

## Introduction to MATLAB

---

MATLAB (short for MATrix LABoratory) is a special-purpose computer program optimized to perform engineering and scientific calculations. It started life as a program designed to perform matrix mathematics, but over the years it has grown into a flexible computing system capable of solving essentially any technical problem.

The MATLAB program implements the MATLAB programming language, and provides a very extensive library of predefined functions to make technical programming tasks easier and more efficient. This book introduces the MATLAB language as it is implemented in MATLAB Version 7 and shows how to use it to solve typical technical problems.

MATLAB is a huge program, with an incredibly rich variety of functions. Even the basic version of MATLAB without any toolkits is much richer than other technical programming languages. There are more than 1000 functions in the basic MATLAB product alone, and the toolkits extend this capability with many more functions in various specialties. This book makes no attempt to introduce the user to all of MATLAB's functions. Instead, it teaches a user the basics of how to write, debug, and optimize good MATLAB programs as well as a subset of the most important functions. Just as importantly, it teaches the programmer how to use MATLAB's own tools to locate the right function for a specific purpose from the enormous choice available.

## 1.1 The Advantages of MATLAB

MATLAB has many advantages compared to conventional computer languages for technical problem solving. Among them are:

### 1. Ease of Use

MATLAB is an interpreted language, like many versions of Basic. Like Basic, it is very easy to use. The program can be used as a scratch pad to evaluate expressions typed at the command line, or it can be used to execute large pre-written programs. Programs may be easily written and modified with the built-in integrated development environment and debugged with the MATLAB debugger. Because the language is so easy to use, it is ideal for the rapid prototyping of new programs.

Many program development tools are provided to make the program easy to use. They include an integrated editor/debugger, on-line documentation and manuals, a workspace browser, and extensive demos.

### 2. Platform Independence

MATLAB is supported on many different computer systems, providing a large measure of platform independence. At the time of this writing, the language is supported on Windows NT/2000/XP, Linux, several versions of Unix, and the Macintosh. Programs written on any platform will run on all of the other platforms, and data files written on any platform may be read transparently on any other platform. As a result, programs written in MATLAB can migrate to new platforms when the needs of the user change.

### 3. Predefined Functions

MATLAB comes complete with an extensive library of predefined functions that provide tested and pre-packaged solutions to many basic technical tasks. For example, suppose that you are writing a program that must calculate the statistics associated with an input data set. In most languages, you would need to write your own subroutines or functions to implement calculations such as the arithmetic mean, standard deviation, median, etc. These and hundreds of other functions are built right into the MATLAB language, making your job much easier.

In addition to the large library of functions built into the basic MATLAB language, there are many special-purpose toolboxes available to help solve complex problems in specific areas. For example, a user can buy standard toolboxes to solve problems in signal processing, control systems, communications, image processing, and neural networks, among many others. There is also an extensive collection of free user-contributed MATLAB programs that are shared through the MATLAB Web site.

### 4. Device-Independent Plotting

Unlike most other computer languages, MATLAB has many integral plotting and imaging commands. The plots and images can be displayed on any graphical output device supported by the computer on which MATLAB is

running. This capability makes MATLAB an outstanding tool for visualizing technical data.

#### 5. Graphical User Interface

MATLAB includes tools that allow a programmer to interactively construct a graphical user interface (GUI) for his or her program. With this capability, the programmer can design sophisticated data-analysis programs that can be operated by relatively inexperienced users.

#### 6. MATLAB Compiler

MATLAB's flexibility and platform independence is achieved by compiling MATLAB programs into a device-independent p-code, and then interpreting the p-code instructions at runtime. This approach is similar to that used by Microsoft's Visual Basic language. Unfortunately, the resulting programs can sometimes execute slowly because the MATLAB code is interpreted rather than compiled. We will point out features that tend to slow program execution when we encounter them.

A separate MATLAB compiler is available. This compiler can compile a MATLAB program into a true executable that runs faster than the interpreted code. It is a great way to convert a prototype MATLAB program into an executable suitable for sale and distribution to users.

## 1.2 Disadvantages of MATLAB

MATLAB has two principal disadvantages. The first is that it is an interpreted language and therefore can execute more slowly than compiled languages. This problem can be mitigated by properly structuring the MATLAB program, and by the use of the MATLAB compiler to compile the final MATLAB program before distribution and general use.

The second disadvantage is cost: a full copy of MATLAB is five to ten times more expensive than a conventional C or Fortran compiler. This relatively high cost is more than offset by the reduced time required for an engineer or scientist to create a working program, so MATLAB is cost-effective for businesses. However, it is too expensive for most individuals to consider purchasing. Fortunately, there is also an inexpensive Student Edition of MATLAB, which is a great tool for students wishing to learn the language. The Student Edition of MATLAB is essentially identical to the full edition.

## 1.3 The MATLAB Environment

The fundamental unit of data in any MATLAB program is the **array**. An array is a collection of data values organized into rows and columns and known by a single name. Individual data values within an array may be accessed by including the name of the array followed by subscripts in parentheses that identify the row and

column of the particular value. Even scalars are treated as arrays by MATLAB—they are simply arrays with only one row and one column. We will learn how to create and manipulate MATLAB arrays in Section 1.4.

When MATLAB executes, it can display several types of windows that accept commands or display information. The three most important types of windows are Command Windows, where commands may be entered; Figure Windows, which display plots and graphs; and Edit Windows, which permit a user to create and modify MATLAB programs. We will see examples of all three types of windows in this section.

In addition, MATLAB can display other windows that provide help and that allow the user to examine the values of variables defined in memory. We will examine some of these additional windows here. We will examine the others when we discuss how to debug MATLAB programs.

## The MATLAB Desktop

When you start MATLAB Version 7, a special window called the MATLAB desktop appears. The desktop is a window that contains other windows showing MATLAB data, plus toolbars and a “Start” button similar to that used by Windows 2000 or XP. By default, most MATLAB tools are “docked” to the desktop, so that they appear inside the desktop window. However, the user can choose to “undock” any or all tools, making them appear in windows separate from the desktop.

The default configuration of the MATLAB desktop is shown in Figure 1.1. It integrates many tools for managing files, variables, and applications within the MATLAB environment.

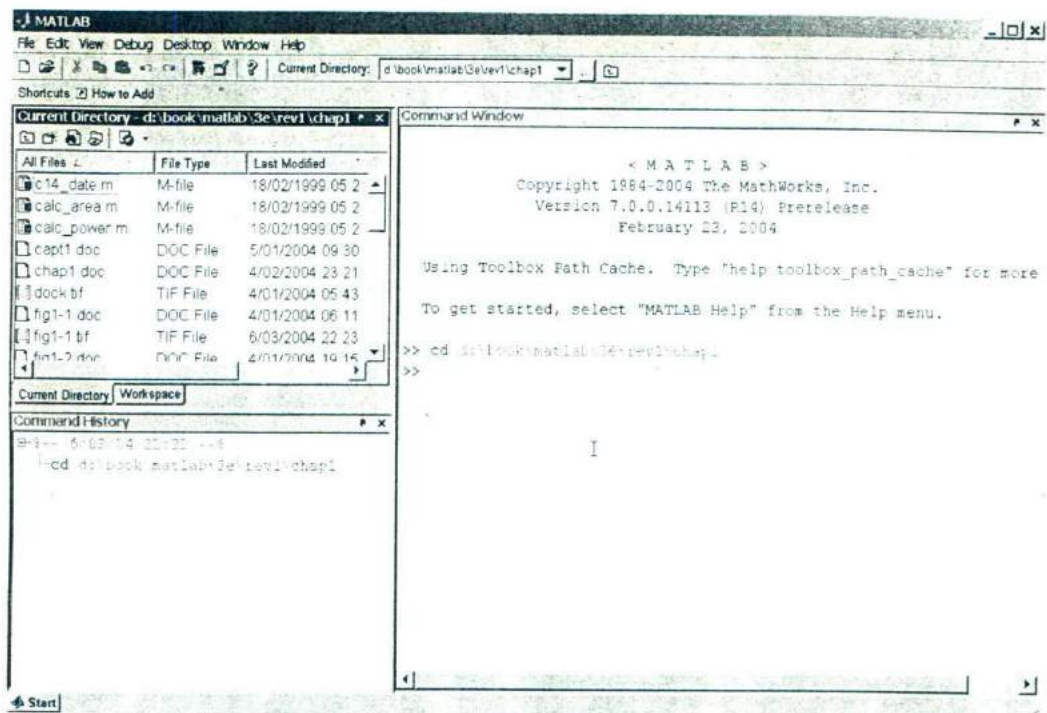
The major tools within or accessible from the MATLAB desktop are:

- The Command Window
- The Command History Window
- The Start Button
- The Documents Window, including the Editor/Debugger and Array Editor
- The Figure Windows
- The Workspace Browser
- The Help Browser
- The Path Browser

We will discuss the functions of these tools in later sections of this chapter.

## The Command Window

The right hand side of the default MATLAB desktop contains the **Command Window**. A user can enter interactive commands at the command prompt (`>>`) in the Command Window, and they will be executed on the spot.



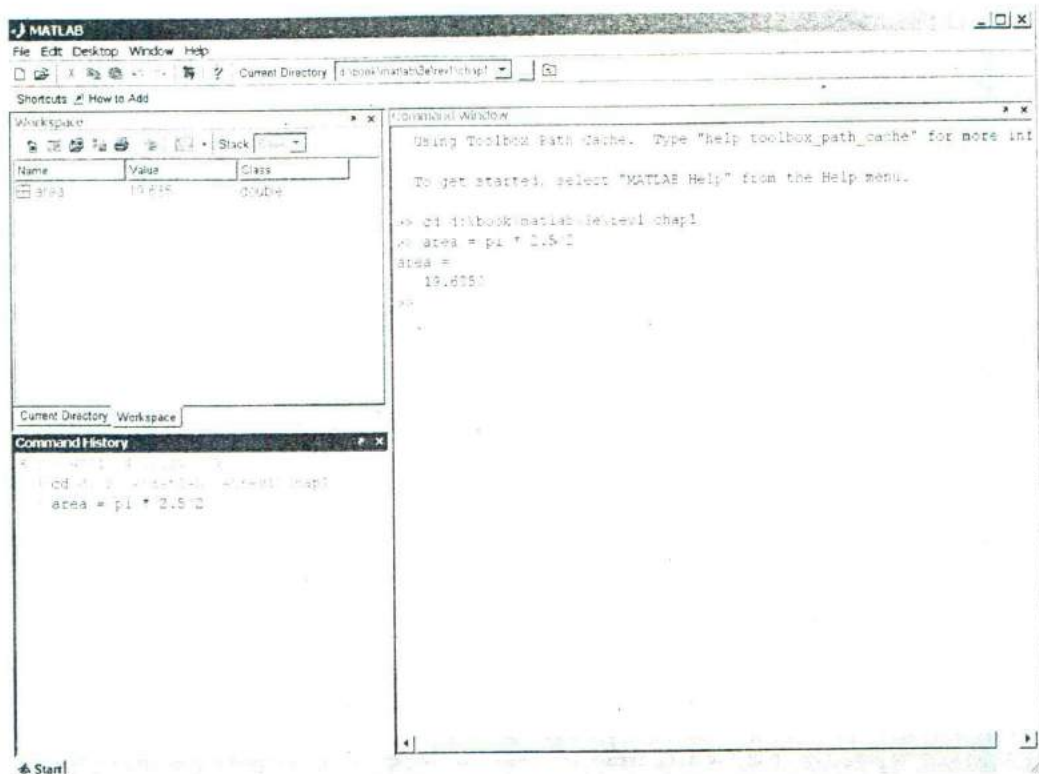
**Figure 1.1** The default MATLAB desktop. The exact appearance of the desktop may differ slightly on different types of computers.

