True if `s` is a cell array of strings.
`ischar(s)`	True if `s` is a character string.
`isempty(A)`	True if `A` is the empty array, [ ].
`isequal(A, B)`	True if `A` and `B` have identical elements and dimensions.
`isfield(S, 'name')`	True if `'name'` is a field of structure `S`.
`isfinite(A)`	True in the locations of array `A` that are finite.
`isinf(A)`	True in the locations of array `A` that are infinite.
`isletter(A)`	True in the locations of `A` that are letters of the alphabet.
`islogical(A)`	True if `A` is a logical array.
`ismember(A, B)`	True in locations where elements of `A` are also in `B`.
`isnan(A)`	True in the locations of `A` that are `NaN`s (see Table 2.10 for a definition of `NaN`).
`isnumeric(A)`	True if `A` is a numeric array.
`isprime(A)`	True in locations of `A` that are prime numbers.
`isreal(A)`	True if the elements of `A` have no imaginary parts.
`isspace(A)`	True at locations where the elements of `A` are whitespace characters.
`issparse(A)`	True if `A` is a sparse matrix.
`isstruct(S)`	True if `S` is a structure.

TABLE 2.10
Some important variables and constants.

Function	Value Returned
`ans`	Most recent answer (variable). If no output variable is assigned to an expression, MATLAB automatically stores the result in `ans`.
`eps`	Floating-point relative accuracy. This is the distance between 1.0 and the next largest number representable using double-precision floating point.
`i` (or `j`)	Imaginary unit, as in `1 + 2i`.
`NaN` or `nan`	Stands for Not-a-Number (e.g., `0/0`).
`pi`	3.14159265358979
`realmax`	The largest floating-point number that your computer can represent.
`realmin`	The smallest floating-point number that your computer can represent.
`computer`	Your computer type.
`version`	MATLAB version string.

Number Representation

MATLAB uses conventional decimal notation, with an optional decimal point and leading plus or minus sign, for numbers. Scientific notation uses the letter `e` to specify a power-of-ten scale factor. Imaginary numbers use either `i` or `j` as a suffix. Some examples of valid number representations are

```
3           -99          0.0001
9.6397238   1.60210e-20  6.02252e23
1i          -3.14159j    3e5i
```

All numbers are stored internally using the long format specified by the Institute of Electrical and Electronics Engineers (IEEE) floating-point standard. Floating-point numbers have a finite precision of roughly 16 significant decimal digits and a finite range of approximately 10^{-308} to 10^{+308}.

2.10.3 Flow Control

The ability to control the flow of operations based on a set of predefined conditions is at the heart of all programming languages. In fact, conditional branching was one of two key developments that led to the formulation of general-purpose computers in the 1940s (the other development was the use of memory to hold stored programs and data). MATLAB provides the eight flow control statements summarized in Table 2.11. Keep in mind the observation made in the previous section that MATLAB treats a logical 1 or nonzero number as `true`, and a logical or numeric 0 as `false`.

Statement	Description
`if`	`if`, together with `else` and `elseif`, executes a group of statements based on a specified logical condition.
`for`	Executes a group of statements a fixed (specified) number of times.
`while`	Executes a group of statements an indefinite number of times, based on a specified logical condition.
`break`	Terminates execution of a `for` or `while` loop.
`continue`	Passes control to the next iteration of a `for` or `while` loop, skipping any remaining statements in the body of the loop.
`switch`	`switch`, together with `case` and `otherwise`, executes different groups of statements, depending on a specified value or string.
`return`	Causes execution to return to the invoking function.
`try...catch`	Changes flow control if an error is detected during execution.

TABLE 2.11 Flow control statements.

if, else, and elseif

Conditional statement `if` has the syntax

```
if expression
   statements
end
```

The *expression* is evaluated and, if the evaluation yields `true`, MATLAB executes one or more commands, denoted here as *statements*, between the `if` and `end` lines. If *expression* is `false`, MATLAB skips all the statements between the `if` and `end` lines and resumes execution at the line following the `end` line. When nesting `if`s, each `if` must be paired with a matching `end`.

The `else` and `elseif` statements further conditionalize the `if` statement. The general syntax is

```
if expression1
   statements1
elseif expression2
   statements2
else
   statements3
end
```

If *expression1* is `true`, *statements1* are executed and control is transferred to the `end` statement. If *expression1* evaluates to `false`, then *expression2* is evaluated. If this expression evaluates to `true`, then *statements2* are executed and control is transferred to the `end` statement. Otherwise (`else`) *statements3* are executed. Note that the `else` statement has no condition. The `else` and `elseif` statements can appear by themselves after an `if` statement; they do not need to appear in pairs, as shown in the preceding general syntax. It is acceptable to have multiple `elseif` statements.