As an example of a simple interactive calculation, suppose that you want to calculate the area of a circle with a radius of 2.5 m. This can be done in the MATLAB Command Window by typing:

```
>> area = pi * 2.5^2
area =
    19.6350
```

MATLAB calculates the answer as soon as the Enter key is pressed, and stores the answer in a variable (really a  $1 \times 1$  array) called `area`. The contents of the variable are displayed in the Command Window as shown in Figure 1.2, and the variable can be used in further calculations. (Note that  $\pi$  is predefined in MATLAB, so we can just use `pi` without first declaring it to be 3.141592...).

If a statement is too long to type on a single line, it may be continued on successive lines by typing an **ellipsis** (`...`) at the end of the first line, and then continuing on the next line. For example, the following two statements are identical.



**Figure 1.2** The Command Window appears on the right side of the desktop. Users enter commands and see responses here.

$$x1 = 1 + 1/2 + 1/3 + 1/4 + 1/5 + 1/6;$$

and

$$x1 = 1 + 1/2 + 1/3 + 1/4 \dots \\ + 1/5 + 1/6;$$

Instead of typing commands directly in the Command Window, a series of commands can be placed into a file, and the entire file can be executed by typing its name in the Command Window. Such files are called **script files**. Script files (and functions, which we will see later) are also known as **M-files**, because they have a file extension of “.m”.

## The Command History Window

The Command History window displays a list of the commands that a user has entered in the Command Window. The list of previous commands can extend back to previous executions of the program. Commands remain in the list until they are deleted. To re-execute any command, simply double-click it with the left mouse

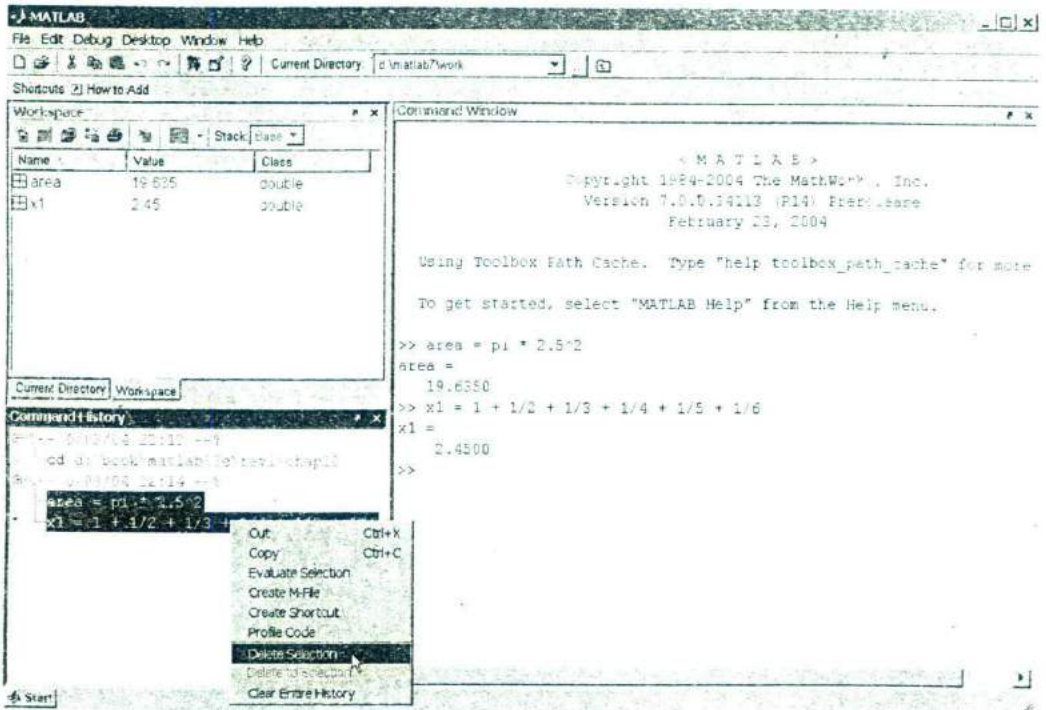




Figure 1.3 The Command History Window, showing two commands being deleted.

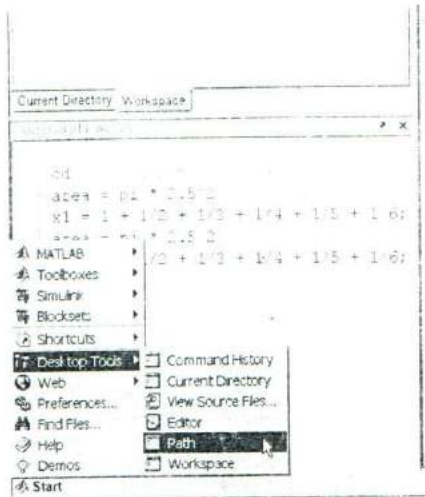
button. To delete one or more commands from the Command History window, select the commands and right-click them with the mouse. A popup menu will be displayed that allows the user to delete the items (see Figure 1.3).

## The Start Button

The Start Button (see Figure 1.4) allows a user to access MATLAB tools, desktop tools, help files, etc. It works just like the Start button on a Windows desktop. To start a particular tool, just click on the Start Button and select the tool from the appropriate sub-menu.

## The Edit/Debug Window

An **Edit Window** is used to create new M-files or to modify existing ones. An Edit Window is created automatically when you create a new M-file or open an existing one. You can create a new M-file with the “File/New M-file” selection from the desktop menu or by clicking the  toolbar icon. You can open an existing M-file with the “File/Open” selection from the desktop menu or by clicking the  toolbar icon.



**Figure 1.4** The Start Button, which allows a user to select from a wide variety of MATLAB and desktop tools.

An Edit Window displaying a simple M-file called `calc_area.m` is shown in Figure 1.5. This file calculates the area of a circle given its radius and displays the result. By default, the Edit Window is an independent window not docked to the desktop, as shown in Figure 1.5(a). The Edit Window can also be docked to the MATLAB desktop. In that case, it appears within a container called the Documents Window, as shown in Figure 1.5(b). We will learn how to dock and undock a window later in this chapter.

The Edit Window is essentially a programming text editor with the MATLAB languages features highlighted in different colors. Comments in an M-file appear in green, variables and numbers appear in black, complete character strings appear in magenta, incomplete character strings appear in red, and language keywords appear in blue.

After an M-file is saved, it may be executed by typing its name in the Command Window. For the M-file in Figure 1.5, the results are:

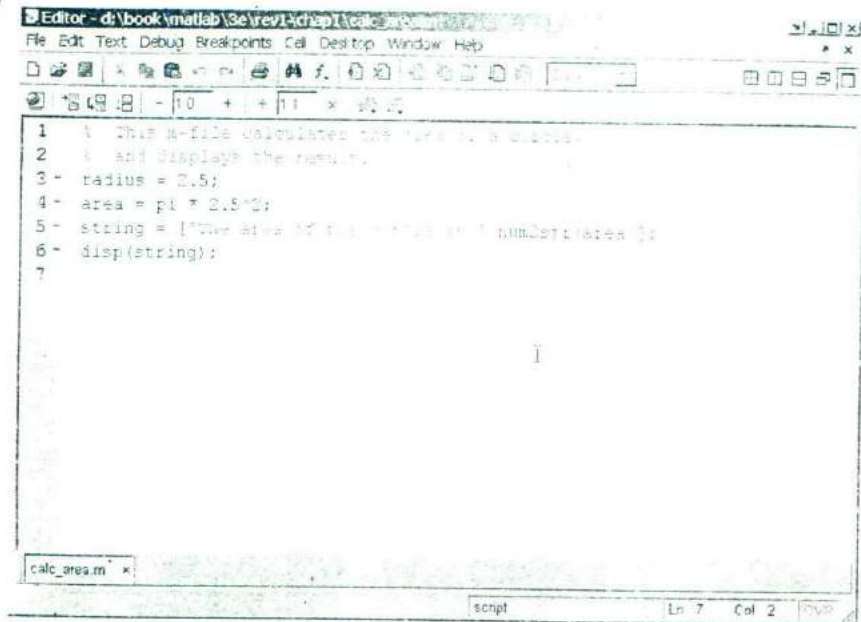
```
>> calc_area
The area of the circle is 19.635
```

The Edit Window also doubles as a debugger, as we shall see in Chapter 2.

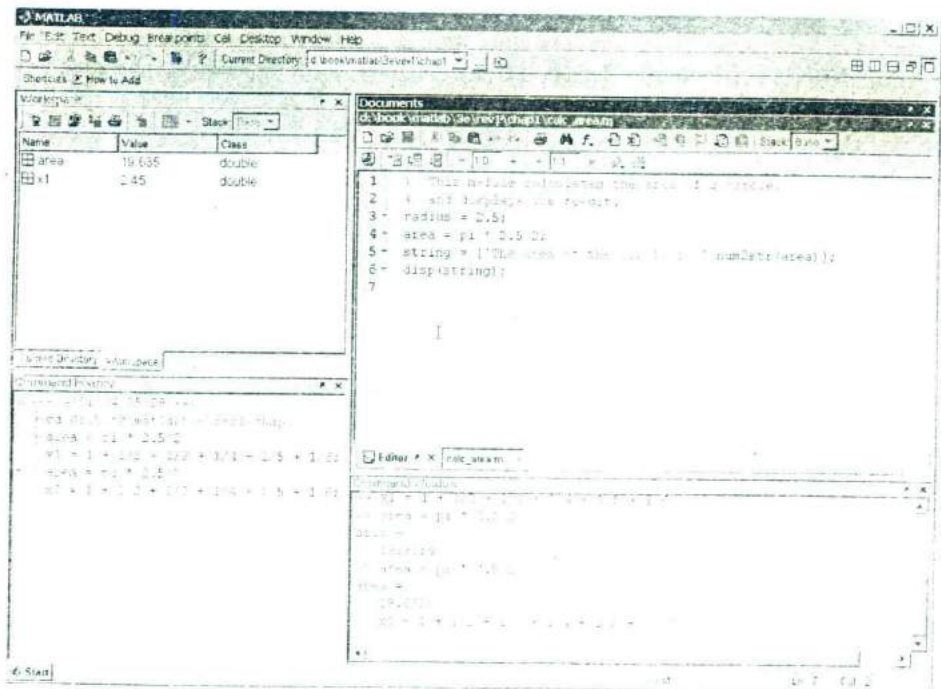
## Figure Windows

A **Figure Window** is used to display MATLAB graphics. A figure can be a two- or three-dimensional plot of data, an image, or a graphical user interface





(a)



(b)


**Figure 1.5** (a) The MATLAB Editor, displayed as an independent window. (b) The MATLAB Editor, docked to the MATLAB desktop.

(GUI). A simple script file that calculates and plots the function  $\sin x$  is shown below:

```
% sin_x.m: This M-file calculates and plots the
% function sin(x) for 0 <= x <= 6.
x = 0:0.1:6;
y = sin(x);
plot(x,y);
```

If this file is saved under the name `sin_x.m`, then a user can execute the file by typing "sin\_x" in the Command Window. When this script file is executed, MATLAB opens a figure window and plots the function  $\sin x$  in it. The resulting plot is shown in Figure 1.6.

## Docking and Undocking Windows

MATLAB windows such as the Command Window, the Edit Window, and Figure Windows can either be *docked* to the desktop, or they can be *undocked*. When a window is docked, it appears as a pane within the MATLAB desktop. When it is undocked it appears as an independent window on the computer screen separate from the desktop. When a window is docked to the desktop, the upper right-hand corner contains a small button with an arrow pointing up and to the right () . If this button is clicked, then the window will become an independent window.

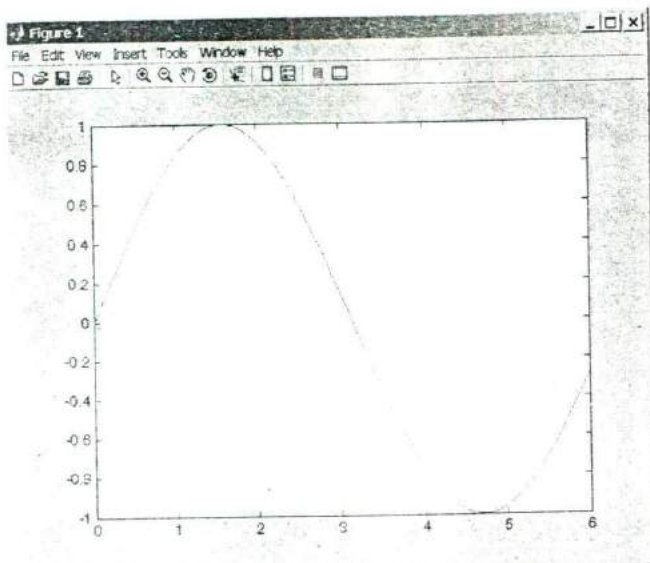


Figure 1.6 MATLAB plot of  $\sin x$  versus  $x$ .

When the window is an independent window, the upper right-hand corner contains a small button with an arrow pointing down and to the right (↘). If this button is clicked, then the window will be redocked with the desktop. Figure 1.5 shows the Edit Window in both its docked and undocked state. Note the undock and dock arrows in the upper right hand corner.

## The MATLAB Workspace

A statement such as

```
z = 10;
```

creates a variable named *z*, stores the value 10 in it, and saves it in a part of computer memory known as the **workspace**. A workspace is the collection of all the variables and arrays that can be used by MATLAB when a particular command, M-file, or function is executing. All commands executed in the Command Window (and all script files executed from the Command Window) share a common workspace, so they can all share variables. As we will see later, MATLAB functions differ from script files in that each function has its own separate workspace.

A list of the variables and arrays in the current workspace can be generated with the `whos` command. For example, after M-files `calc_area` and `sin_x` are executed, the workspace contains the following variables:

```
» whos
  Name      Size      Bytes    Class
  area      1x1         8      double array
  radius    1x1         8      double array
  string     1x32        64      char array
  x          1x61       488      double array
  y          1x61       488      double array
```

Grand total is 156 elements using 1056 bytes

Script file `calc_area` created variables `area`, `radius`, and `string`, while script file `sin_x` created variables `x` and `y`. Note that all of the variables are in the same workspace, so if two script files are executed in succession, the second script file can use variables created by the first script file.

The contents of any variable or array may be determined by typing the appropriate name in the Command Window. For example, the contents of `string` can be found as follows:

```
» string
string =
The area of the circle is 19.635
```

A variable can be deleted from the workspace with the `clear` command. The `clear` command takes the form

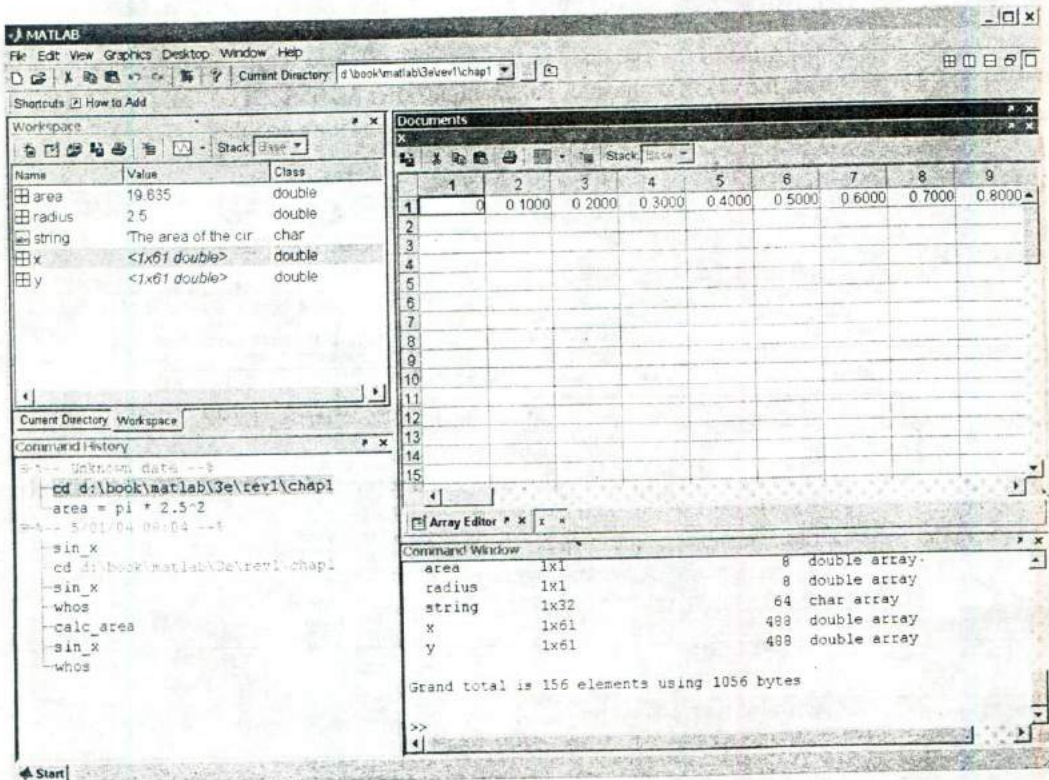
```
clear var1 var2 ...
```

where `var1` and `var2` are the names of the variables to be deleted. The command `clear variables` or simply `clear` deletes all variables from the current workspace.

## The Workspace Browser

The contents of the current workspace can also be examined with a GUI-based Workspace Browser. The Workspace Browser appears by default in the upper left-hand corner of the desktop. It provides a graphic display of the same information as the `whos` command, and it also shows the actual contents of each array if the information is short enough to fit within the display area. The Workspace Browser is dynamically updated whenever the contents of the workspace change.

A typical Workspace Browser window is shown in Figure 1.7. As you can see, it displays the same information as the `whos` command. Double-clicking on




**Figure 1.7** The Workspace Browser and the Array Editor. The Array Editor is invoked by double-clicking a variable in the Workspace Browser. It allows a user to change the values contained in a variable or array.

any variable in the window will bring up the Array Editor, which allows the user to modify the information stored in the variable.

One or more variables may be deleted from the workspace by selecting them in the Workspace Browser with the mouse and pressing the delete key, or by right-clicking with the mouse and selecting the delete option.

## Getting Help

There are three ways to get help in MATLAB. The preferred method is to use the Help Browser. The Help Browser can be started by selecting the  icon from the desktop toolbar, or by typing `helpdesk` or `helpwin` in the Command Window. A user can get help by browsing the MATLAB documentation, or he or she can search for the details of a particular command. The Help Browser is shown in Figure 1.8.