EXAMPLE 2.10: Conditional branching and introduction of functions `error`, `length`, and `numel`.

■ Suppose that we want to write a function that computes the average intensity of an image. As discussed earlier, a two-dimensional array `f` can be converted to a column vector, `v`, by letting `v = f(:)`. Therefore, we want our function to be able to work with both vector and image inputs. The program should produce an error if the input is not a one- or two-dimensional array.

```
function av = average(A)
%AVERAGE Computes the average value of an array.
%  AV = AVERAGE(A) computes the average value of input
%  array, A, which must be a 1-D or 2-D array.

% Check the validity of the input. (Keep in mind that
% a 1-D array is a special case of a 2-D array.)
if ndims(A) > 2
   error('The dimensions of the input cannot exceed 2.')
end
```

```
% Compute the average
av = sum(A(:))/length(A(:));
```

Note that the input is converted to a 1-D array by using `A(:)`. In general, `length(A)` returns the size of the longest dimension of an array, `A`. In this example, because `A(:)` is a vector, `length(A)` gives the number of elements of `A`. This eliminates the need to test whether the input is a vector or a 2-D array. Another way to obtain the number of elements in an array directly is to use function `numel`, whose syntax is

```
n = numel(A)
```

Thus, if `A` is an image, `numel(A)` gives its number of pixels. Using this function, the last executable line of the previous program becomes

```
av = sum(A(:))/numel(A);
```

Finally, note that the `error` function terminates execution of the program and outputs the message contained within the parentheses (the quotes shown are required). ■

for

As indicated in Table 2.11, a `for` loop executes a group of statements a specified number of times. The syntax is

```
for index = start:increment:end
   statements
end
```

It is possible to nest two or more `for` loops, as follows:

```
for index1 = start1:increment1:end
   statements1
   for index2 = start2:increment2:end
      statements2
   end
   additional loop1 statements
end
```

For example, the following loop executes 11 times:

```
count = 0;
for k = 0:0.1:1
   count = count + 1;
end
```

If the loop increment is omitted, it is taken to be 1. Loop increments also can be negative, as in `k = 0:-1:-10`. Note that no semicolon is necessary at the end of a `for` line. MATLAB automatically suppresses printing the values of a loop index. As discussed in detail in Section 2.10.4, considerable gains in program execution speed can be achieved by replacing `for` loops with so-called *vectorized code* whenever possible.

EXAMPLE 2.11: Using a `for` loop to write multiple images to file.

■ Example 2.2 compared several images using different JPEG quality values. Here, we show how to write those files to disk using a `for` loop. Suppose that we have an image, `f`, and we want to write it to a series of JPEG files with quality factors ranging from 0 to 100 in increments of 5. Further, suppose that we want to write the JPEG files with filenames of the form `series_xxx.jpg`, where `xxx` is the quality factor. We can accomplish this using the following `for` loop:

```
for q = 0:5:100
   filename = sprintf('series_%3d.jpg', q);
   imwrite(f, filename, 'quality', q);
end
```

Function `sprintf`, whose syntax in this case is

See the help page for `sprintf` *for other syntax forms applicable to this function.*

```
s = sprintf('characters1%ndcharacters2', q)
```

writes formatted data as a string, `s`. In this syntax form, `characters1` and `characters2` are character strings, and `%nd` denotes a decimal number (specified by `q`) with `n` digits. In this example, `characters1` is `series_`, the value of `n` is 3, `characters2` is `.jpg`, and `q` has the values specified in the loop. ■

while

A `while` loop executes a group of statements for as long as the expression controlling the loop is `true`. The syntax is

```
while expression
   statements
end
```

As in the case of `for`, `while` loops can be nested:

```
while expression1
   statements1
   while expression2
      statements2
   end
   additional loop1 statements
end
```

For example, the following nested `while` loops terminate when both `a` and `b` have been reduced to 0:

```
a = 10;
b = 5;
while a
   a = a - 1;
   while b
      b = b - 1;
   end
end
```

Note that to control the loops we used MATLAB's convention of treating a numerical value in a logical context as `true` when it is nonzero and as `false` when it is 0. In other words, `while a` and `while b` evaluate to `true` as long as `a` and `b` are nonzero.