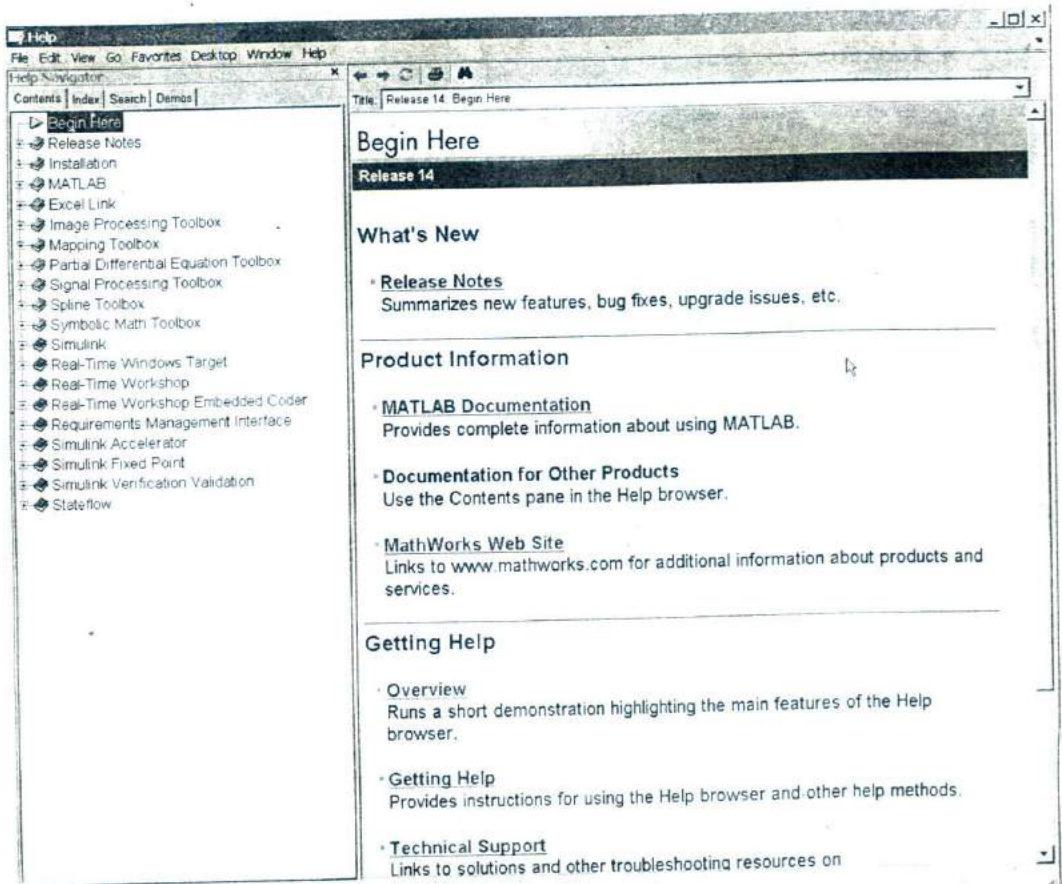


Figure 1.8 The Help Browser.

There are also two command-line oriented ways to get help. The first way is to type `help` or `help` followed by a function name in the Command Window. If you just type `help`, MATLAB will display a list of possible help topics in the Command Window. If a specific function or a toolbox name is included, help will be provided for that particular function or toolbox.

The second way to get help is the `lookfor` command. The `lookfor` command differs from the `help` command in that the `help` command searches for an exact function name match, while the `lookfor` command searches the quick summary information in each function for a match. This makes `lookfor` slower than `help`, but it improves the chances of getting back useful information. For example, suppose that you were looking for a function to take the inverse of a matrix. Since MATLAB does not have a function named `inverse`, the command “`help inverse`” will produce nothing. On the other hand, the command “`lookfor inverse`” will produce the following results:

```
» lookfor inverse
INVHILB   Inverse Hilbert matrix.
ACOS      Inverse cosine.
ACOSH     Inverse hyperbolic cosine.
ACOT      Inverse cotangent.
ACOTH     Inverse hyperbolic cotangent.
ACSC      Inverse cosecant.
ACSCH     Inverse hyperbolic cosecant.
ASEC      Inverse secant.
ASECH     Inverse hyperbolic secant.
ASIN      Inverse sine.
ASINH     Inverse hyperbolic sine.
ATAN      Inverse tangent.
ATAN2     Four quadrant inverse tangent.
ATANH     Inverse hyperbolic tangent.
ERFINV    Inverse error function.
INV       Matrix inverse.
PINV      Pseudoinverse.
IFFT      Inverse discrete Fourier transform.
IFFT2     Two-dimensional inverse discrete Fourier transform.
IFFTN     N-dimensional inverse discrete Fourier transform.
IPERMUTE  Inverse permute array dimensions.
```

From this list, we can see that the function of interest is named `inv`.

## A Few Important Commands

If you are new to MATLAB, a few demonstrations may help to give you a feel for its capabilities. To run MATLAB's built-in demonstrations, type `demo` in the Command Window or select “demos” from the Start Button.

The contents of the Command Window can be cleared at any time using the `clc` command, and the contents of the current Figure Window can be cleared at any time using the `clf` command. The variables in the workspace can be cleared with the `clear` command. As we have seen, the contents of the workspace persist between the executions of separate commands and M-files, so it is possible for the results of one problem to have an effect on the next one that you may attempt to solve. To avoid this possibility, it is a good idea to issue the `clear` command at the start of each new independent calculation.

Another important command is the **abort** command. If an M-file appears to be running for too long, it may contain an infinite loop, and it will never terminate. In this case, the user can regain control by typing control-c (abbreviated `^c`) in the Command Window. This command is entered by holding down the control key while typing a “c.” When MATLAB detects a `^c`, it interrupts the running program and returns a command prompt.

The exclamation point (!) is another important special character. Its special purpose is to send a command to the computer’s operating system. Any characters after the exclamation point will be sent to the operating system and executed as though they had been typed at the operating system’s command prompt. This feature lets you embed operating system commands directly into MATLAB programs.

Finally, it is possible to keep track of everything done during a MATLAB session with the **diary** command. The form of this command is

```
diary filename
```

After this command is typed, a copy of all input and most output typed in the Command Window is echoed in the diary file. This is a great tool for recreating events when something goes wrong during a MATLAB session. The command “diary off” suspends input into the diary file, and the command “diary on” resumes input again.

## The MATLAB Search Path

MATLAB has a search path that it uses to find M-files. MATLAB’s M-files are organized in directories on your file system. Many of these directories of M-files are provided along with MATLAB, and users may add others. If a user enters a name at the MATLAB prompt, the MATLAB interpreter attempts to find the name as follows:

1. It looks for the name as a variable. If it is a variable, MATLAB displays the current contents of the variable.
2. It checks to see if the name is an M-file in the current directory. If it is, MATLAB executes that function or command.
3. It checks to see if the name is an M-file in any directory in the search path. If it is, MATLAB executes that function or command.

Note that MATLAB checks for variable names first, so *if you define a variable with the same name as a MATLAB function or command, that function or command becomes inaccessible*. This is a common mistake made by novice users.

### Programming Pitfalls

Never use a variable with the same name as a MATLAB function or command. If you do so, that function or command will become inaccessible.

Also, if there is more than one function or command with the same name, the *first* one found on the search path will be executed, and all of the others will be inaccessible. This is a common problem for novice users, since they sometimes create M-files with the same names as standard MATLAB functions, making them inaccessible.

### Programming Pitfalls

Never create an M-file with the same name as a MATLAB function or command.

MATLAB includes a special command (`which`) to help you find out just which version of a file is being executed and where it is located. This can be

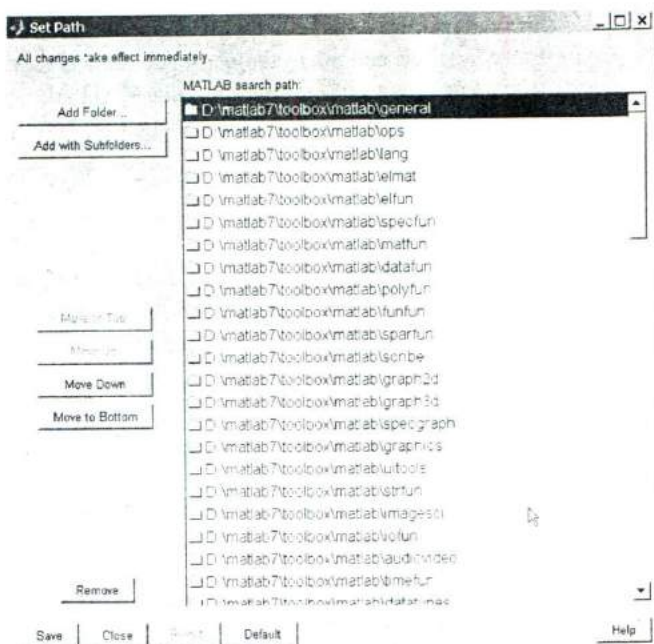


Figure 1.9 The Path Tool.



useful in finding filename conflicts. The format of this command is `which functionname`, where `functionname` is the name of the function that you are trying to locate. For example, the cross-product function, `cross.m` can be located as follows:

```
» which cross
C:\Matlab7\toolbox\matlab\specfun\cross.m
```

The MATLAB search path can be examined and modified at any time by selecting “Desktop Tools/Path” from the Start Button, or by typing `editpath` in the Command Window. The Path Tool is shown in Figure 1.9. It allows a user to add, delete, or change the order of directories in the path.

Other path-related functions include:

- `addpath`—Add directory to MATLAB search path.
- `path`—Display MATLAB search path.
- `path2rc`—Adds current directory to MATLAB search path.
- `rmpath`—Remove directory from MATLAB search path.

## 1.4 Using MATLAB as a Scratch Pad

In its simplest form, MATLAB can be used as a scratch pad to perform mathematical calculations. The calculations to be performed are typed directly into the Command Window, using the symbols `+`, `-`, `*`, `/`, and `^` for addition, subtraction, multiplication, division, and exponentiation, respectively. After an expression is typed, the results of the expression will be automatically calculated and displayed. For example, suppose we would like to calculate the volume of a cylinder of radius  $r$  and length  $l$ . The area of the circle at the base of the cylinder is given by the equation

$$A = \pi r^2 \quad (1-1)$$

and the total volume of the cylinder will be

$$V = Al \quad (1-2)$$

If the radius of the cylinder is 0.1 m and the length is 0.5 m, then the volume of the cylinder can be found using the MATLAB statements (user inputs are shown in bold face):

```
» A = pi * 0.1^2
A =
    0.0314
» V = A * 0.5
V =
    0.0157
```

Note that `pi` is predefined to be the value 3.141592 . . . Also, note that the value stored in `A` was saved by MATLAB and reused when we calculated `V`.

**Quiz 1.1**

This quiz provides a quick check to see if you have understood the concepts introduced in Chapter 1. If you have trouble with the quiz, reread the sections, ask your instructor, or discuss the material with a fellow student. The answers to this quiz are found in the back of the book.

1. What is the purpose of the MATLAB Command Window? The Edit Window? The Figure Window?
2. List the different ways that you can get help in MATLAB.
3. What is a workspace? How can you determine what is stored in a MATLAB workspace?
4. How can you clear the contents of a workspace?
5. The distance traveled by a ball falling in the air is given by the equation

$$x = x_0 + v_0t + \frac{1}{2}at^2$$

Use MATLAB to calculate the position of the ball at time  $t = 5$  s if  $x_0 = 10$  m,  $v_0 = 15$  m/s, and  $a = -9.81$  m/sec<sup>2</sup>.

6. Suppose that  $x = 3$  and  $y = 4$ . Use MATLAB to evaluate the following expression:

$$\frac{x^2y^3}{(x-y)^2}$$

The following questions are intended to help you become familiar with MATLAB tools.

7. Execute the M-files `calc_area.m` and `sin_x.m` in the Command Window (these M-files are available from the book's Web site). Then use the Workspace Browser to determine what variables are defined in the current workspace.
8. Use the Array Editor to examine and modify the contents of variable `x` in the workspace. The type the command `plot(x,y)` in the Command Window. What happens to the data displayed in the Figure Window?

**1.5 Summary**

In this chapter, we learned about the basic types of MATLAB windows, the workspace, and how to get on-line help. The MATLAB desktop appears when the program is started. It integrates many of the MATLAB tools in a single location. These tools include the Command Window, the Command History Window, the

Start Button, the Workspace Browser, the Array Editor, and the Current Directory Viewer. The Command Window is the most important of the windows. It is the one in which all commands are typed and results are displayed.

The Edit/Debug Window is used to create or modify M-files. It displays the contents of the M-file with the contents of the file color-coded according to function: comments, keywords, strings, and so forth. This window can be docked to the desktop, but by default it is independent.

The Figure Window is used to display graphics.

A MATLAB user can get help by using either the Help Browser or the command-line help functions `help` and `lookfor`. The Help Browser allows full access to the entire MATLAB documentation set. The command-line function `help` displays help about a specific function in the Command Window. Unfortunately, you must know the name of the function in order to get help about it. The function `lookfor` searches for a given string in the first comment line of every MATLAB function, and displays any matches.

When a user types a command in the Command Window, MATLAB searches for that command in the directories specified in the MATLAB path. It will execute the *first* M-file in the path that matches the command, and any further M-files with the same name will never be found. The Path Tool can be used to add, delete, or modify directories in the MATLAB path.

## MATLAB Summary

The following summary lists all of the MATLAB special symbols described in this chapter, along with a brief description of each one.

---

### Special Symbols

---

+	Addition
-	Subtraction
*	Multiplication
/	Division
^	Exponentiation

---

## 1.6 Exercises

- 1.1 The following MATLAB statements plot the function  $y(x) = 2e^{-0.2x}$  for the range  $0 \leq x \leq 10$ .

```
x = 0:0.1:10;
y = 2 * exp(0.2 * x);
plot(x,y);
```

Use the MATLAB Edit Window to create a new empty M-file, type these statements into the file, and save the file with the name `test1.m`. Then, execute the program by typing the name `test1` in the Command Window. What result do you get?

- 1.2 Get help on the MATLAB function `exp` using: (a) The “`help exp`” command typed in the Command Window and (b) the Help Browser.
- 1.3 Use the `lookfor` command to determine how to take the base-10 logarithm of a number in MATLAB.
- 1.4 Suppose that  $u = 1$  and  $v = 3$ . Evaluate the following expressions using MATLAB.

$$(a) \frac{4u}{3v}$$

$$(b) \frac{2v^{-2}}{(u+v)^2}$$

$$(c) \frac{v^3}{v^3 - u^3}$$

$$(d) \frac{4}{3}\pi v^2$$

- 1.5 Use the MATLAB Help Browser to find the command required to show MATLAB’s current directory. What is the current directory when MATLAB starts up?
- 1.6 Use the MATLAB Help Browser to find out how to create a new directory from within MATLAB. Then, create a new directory called `mynewdir` under the current directory. Add the new directory to the top of MATLAB’s path.
- 1.7 Change the current directory to `mynewdir`. Then open an Edit Window and add the following lines:

```
% Create an input array from -2*pi to 2*pi
t = -2*pi:pi/10:2*pi;

% Calculate |sin(t)|
x = abs(sin(t));

% Plot result
plot(t,x);
```

Save the file with the name `test2.m`, and execute it by typing `test2` in the Command Window. What happens?

- 1.8 Close the Figure Window, and change back to the original directory that MATLAB started up in. Next type “`test2`” in the Command Window. What happens, and why?

## MATLAB Basics

---

In this chapter, we introduce some basic elements of the MATLAB language. By the end of the chapter, you will be able to write simple but functional MATLAB programs.

### 2.1 Variables and Arrays

---

The fundamental unit of data in any MATLAB program is the **array**. An array is a collection of data values organized into rows and columns and known by a single name. Individual data values within an array are accessed by including the name of the array followed by subscripts in parentheses that identify the row and column of the particular value. Even scalars are treated as arrays by MATLAB—they are simply arrays with only one row and one column.

Arrays can be classified as either **vectors** or **matrices**. The term “vector” is usually used to describe an array with only one dimension, while the term “matrix” is usually used to describe an array with two or more dimensions. In this text, we will use the term “vector” when discussing one-dimensional arrays and the term “matrix” when discussing arrays with two or more dimensions. If a particular discussion applies to both types of arrays, we will use the generic term “array.” (See Figure 2.1)

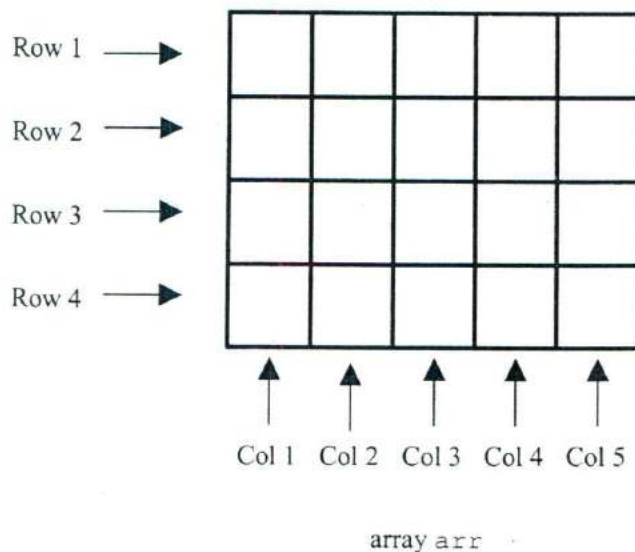
The **size** of an array is specified by the number of rows and the number of columns in the array, with the number of rows mentioned first. The total number of elements in the array will be the product of the number of rows and the number of columns. For example, the sizes of the following arrays are

Array	Size
$a = \begin{bmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{bmatrix}$	This is $3 \times 2$ matrix, containing 6 elements.
$b = [1 \ 2 \ 3 \ 4]$	This is a $1 \times 4$ array containing 4 elements, known as a <b>row vector</b> .
$c = \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}$	This is a $3 \times 1$ array containing 3 elements, known as a <b>column vector</b> .

Individual elements in an array are addressed by the array name followed by the row and column of the particular element. If the array is a row or column vector, then only one subscript is required. For example, in the above arrays  $a(2, 1)$  is 3 and  $c(2) = 2$ .

A MATLAB **variable** is a region of memory containing an array that is known by a user-specified name. The contents of the array may be used or modified at any time by including its name in an appropriate MATLAB command.

MATLAB variable names must begin with a letter, followed by any combination of letters, numbers, and the underscore (`_`) character. Only the first 63



**Figure 2.1** An array is a collection of data values organized into rows and columns.

characters are significant; if more than 63 are used, the remaining characters will be ignored. If two variables are declared with names that only differ in the 64th character, MATLAB will treat them as the same variable. MATLAB will issue a warning if it has to truncate a long variable name to 63 characters.<sup>1</sup>

### Programming Pitfalls

Make sure that your variable names are unique in the first 63 characters. Otherwise, MATLAB will not be able to tell the difference between them.

When writing a program, it is important to pick meaningful names for the variables. Meaningful names make a program *much* easier to read and to maintain. Names such as *day*, *month*, and *year* are quite clear even to a person seeing a program for the first time. Since spaces cannot be used in MATLAB variable names, underscore characters can be substituted to create meaningful names. For example, *exchange rate* might become `exchange_rate`.

### Good Programming Practice

Always give your variables descriptive and easy-to-remember names. For example, a currency exchange rate could be given the name `exchange_rate`. This practice will make your programs clearer and easier to understand.

It is also important to include a **data dictionary** in the header of any program that you write. A data dictionary lists the definition of each variable used in a program. The definition should include both a description of the contents of the item and the units in which it is measured. A data dictionary may seem unnecessary while the program is being written, but it is invaluable when you or another person have to go back and modify the program at a later time.

### Good Programming Practice

Create a data dictionary for each program to make program maintenance easier.

<sup>1</sup>Until MATLAB 6.5, the maximum number of significant characters in a variable name was 31. If you are writing a program that might be run on MATLAB 6.1 or earlier, be sure to limit your variable names to 31 characters or less. Otherwise, your program might work properly on MATLAB 7 but fail when it is executed on an earlier version.

The MATLAB language is case-sensitive, which means that uppercase and lowercase letters are not the same. Thus the variables `name`, `NAME`, and `Name` are all different in MATLAB. You must be careful to use the same capitalization every time that variable name is used. While it is not required, it is customary to use all lowercase letters for ordinary variable names.

### \* Good Programming Practice

Be sure to capitalize a variable exactly the same way each time that it is used. It is good practice to use only lower-case letters in variable names.

The most common types of MATLAB variables are `double` and `char`. Variables of type `double` consist of scalars or arrays of 64-bit double-precision floating-point numbers. They can hold real, imaginary, or complex values. The real and imaginary components of each variable can be positive or negative numbers in the range  $10^{-308}$  to  $10^{308}$ , with 15 to 16 significant decimal digits of accuracy. They are the principal numerical data type in MATLAB.

A variable of type `double` is automatically created whenever a numerical value is assigned to a variable name. The numerical values assigned to `double` variables can be real, imaginary, or complex. A real value is just a number. For example, the following statement assigns the real value 10.5 to the `double` variable `var`:

```
var = 10.5;
```

An imaginary value is defined by appending the letter `i` or `j` to a number. For example, `10i` and `-4j` are both imaginary values. The following statement assigns the imaginary value `4i` to the `double` variable `var`:

```
var = 4i;
```

A complex value has both a real and an imaginary component. It is created by adding a real and an imaginary number together. For example, the following statement assigns the complex value `10 + 10i` to variable `var`:

```
var = 10 + 10i;
```

Variables of type `char` consist of scalars or arrays of 16-bit values, each representing a single character. Arrays of this type are used to hold character strings. They are automatically created whenever a single character or a character string is assigned to a variable name. For example, the following statement creates a variable of type `char` whose name is `comment` and stores the specified string in it. After the statement is executed, `comment` will be a  $1 \times 26$  character array.

```
comment = 'This is a character string';
```



In a language such as C, the type of every variable must be explicitly declared in a program before it is used. These languages are said to be **strongly typed**. In contrast, MATLAB is a **weakly typed** language. Variables may be created at any time by simply assigning values to them, and the type of data assigned to the variable determines the type of variable that is created.