As in the case of `for` loops, considerable gains in program execution speed can be achieved by replacing `while` loops with vectorized code (Section 2.10.4) whenever possible.

break

As its name implies, `break` terminates the execution of a `for` or `while` loop. When a `break` statement is encountered, execution continues with the next statement outside the loop. In nested loops, `break` exits only from the innermost loop that contains it.

continue

The `continue` statement passes control to the next iteration of the `for` or `while` loop in which it appears, skipping any remaining statements in the body of the loop. In nested loops, `continue` passes control to the next iteration of the loop enclosing it.

switch

This is the statement of choice for controlling the flow of an M-function based on different types of inputs. The syntax is

```
switch switch_expression
   case case_expression
      statement(s)
   case {case_expression1, case_expression2,...}
      statement(s)
   otherwise
      statement(s)
end
```

The switch construct executes groups of statements based on the value of a variable or expression. The keywords case and otherwise delineate the groups. Only the first matching case is executed.† There must always be an end to match the switch statement. The curly braces are used when multiple expressions are included in the same case statement. As a simple example, suppose that we have an M-function that accepts an image f and converts it to a specified class, call it newclass. Only three image classes are acceptable for the conversion: uint8, uint16, and double. The following code fragment performs the desired conversion and outputs an error if the class of the input image is not one of the acceptable classes:

```
switch newclass
   case 'uint8'
      g = im2uint8(f);
   case 'uint16'
      g = im2uint16(f);
   case 'double'
      g = im2double(f);
   otherwise
      error('Unknown or improper image class.')
end
```

The switch construct is used extensively throughout the book.

EXAMPLE 2.12: Extracting a subimage from a given image.

■ In this example we write an M-function (based on for loops) to extract a rectangular subimage from an image. Although, as shown in the next section, we could do the extraction using a single MATLAB statement, we use the present example later to compare the speed between loops and vectorized code. The inputs to the function are an image, the size (number of rows and columns) of the subimage we want to extract, and the coordinates of the top, left corner of the subimage. Keep in mind that the image origin in MATLAB is at (1, 1), as discussed in Section 2.1.1.

```
function s = subim(f, m, n, rx, cy)
%SUBIM Extracts a subimage, s, from a given image, f.
%  The subimage is of size m-by-n, and the coordinates
%  of its top, left corner are (rx, cy).
s = zeros(m, n);
rowhigh = rx + m - 1;
colhigh = cy + n - 1;
xcount = 0;
for r = rx:rowhigh
   xcount = xcount + 1;
   ycount = 0;
```

†Unlike the C language switch construct, MATLAB's switch does not "fall through." That is, switch executes only the first matching case; subsequent matching cases do not execute. Therefore, break statements are not used.

```
   for c = cy:colhigh
      ycount = ycount + 1;
      s(xcount, ycount) = f(r, c);
   end
end
```

In the following section we give a significantly more efficient implementation of this code. As an exercise, the reader should implement the preceding program using `while` instead of `for` loops. ■

2.10.4 Code Optimization

As discussed in some detail in Section 1.3, MATLAB is a programming language specifically designed for array operations. Taking advantage of this fact whenever possible can result in significant increases in computational speed. In this section we discuss two important approaches for MATLAB code optimization: *vectorizing* loops and *preallocating* arrays.

Vectorizing Loops

Vectorizing simply means converting `for` and `while` loops to equivalent vector or matrix operations. As will become evident shortly, vectorization can result not only in significant gains in computational speed, but it also helps improve code readability. Although multidimensional vectorization can be difficult to formulate at times, the forms of vectorization used in image processing generally are straightforward.

We begin with a simple example. Suppose that we want to generate a 1-D function of the form

$$f(x) = A \sin(x/2\pi)$$

for $x = 0, 1, 2, \ldots, M - 1$. A `for` loop to implement this computation is

```
for x = 1:M   % Array indices in MATLAB cannot be 0.
   f(x) = A*sin((x - 1)/(2*pi));
end
```

However, this code can be made considerably more efficient by vectorizing it; that is, by taking advantage of MATLAB indexing, as follows:

```
x = 0:M - 1;
f = A*sin(x/(2*pi));
```

As this simple example illustrates, 1-D indexing generally is a simple process. When the functions to be evaluated have two variables, optimized indexing is slightly more subtle. MATLAB provides a direct way to implement 2-D function evaluations via function `meshgrid`, which has the syntax

```
[C, R] = meshgrid(c, r)
```

This function transforms the domain specified by *row* vectors c and r into arrays C and R that can be used for the evaluation of functions of two variables and 3-D surface plots (note that columns are listed first in both the input and output of meshgrid).

The rows of output array C are copies of the vector c, and the columns of the output array R are copies of the vector r. For example, suppose that we want to form a 2-D function whose elements are the sum of the squares of the values of coordinate variables x and y for x = 0, 1, 2 and y = 0, 1. The vector r is formed from the row components of the coordinates: r = [0 1 2]. Similarly, c is formed from the column component of the coordinates: c = [0 1] (keep in mind that both r and c are row vectors here). Substituting these two vectors into meshgrid results in the following arrays:

```
>> [C, R]= meshgrid(c, r)
C =
     0     1
     0     1
     0     1
R =
     0     0
     1     1
     2     2
```

The function in which we are interested is implemented as

```
>> h = R.^2 + C.^2
```

which gives the following result:

```
h =
     0     1
     1     2
     4     5
```

Note that the dimensions of h are length(r) x length(c). Also note, for example, that h(1,1) = R(1,1)^2 + C(1,1)^2. Thus, MATLAB automatically took care of indexing h. This is a potential source for confusion when 0s are involved in the coordinates because of the repeated warnings in this book and in manuals that MATLAB arrays cannot have 0 indices. As this simple illustration shows, when forming h, MATLAB used the *contents* of R and C for computations. The *indices* of h, R, and C, started at 1. The power of this indexing scheme is demonstrated in the following example.

EXAMPLE 2.13: An illustration of the computational advantages of vectorization, and intruduction of the timing functions `tic` and `toc`.

■ In this example we write an M-function to compare the implementation of the following two-dimensional image function using `for` loops and vectorization:

$$f(x, y) = A \sin(u_0 x + v_0 y)$$

for $x = 0, 1, 2, \ldots, M - 1$ and $y = 0, 1, 2, \ldots, N - 1$. We also introduce the timing functions `tic` and `toc`.

The function inputs are A, u_0, v_0, M and N. The desired outputs are the images generated by both methods (they should be identical), and the ratio of the time it takes to implement the function with `for` loops to the time it takes to implement it using vectorization. The solution is as follows:

```
function [rt, f, g] = twodsin(A, u0, v0, M, N)
%TWODSIN Compares for loops vs. vectorization.
%  The comparison is based on implementing the function
%  f(x, y) = Asin(u0x + v0y) for x = 0, 1, 2,..., M - 1 and
%  y = 0, 1, 2,..., N - 1. The inputs to the function are
%  M and N and the constants in the function.

% First implement using for loops.

tic   % Start timing.
```


```
for r = 1:M
   u0x = u0*(r - 1);
   for c = 1:N
      v0y = v0*(c - 1);
      f(r, c) = A*sin(u0x + v0y);
   end
end

t1 = toc;   % End timing.
```


```
% Now implement using vectorization.  Call the image g.

tic   % Start timing.

r = 0:M - 1;
c = 0:N - 1;
[C, R] = meshgrid(c, r);
g = A*sin(u0*R + v0*C);

t2 = toc;   % End timing.

% Compute the ratio of the two times.

rt = t1/(t2 + eps); % Use eps in case t2 is close to 0.
```

Running this function at the MATLAB prompt,

```
>> [rt, f, g] = twodsin(1, 1/(4*pi), 1/(4*pi), 512, 512);
```

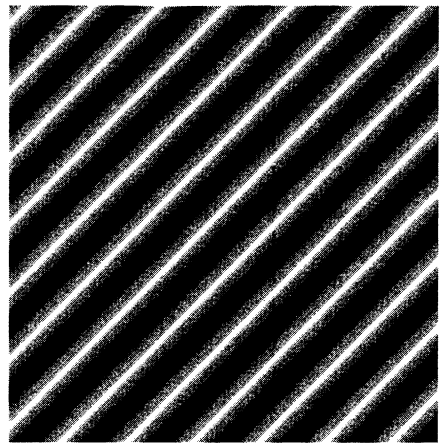


FIGURE 2.7 Sinusoidal image generated in Example 2.13.

yielded the following value of `rt`:

```
>> rt
rt =
    34.2520
```

We convert the image generated (`f` and `g` are identical) to viewable form using function `mat2gray`:

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

and display it using `imshow`,

```
>> imshow(g)
```

Figure 2.7 shows the result. ■

The vectorized code in Example 2.13 runs on the order of 30 times faster than the implementation based on `for` loops. This is a significant computational advantage that becomes increasingly meaningful as relative execution times become longer. For example, if `M` and `N` are large and the vectorized program takes 2 minutes to run, it would take over 1 hour to accomplish the same task using `for` loops. Numbers like these make it worthwhile to vectorize as much of a program as possible, especially if routine use of the program in envisioned.