## 2.2 Initializing Variables in MATLAB

MATLAB variables are automatically created when they are initialized. There are three common ways to initialize a variable in MATLAB:

1. Assign data to the variable in an assignment statement.
2. Input data into the variable from the keyboard.
3. Read data from a file.

The first two ways are discussed here, and the third approach is discussed in Section 2.6.

### Initializing Variables in Assignment Statements

The simplest way to initialize a variable is to assign it one or more values in an **assignment statement**. An assignment statement has the general form

```
var = expression
```

where *var* is the name of a variable, and *expression* is a scalar constant, an array, or combination of constants, other variables, and mathematical operations (+, -, etc.). The value of the expression is calculated using the normal rules of mathematics, and the resulting values are stored in named variable. Simple examples of initializing variables with assignment statements include

```
var = 40i;
var2 = var/5;
array = [1 2 3 4];
x = 1; y = 2;
```

The first example creates a scalar variable of type *double*, and stores the imaginary number  $40i$  in it. The second example creates a scalar variable and stores the result of the expression  $\text{var}/5$  in it. The third example creates a variable and stores a 4-element row vector in it. The last example shows that multiple assignment statements can be placed on a single line, provided that they are separated by semicolons or commas. Note that if any of the variables had already existed when the statements were executed, their old contents would have been lost.

As the third example shows, variables can also be initialized with arrays of data. Such arrays are constructed using brackets (`[]`) and semicolons. All of the elements of an array are listed in **row order**. In other words, the values in each row

are listed from left to right, with the topmost row first and the bottom most row last. Individual values within a row are separated by blank spaces or commas, and the rows themselves are separated by semicolons or new lines. For example the following expressions are all legal arrays that can be used to initialize a variable:

---

```
3.4;
```

This expression creates a  $1 \times 1$  array (a scalar) containing the value 3.4. The brackets are not required in this case.

```
[1.0 2.0 3.0]
```

This expression creates a  $1 \times 3$  array containing the row vector [1 2 3].

```
[1.0; 2.0; 3.0]
```

This expression creates a  $3 \times 1$  array containing the column

vector  $\begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}$ .

```
[1, 2, 3; 4, 5, 6]
```

This expression creates a  $2 \times 3$  array containing

the matrix  $\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}$ .

```
1, 2, 3
```

This expression creates a  $2 \times 3$  array containing the matrix

```
4, 5, 6]
```

$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}$ . The end of the first line terminates the first row.

```
[]
```

This expression creates an **empty array**, which contains no rows and no columns. (Note that this is not the same as an array containing zeros.)

---

The number of elements in every row of an array must be the same, and the number of elements in every column must be the same. An expression such as

```
[1 2 3; 4 5];
```

is illegal because row 1 has three elements while row 2 has only two elements.

### Programming Pitfalls

The number of elements in every row of an array must be the same, and the number of elements in every column must be the same. Attempts to define an array with different numbers of elements in its rows or different numbers of elements in its columns will produce an error when the statement is executed.

---

The expressions used to initialize arrays can include algebraic operations and all or portions of previously defined arrays. For example, the assignment statements

```
a = [0 1+7];
b = [a(2) 7 a];
```

will define an array  $a = [0 \ 8]$  and an array  $b = [8 \ 7 \ 0 \ 8]$ .

Also, not all of the elements in an array need be defined when it is created. If a specific array element is defined and one or more of the elements before it are not, then the earlier elements will automatically be created and initialized to zero. For example, if  $c$  is not previously defined, the statement

```
c(2,3) = 5;
```

will produce the matrix  $c = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 5 \end{bmatrix}$ . Similarly, an array can be extended by specifying a value for an element beyond the currently defined size. For example, suppose that array  $d = [1 \ 2]$ . Then the statement

```
d(4) = 4;
```

will produce the array  $d = [1 \ 2 \ 0 \ 4]$ .

The semicolon at the end of each assignment statement shown above has a special purpose: it *suppresses the automatic echoing of values* that normally occurs whenever an expression is evaluated in an assignment statement. If an assignment statement is typed without the semicolon, the results of the statement are automatically displayed in the command window:

```
>> e = [1, 2, 3; 4, 5, 6]
e =
    1    2    3
    4    5    6
```

If a semicolon is added at the end of the statement, the echoing disappears. Echoing is an excellent way to quickly check your work, but it seriously slows down the execution of MATLAB programs. For that reason, we normally suppress echoing at all times.

However, echoing the results of calculations makes a great quick-and-dirty debugging tool. If you are not certain what the results of a specific assignment statement are, just leave off the semicolon from that statement, and the results will be displayed in the command window as the statement is executed.

### \* Good Programming Practice

Use a semicolon at the end of all MATLAB assignment statements to suppress echoing of assigned values in the Command Window. This greatly speeds program execution.

### \* Good Programming Practice

If you need to examine the results of a statement during program debugging, you may remove the semicolon from that statement only so that its results are echoed in the Command Window.

## Initializing with Shortcut Expressions

It is easy to create small arrays by explicitly listing each term in the array, but what happens when the array contains hundreds or even thousands of elements? It is just not practical to write out each element in the array separately!

MATLAB provides a special shortcut notation for these circumstances using the **colon operator**. The colon operator specifies a whole series of values by specifying the first value in the series, the stepping increment, and the last value in the series. The general form of a colon operator is

```
first:incr:last
```

where *first* is the first value in the series, *incr* is the stepping increment, and *last* is the last value in the series. If the increment is one, it may be omitted. For example, the expression `1:2:10` is a shortcut for a  $1 \times 5$  row vector containing the values 1, 3, 5, 7, and 9.

```
>> x = 1:2:10
x =
    1    3    5    7    9
```

With colon notation, an array can be initialized to have the hundred values  $\frac{\pi}{100}, \frac{2\pi}{100}, \frac{3\pi}{100}, \dots, \pi$  as follows:

```
angles = (0.01:0.01:1.00) * pi;
```

Shortcut expressions can be combined with the **transpose operator** (`'`) to initialize column vectors and more complex matrices. The transpose operator swaps the row and columns of any array that it is applied to. Thus the expression

```
f = [1:4]';
```

generates a 4-element row vector `[1 2 3 4]`, and then transposes it into the

4-element column vector  $f = \begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \end{bmatrix}$ . Similarly, the expressions

```
g = 1:4;
h = [g' g'];
```

will produce the matrix  $h = \begin{bmatrix} 1 & 1 \\ 2 & 2 \\ 3 & 3 \\ 4 & 4 \end{bmatrix}$ .

## Initializing with Built-In Functions

Arrays can also be initialized using built-in MATLAB functions. For example, the function `zeros` can be used to create an all-zero array of any desired size. There are several forms of the `zeros` function. If the function has a single scalar argument, it will produce a square array using the single argument as both the number of rows and the number of columns. If the function has two scalar arguments, the first argument will be the number of rows, and the second argument will be the number of columns. Since the `size` function returns two values containing the number of rows and columns in an array, it can be combined with the `zeros` function to generate an array of zeros that is the same size as another array. Some examples using the `zeros` function follow:

```
a = zeros(2);
b = zeros(2,3);
c = [1 2; 3 4];
d = zeros(size(c));
```

These statements generate the following arrays:

$$a = \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix} \quad b = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

$$c = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \quad d = \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix}$$

Similarly, the `ones` function can be used to generate arrays containing all ones, and the `eye` function can be used to generate arrays containing **identity matrices**, in which all on-diagonal elements are one, while all off-diagonal elements are zero. Table 2.1 contains list of common MATLAB functions useful for initializing variables.

## Initializing Variables with Keyboard Input

It is also possible to prompt a user and initialize a variable with data that he or she types directly at the keyboard. This option allows a script file to prompt a user for input data values while it is executing. The `input` function displays a prompt string in the Command Window and then waits for the user to type in a response. For example, consider the following statement:

```
my_val = input('Enter an input value:');
```

Table 2.1 MATLAB Functions Useful for Initializing Variables

Function	Purpose
<code>zeros(n)</code>	Generates an $n \times n$ matrix of zeros.
<code>zeros(n,m)</code>	Generates an $n \times m$ matrix of zeros.
<code>zeros(size(arr))</code>	Generates a matrix of zeros of the same size as <code>arr</code> .
<code>ones(n)</code>	Generates an $n \times n$ matrix of ones.
<code>ones(n,m)</code>	Generates an $n \times m$ matrix of ones.
<code>ones(size(arr))</code>	Generates a matrix of ones of the same size as <code>arr</code> .
<code>eye(n)</code>	Generates an $n \times n$ identity matrix.
<code>eye(n,m)</code>	Generates an $n \times m$ identity matrix.
<code>length(arr)</code>	Returns the length of a vector, or the longest dimension of a 2-D array.
<code>size(arr)</code>	Returns two values specifying the number of rows and columns in <code>arr</code> .

When this statement is executed, MATLAB prints out the string 'Enter an input value: ', and then waits for the user to respond. If the user enters a single number, it may just be typed in. If the user enters an array, it must be enclosed in brackets. In either case, whatever is typed will be stored in variable `my_val` when the return key is entered. If only the return key is entered, then an empty matrix will be created and stored in the variable.

If the `input` function includes the character 's' as a second argument, then the input data is returned to the user as a character string. Thus, the statement

```
» in1 = input('Enter data: ');
Enter data: 1.23
```

stores the value 1.23 into `in1`, while the statement

```
» in2 = input('Enter data: ','s');
Enter data: 1.23
```

stores the character string '1.23' into `in2`.

### Quiz 2.1

This quiz provides a quick check to see if you have understood the concepts introduced in Sections 2.1 and 2.2. If you have trouble with the quiz, reread the sections, ask your instructor, or discuss the material with a fellow student. The answers to this quiz are found in the back of the book.

1. What is the difference between an array, a matrix, and a vector?
2. Answer the following questions for the array shown below.