The preceding discussion on vectorization is focused on computations involving the coordinates of an image. Often, we are interested in extracting and processing regions of an image. Vectorization of programs for extracting such regions is particularly simple if the region to be extracted is rectangular and encompasses all pixels within the rectangle, which generally is the case in this type of operation. The basic vectorized code to extract a region, `s`, of size `m` × `n` and with its top left corner at coordinates (`rx, cy`) is as follows:

```
rowhigh = rx + m - 1;
colhigh = cy + n - 1;
```

```
s = f(rx:rowhigh, cy:colhigh);
```

where f is the image from which the region is to be extracted. The for loops to accomplish the same thing were already worked out in Example 2.12. Implementing both methods and timing them as in Example 2.13 would show that the vectorized code runs on the order of 1000 times faster in this case than the code based on for loops.

Preallocating Arrays

Another simple way to improve code execution time is to preallocate the size of the arrays used in a program. When working with numeric or logical arrays, preallocation simply consists of creating arrays of 0s with the proper dimension. For example, if we are working with two images, f and g, of size 1024×1024 pixels, preallocation consists of the statements

```
>> f = zeros(1024); g = zeros(1024);
```

Preallocation also helps reduce memory fragmentation when working with large arrays. Memory can become fragmented due to dynamic memory allocation and deallocation. The net result is that there may be sufficient physical memory available during computation, but not enough contiguous memory to hold a large variable. Preallocation helps prevent this by allowing MATLAB to reserve sufficient memory for large data constructs at the beginning of a computation.

2.10.5 Interactive I/O

Often, it is desired to write interactive M-functions that display information and instructions to users and accept inputs from the keyboard. In this section we establish a foundation for writing such functions.

See Appendix B for details on constructing graphical user interfaces (GUIs).

Function disp is used to display information on the screen. Its syntax is

```
disp(argument)
```

If argument is an array, disp displays its contents. If argument is a text string, then disp displays the characters in the string. For example,

```
>> A = [1 2; 3 4];
>> disp(A)
    1   2
    3   4
>> sc = 'Digital Image Processing.';
>> disp(sc)
Digital Image Processing.
>> disp('This is another way to display text.')
This is another way to display text.
```

Note that only the contents of `argument` are displayed, without words like `ans =`, which we are accustomed to seeing on the screen when the value of a variable is displayed by omitting a semicolon at the end of a command line.

Function `input` is used for inputting data into an M-function. The basic syntax is


```
t = input('message')
```

This function outputs the words contained in `message` and waits for an input from the user, followed by a return, and stores the input in `t`. The input can be a single number, a character string (enclosed by single quotes), a vector (enclosed by square brackets and elements separated by spaces or commas), a matrix (enclosed by square brackets and rows separated by semicolons), or any other valid MATLAB data structure. The syntax

```
t = input('message', 's')
```

outputs the contents of `message` and accepts a *character* string whose elements can be separated by commas or spaces. This syntax is flexible because it allows multiple individual inputs. If the entries are intended to be numbers, the elements of the string (which are treated as characters) can be converted to numbers of class `double` by using the function `str2num`, which has the syntax


```
n = str2num(t)
```

See Section 12.4 for a detailed discussion of string operations.

For example,

```
>> t = input('Enter your data: ', 's')
Enter your data: 1, 2, 4
t =
    1 2 4
>> class(t)
ans =
    char
>> size(t)
ans =
    1   5
>> n = str2num(t)
n =
    1   2   4
```

```
>> size(n)
ans =
    1   3
>> class(n)
ans =
    double
```

Thus, we see that `t` is a 1×5 character array (the three numbers and the two spaces) and `n` is a 1×3 vector of numbers of class `double`.

If the entries are a mixture of characters and numbers, then we use one of MATLAB's string processing functions. Of particular interest in the present discussion is function `strread`, which has the syntax

```
[a, b, c, ...] = strread(cstr, 'format', 'param', 'value')
```

See the help page for `strread` *for a list of the numerous syntax forms applicable to this function.*

This function reads data from the character string `cstr`, using a specified `format` and `param/value` combinations. In this chapter the formats of interest are `%f` and `%q`, to denote floating-point numbers and character strings, respectively. For `param` we use `delimiter` to denote that the entities identified in `format` will be delimited by a character specified in `value` (typically a comma or space). For example, suppose that we have the string

```
>> t = '12.6, x2y, z';
```

To read the elements of this input into three variables `a`, `b`, and `c`, we write

```
>> [a, b, c] = strread(t, '%f%q%q', 'delimiter', ',')
a =
    12.6000
b =
    'x2y'
c =
    'z'
```

Output `a` is of class `double`; the quotes around outputs `x2y` and `z` indicate that `b` and `c` are `cell` arrays, which are discussed in the next section. We convert them to character arrays simply by letting


```
>> d = char(b)
d =
    x2y
```

and similarly for c. The number (and order) of elements in the format string must match the number and type of expected output variables on the left. In this case we expect three inputs: one floating-point number followed by two character strings.

Function `strcmp(s1, s2)` *compares two strings,* `s1` *and* `s2`*, and returns a logical* `true (1)` *if the strings are equal; otherwise it returns a logical* `false (0)`.

Function `strcmp` is used to compare strings. For example, suppose that we have an M-function `g = imnorm(f, param)` that accepts an image, `f`, and a parameter `param` than can have one of two forms: `'norm1'`, and `'norm255'`. In the first instance, `f` is to be scaled to the range [0, 1]; in the second, it is to be scaled to the range [0, 255]. The output should be of class `double` in both cases. The following code fragment accomplishes the required normalization:

```
f = double(f);
f = f - min(f(:));
f = f./max(f(:));
if strcmp(param, 'norm1')
    g = f;
elseif strcmp(param, 'norm255')
    g = 255*f;
else
    error('Unknown value of param.')
end
```

An error would occur if the value specified in `param` is not `'norm1'` or `'norm255'`. Also, an error would be issued if other than all lowercase characters are used for either normalization factor. We can modify the function to accept either lower or uppercase characters by converting any input to lowercase using function `lower`, as follows:


```
param = lower(param)
```

Similarly, if the code uses uppercase letters, we can convert any input character string to uppercase using function `upper`:


```
param = upper(param)
```

2.10.6 A Brief Introduction to Cell Arrays and Structures

Cell arrays and structures are discussed in detail in Section 11.1.1.

When dealing with mixed variables (e.g., characters and numbers), we can make use of cell arrays. A *cell array* in MATLAB is a multidimensional array whose elements are copies of other arrays. For example, the cell array

```
c = {'gauss', [1 0; 0 1], 3}
```

contains three elements: a character string, a 2×2 matrix, and a scalar (note the use of curly braces to enclose the arrays). To select the contents of a cell array we enclose an integer address in curly braces. In this case, we obtain the following results:

```
>> c{1}
ans =
    gauss
>> c{2}
ans =
    1   0
    0   1
>> c{3}
ans =
    3
```

An important property of cell arrays is that they contain *copies* of the arguments, not pointers to the arguments. For example, if we were working with cell array

```
c = {A, B}
```

in which `A` and `B` are matrices, and these matrices changed sometime later in a program, the contents of `c` would not change.

Structures are similar to cell arrays, in the sense that they allow grouping of a collection of dissimilar data into a single variable. However, unlike cell arrays where cells are addressed by numbers, the elements of structures are addressed by names called *fields*. Depending on the application, using fields adds clarity and readability to an M-function. For instance, letting `S` denote the structure variable and using the (arbitrary) field names `char_string`, `matrix`, and `scalar`, the data in the preceding example could be organized as a structure by letting

```
S.char_string = 'gauss';
S.matrix = [1 0; 0 1];
S.scalar = 3;
```

Note the use of a dot to append the various fields to the structure variable. Then, for example, typing `S.matrix` at the prompt, would produce

```
>> S.matrix
ans =
    1   0
    0   1
```

which agrees with the corresponding output for cell arrays. The clarity of using `S.matrix` as opposed to `c{2}` is evident in this case. This type of readability can be important if a function has numerous outputs that must be interpreted by a user.

Summary

The material in this chapter is the foundation for the discussions that follow. At this point, the reader should be able to retrieve an image from disk, process it via simple manipulations, display the result, and save it to disk. It is important to note that the key lesson from this chapter is how to combine MATLAB and IPT functions with programming constructs to generate solutions that expand the capabilities of those functions. In fact, this is the model of how material is presented in the following chapters. By combining standard functions with new code, we show prototypic solutions to a broad spectrum of problems of interest in digital image processing.