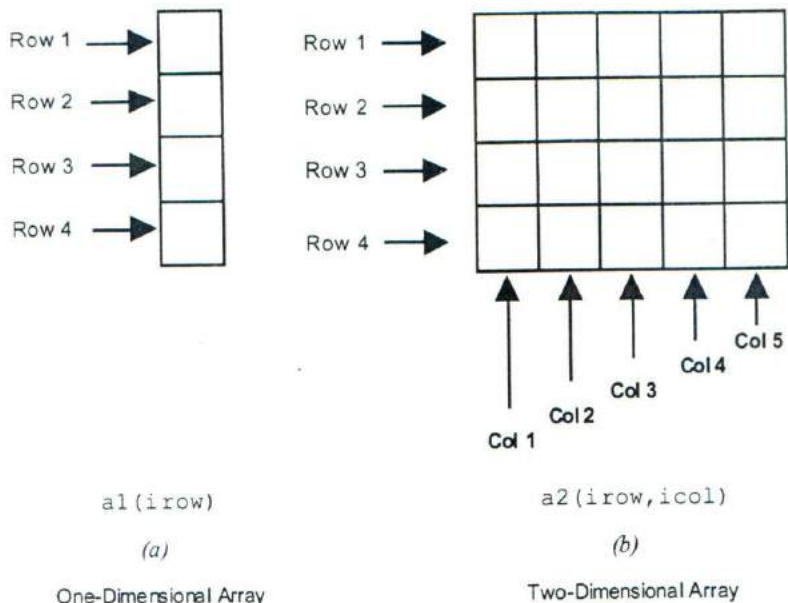
$$c = \begin{bmatrix} 1.1 & -3.2 & 3.4 & 0.6 \\ 0.6 & 1.1 & -0.6 & 3.1 \\ 1.3 & 0.6 & 5.5 & 0.0 \end{bmatrix}$$

- (a) What is the size of  $c$ ?
  - (b) What is the value of  $c(2, 3)$ ?
  - (c) List the subscripts of all elements containing the value 0.6.
3. Determine the size of the following arrays. Check your answers by entering the arrays into MATLAB and using the `whos` command or the Workspace Browser. Note that the later arrays may depend on the definitions of arrays given earlier in this exercise.
- (a) `u = [10 20*i 10+20];`
  - (b) `v = [-1; 20; 3];`
  - (c) `w = [1 0 -9; 2 -2 0; 1 2 3];`
  - (d) `x = [u' v];`
  - (e) `y(3,3) = -7;`
  - (f) `z = [zeros(4,1) ones(4,1) zeros(1,4)'];`
  - (g) `v(4) = x(2,1);`
4. What is the value of `w(2,1)` above?
  5. What is the value of `x(2,1)` above?
  6. What is the value of `y(2,1)` above?
  7. What is the value of `v(3)` after statement (g) is executed?
- 

## 2.3 Multidimensional Arrays

As we have seen, MATLAB arrays can have one or more dimensions. One-dimensional arrays can be visualized as a series of values laid out in a column, with a single subscript used to select the individual array elements (Figure 2.2a). Such arrays are useful to describe data that is a function of one independent variable, such as a series of temperature measurements made at fixed intervals of time.

Some types of data are functions of more than one independent variable. For example, we might wish to measure the temperature at five different locations at four different times. In this case, our 20 measurements could logically be grouped into five different columns of four measurements each, with a separate column for each location (Figure 2.2b). We will use two subscripts to access a given element in this array: the first one to select the row and the second one to select the column. Such arrays are called **two-dimensional arrays**. The number of elements in a two-dimensional array will be the product of the number of rows and the number of columns in the array.



**Figure 2.2** Representations of one- and two-dimensional arrays.

MATLAB allows us to create arrays with as many dimensions as necessary for any given problem. These arrays have one subscript for each dimension, and an individual element is selected by specifying a value for each subscript. The total number of elements in the array will be the product of the maximum value of each subscript. For example, the following two statements create a  $2 \times 3 \times 2$  array *c*:

```

» c(:,:,1)=[1 2 3; 4 5 6];
» c(:,:,2)=[7 8 9; 10 11 12];
» whos c

```

Name	Size	Bytes	Class
c	2x3x2	96	double array

This array contains 12 elements ( $2 \times 3 \times 2$ ). Its contents can be displayed just like any other array.

```

» c
c(:,:,1) =
     1     2     3
     4     5     6
c(:,:,2) =
     7     8     9
    10    11    12

```



## Storing Multidimensional Arrays in Memory

A two-dimensional array with  $m$  rows and  $n$  columns will contain  $m \times n$  elements, and these elements will occupy  $m \times n$  successive locations in the computer's memory. How are the elements of the array arranged in the computer's memory? MATLAB always allocates array elements in **column major order**. That is, MATLAB allocates the first column in memory, then the second, then the third, etc., until all of the columns have been allocated. Figure 2.3 illustrates this memory allocation scheme for a  $4 \times 3$  array  $a$ . As we can see, element  $a(1, 2)$  is really the fifth element allocated in memory. The order according to which elements are allocated in memory will become important when we discuss single-subscript addressing in the next section, and low-level I/O functions in Chapter 8.

This same allocation scheme applies to arrays with more than two dimensions. The first array subscript is incremented most rapidly, the second subscript is incremented less rapidly, etc., and the last subscript is incremented most slowly. For example, in a  $2 \times 2 \times 2$  array, the elements would be allocated in the following order: (1,1,1), (2,1,1), (1,2,1), (2,2,1), (1,1,2), (2,1,2), (1,2,2), (2,2,2).

## Accessing Multidimensional Arrays with One Dimension

One of MATLAB's peculiarities is that it will permit a user or programmer to treat a multidimensional array as though it were a one-dimensional array whose length is equal to the number of elements in the multidimensional array. If a multidimensional array is addressed with a single dimension, then the elements will be accessed in the order in which they were allocated in memory.

For example, suppose that we declare the  $4 \times 3$  element array  $a$  as follows:

```
>> a = [1 2 3; 4 5 6; 7 8 9; 10 11 12]
a =
     1     2     3
     4     5     6
     7     8     9
    10    11    12
```

Then the value of  $a(5)$  will be 2, which is the value of element  $a(1, 2)$ , because  $a(1, 2)$  was allocated fifth in memory.

Under normal circumstances, you should never use this feature of MATLAB. Addressing multidimensional arrays with a single subscript is a recipe for confusion.

### \* Good Programming Practice

Always use the proper number of dimensions when addressing a multidimensional array.

1	2	3
4	5	6
7	8	9
10	11	12

a

(a)

**Arrangement  
in Computer  
Memory**

1	a(1,1)
4	a(2,1)
7	a(3,1)
10	a(4,1)
2	a(1,2)
5	a(2,2)
8	a(3,2)
11	a(4,2)
3	a(1,3)
6	a(2,3)
9	a(3,3)
12	a(4,3)
.	
.	
.	

(b)

**Figure 2.3** (a) Data values for array a. (b) Layout of values in memory for array a.

## 2.4 Subarrays

It is possible to select and use subsets of MATLAB arrays as though they were separate arrays. To select a portion of an array, just include a list of all of the elements to be selected in the parentheses after the array name. For example, suppose that array `arr1` is defined as follows:

```
arr1 = [1.1 -2.2 3.3 -4.4 5.5];
```

Then `arr1(3)` is just the number 3.3, `arr1([1 4])` is the array `[1.1 -4.4]`, and `arr1(1:2:5)` is the array `[1.1 3.3 5.5]`.

For a two-dimensional array, a colon can be used in a subscript to select all of the values of that subscript. For example, suppose

```
arr2 = [1 2 3; -2 -3 -4; 3 4 5];
```

This statement would create an array `arr2` containing the values

$\begin{bmatrix} 1 & 2 & 3 \\ -2 & -3 & -4 \\ 3 & 4 & 5 \end{bmatrix}$ . With this definition, the subarray `arr2(1,:)` would be

`[1 2 3]`, and the subarray `arr2(:,1:2:3)` would be  $\begin{bmatrix} 1 & 3 \\ -2 & -4 \\ 3 & 5 \end{bmatrix}$ .

### The end Function

MATLAB includes a special function named `end` that is very useful for creating array subscripts. When used in an array subscript, `end` returns the highest value taken on by that subscript. For example, suppose that array `arr3` is defined as follows:

```
arr3 = [1 2 3 4 5 6 7 8];
```

Then `arr3(5:end)` would be the array `[5 6 7 8]`, and `array(end)` would be the value 8.

The value returned by `end` is always the highest value of a given subscript. If `end` appears in different subscripts, it can return different values within the same expression. For example, suppose that the  $3 \times 4$  array `arr4` is defined as follows:

```
arr4 = [1 2 3 4; 5 6 7 8; 9 10 11 12];
```

Then the expression `arr4(2:end,2:end)` would return the array

$\begin{bmatrix} 6 & 7 & 8 \\ 10 & 11 & 12 \end{bmatrix}$ . Note that the first `end` returned the value 3, while the second `end` returned the value 4!

## Using Subarrays on the Left-Hand Side of an Assignment Statement

It is also possible to use subarrays on the left-hand side of an assignment statement to update only some of the values in an array, as long as the **shape** (the number of rows and columns) of the values being assigned matches the shape of the subarray. If the shapes do not match, then an error will occur. For example, suppose that the  $3 \times 4$  array `arr4` is defined as follows:

```
» arr4 = [1 2 3 4; 5 6 7 8; 9 10 11 12]
arr4 =
     1     2     3     4
     5     6     7     8
     9    10    11    12
```

Then the following assignment statement is legal, since the expressions on both sides of the equal sign have the same shape ( $2 \times 2$ ):

```
» arr4(1:2,[1 4]) = [20 21; 22 23]
arr4 =
    20     2     3    21
    22     6     7    23
     9    10    11    12
```

Note that the array elements (1,1), (1,4), (2,1), and (2,4) were updated. In contrast, the following expression is illegal, because the two sides do not have the same shape.

```
» arr5(1:2,1:2) = [3 4]
??? In an assignment A(matrix,matrix) = B, the number of rows in B and the number of elements in the A row index matrix must be the same.
```

### Programming Pitfalls

For assignment statements involving subarrays, the *shapes of the subarrays on either side of the equal sign must match*. MATLAB will produce an error if they do not match.

There is a major difference in MATLAB between assigning values to a subarray and assigning values to an array. If values are assigned to a subarray, *only those values are updated, while all other values in the array remain unchanged*. On the other hand, if values are assigned to an array, *the entire contents of the*

array are deleted and replaced by the new values. For example, suppose that the  $3 \times 4$  array `arr4` is defined as follows:

```
» arr4 = [1 2 3 4; 5 6 7 8; 9 10 11 12]
arr4 =
     1     2     3     4
     5     6     7     8
     9    10    11    12
```

Then the following assignment statement replaces the *specified elements* of `arr4`:

```
» arr4(1:2,[1 4]) = [20 21; 22 23]
arr4 =
    20     2     3    21
    22     6     7    23
     9    10    11    12
```

In contrast, the following assignment statement replaces the *entire contents* of `arr4` with a  $2 \times 2$  array:

```
» arr4 = [20 21; 22 23]
arr4 =
    20    21
    22    23
```

### \* Good Programming Practice

Be sure to distinguish between assigning values to a subarray and assigning values to an array. MATLAB behaves differently in these two cases.

## Assigning a Scalar to a Subarray

A scalar value on the right-hand side of an assignment statement always matches the shape specified on the left-hand side. The scalar value is copied into every element specified on the left-hand side of the statement. For example, assume that the  $3 \times 4$  array `arr4` is defined as follows:

```
arr4 = [1 2 3 4; 5 6 7 8; 9 10 11 12];
```

Then the expression shown below assigns the value one to four elements of the array:

```
» arr4(1:2,1:2) = 1
arr4 =
     1     1     3     4
     1     1     7     8
     9    10    11    12
```

## 2.5 Special Values

MATLAB includes a number of predefined special values. These predefined values may be used at any time in MATLAB without initializing them first. A list of the most common predefined values is given in Table 2.2.

*These predefined values are stored in ordinary variables, so they can be overwritten or modified by a user. If a new value is assigned to one of the predefined variables, then that new value will replace the default one in all later calculations. For example, consider the following statements that calculate the circumference of a 10-cm circle:*

```
circ1 = 2 * pi * 10
pi = 3;
circ2 = 2 * pi * 10
```

In the first statement, `pi` has its default value of 3.14159... , so `circ1` is 62.8319, which is the correct circumference. The second statement redefines `pi` to be 3, so in the third statement `circ2` is 60. Changing a predefined value in the program has created an incorrect answer and has also introduced a subtle and hard-to-find bug. Imagine trying to locate the source of such a hidden error in a 10,000-line program!

**Table 2.2 Predefined Special Values**

Function	Purpose
<code>pi</code>	Contains $\pi$ to 15 significant digits
<code>i</code> , <code>j</code>	Contain the value $i(\sqrt{-1})$ .
<code>Inf</code>	This symbol represents machine infinity. It is usually generated as a result of a division by 0.
<code>NaN</code>	The symbol stands for Not-a-Number. It is the result of an undefined mathematical operation, such as the division of zero by zero.
<code>clock</code>	This special variable contains the current date and time in the form of a 6-element row vector containing the year, month, day, hour, minute, and second.
<code>date</code>	Contains the current data in a character strings format, such as 24-Nov-1998.
<code>eps</code>	This variable name is short for “epsilon”. It is the smallest difference between two numbers that can be represented on the computer.
<code>ans</code>	A special variable used to store the result of an expression if that result is not explicitly assigned to some other variable.

## Programming Pitfalls

*Never* redefine the meaning of a predefined variable in MATLAB. It is a recipe for disaster, producing subtle and hard-to-find bugs.

### Quiz 2.2

This quiz provides a quick check to see if you have understood the concepts introduced in Sections 2.3 through 2.5. If you have trouble with the quiz, reread the sections, ask your instructor, or discuss the material with a fellow student. The answers to this quiz are found in the back of the book.

1. Assume that array `c` is defined as shown, and determine the contents of the following sub-arrays:

$$c = \begin{bmatrix} 1.1 & -3.2 & 3.4 & 0.6 \\ 0.6 & 1.1 & -0.6 & 3.1 \\ 1.3 & 0.6 & 5.5 & 0.0 \end{bmatrix}$$

- (a) `c(2, :)`
  - (b) `c(:, end)`
  - (c) `c(1:2, 2:end)`
  - (d) `c(6)`
  - (e) `c(4:end)`
  - (f) `c(1:2, 2:4)`
  - (g) `c([1 3], 2)`
  - (h) `c([2 2], [3 3])`
2. Determine the contents of array `a` after the following statements are executed.
    - (a) `a = [1 2 3; 4 5 6; 7 8 9];`  
`a([3 1], :) = a([1 3], :);`
    - (b) `a = [1 2 3; 4 5 6; 7 8 9];`  
`a([1 3], :) = a([2 2], :);`
    - (c) `a = [1 2 3; 4 5 6; 7 8 9];`  
`a = a([2 2], :);`
  3. Determine the contents of array `a` after the following statements are executed.
    - (a) `a = eye(3, 3);`  
`b = [1 2 3];`  
`a(2, :) = b;`

```
(b) a = eye(3,3);
    b = [4 5 6];
    a(:,3) = b';

(c) a = eye(3,3);
    b = [7 8 9];
    a(3,:) = b([3 1 2]);

(d) a = eye(3,3);
    b = [7 8 9];
    a(3,:) = b([3 1 2]);
```

---

## 2.6 Displaying Output Data

There are several ways to display output data in MATLAB. This simplest way is one we have already seen—just leave the semicolon off of the end of a statement and it will be echoed to the Command Window. We will now explore a few other ways to display data.

### Changing the Default Format

When data is echoed in the Command Window, integer values are always displayed as integers, character values are displayed as strings, and other values are printed using a **default format**. The default format for MATLAB shows four digits after the decimal point, and it may be displayed in scientific notation with an exponent if the number is too large or too small. For example, the statements

```
x = 100.11
y = 1001.1
z = 0.00010011
```

produce the following output

```
x =
    100.1100
y =
    1.0011e+003
z =
    1.0011e-004
```

This default format can be changed using the **format** command. The format command changes the default format according to the values given in Table 2.3. The default format can be modified to display more significant digits of data, to force the display to be in scientific notation, to display data to two decimal digits, or to eliminate extra line feeds to make more data visible in the Command Window at a single time. Experiment with the commands in Table 2.3 for yourself.



**Table 2.3 Output Display Formats**

Format Command	Results	Example <sup>1</sup>
<code>format short</code>	4 digits after decimal (default format)	12.3457
<code>format long</code>	14 digits after decimal	12.345678901234567
<code>format short e</code>	5 digits plus exponent	1.2345e+001
<code>format short g</code>	5 total digits with or without exponent	12.345
<code>format long e</code>	15 digits plus exponent	1.2345678901234567e+011
<code>format long g</code>	15 total digits with or without exponent	12.3456789012345
<code>format bank</code>	"dollars and cents" format	12.35
<code>format hex</code>	hexadecimal display of bits	4028b0fcd32f707a
<code>format rat</code>	approximate ratio of small integers	1000/81
<code>format compact</code>	suppress extra line feeds	
<code>format loose</code>	restore extra line feeds	
<code>format +</code>	Only signs are printed	+

<sup>1</sup>The data value used for the example is 12.345678901234567 in all cases.

## The `disp` function

Another way to display data is with the `disp` function. The `disp` function accepts an array argument, and displays the value of the array in the Command Window. If the array is of type `char`, then the character string contained in the array is printed out.

This function is often combined with the functions `num2str` (convert a number to a string) and `int2str` (convert an integer to a string) to create messages to be displayed in the command window. For example, the following MATLAB statements will display "The value of pi = 3.1416" in the Command Window. The first statement creates a string array containing the message, and the second statement displays the message.

```
str = ['The value of pi = ' num2str(pi)];
disp (str);
```

## Formatted Output with the `fprintf` Function

An even more flexible way to display data is with the `fprintf` function. The `fprintf` function displays one or more values together with related text and lets the programmer control the way the displayed values appear. The general form of this function when it is used to print to the Command Window is:

```
fprintf (format, data)
```

**Table 2.4 Common Special Characters in fprintf Format Strings**

Format String	Results
%d	Display value as an integer.
%e	Display value in exponential format.
%f	Display value in floating point format.
%g	Display value in either floating point or exponential format, whichever is shorter.
\n	Skip to a new line.

where `format` is a string describing the way the data is to be printed, and `data` is one or more scalars or arrays to be printed. The `format` is a character string containing text to be printed plus special characters describing the format of the data. For example, the function

```
fprintf('The value of pi is %f \n',pi)
```

will print out 'The value of pi is 3.141593' followed by a line feed. The characters `%f` are called **conversion characters**; they indicate that the a value in the data list should be printed out in floating point format at that location in the format string. The characters `\n` are **escape characters**; they indicate that a line feed should be issued so that the following text starts on a new line. There are many types of conversion characters and escape characters that may be used in an `fprintf` function. A few of them are listed in Table 2.4, and a complete list can be found in Chapter 8.

It is also possible to specify the width of the field in which a number will be displayed and the number of decimal places to display. This is done by specifying the the width and precision after the `%` sign and before the `f`. For example, the function

```
fprintf('The value of pi is %6.2f \n',pi)
```

will print out 'The value of pi is 3.14' followed by a line feed. The conversion characters `%6.2f` indicate that the first data item in the function should be printed out in floating point format in a field six characters wide, including two digits after the decimal point.

The `fprintf` function has one very significant limitation: *it displays only the real portion of a complex value*. This limitation can lead to misleading results when calculations produce complex answers. In those cases, it is better to use the `disp` function to display answers.

For example, the following statements calculate a complex value `x` and display it using both `fprintf` and `disp`.

```
x = 2 * ( 1 - 2*i )^3;
str = ['disp: x = ' num2str(x)];
disp(str);
fprintf('fprintf: x = %8.4f\n',x);
```

The results printed out by these statements are

```
disp: x = -22+4i
fprintf: x = -22.0000
```

Note that the `fprintf` function ignored the imaginary part of the answer.

### Programming Pitfalls

The `fprintf` function displays only the *real* part of a complex number, which can produce misleading answers when working with complex values.

## 2.7 Data Files

There are many ways to load and save data files in MATLAB, most of which are addressed in Chapter 8. For the moment, we will consider only the `load` and `save` commands, which are the simplest ones to use.

The `save` command saves data from the current MATLAB workspace into a disk file. The most common form of this command is

```
save filename var1 var2 var3
```

where `filename` is the name of the file where the variables are saved, and `var1`, `var2`, etc. are the variables to be saved in the file. By default, the file name will be given the extent “`mat`,” and such data files are called MAT-files. If no variables are specified, then the entire contents of the workspace are saved.

MATLAB saves MAT-files in a special compact format that preserves many details, including the name and type of each variable, the size of each array, and all data values. A MAT-file created on any platform (PC, Mac, Unix, or Linux) can be read on any other platform, so MAT-files are a good way to exchange data between computers if both computers run MATLAB. Unfortunately, the MAT-file is in a format that cannot be read by other programs. If data must be shared with other programs, then the `-ascii` option should be specified, and the data values will be written to the file as ASCII character strings separated by spaces. However, the special information (e.g., variable names and types) is lost when the data is saved in ASCII format, and the resulting data file will be much larger.

For example, suppose the array `x` is defined as

```
x=[1.23 3.14 6.28; -5.1 7.00 0];
```

the command “`save x.dat x -ascii`” will produce a file named `x.dat` containing the following data:

```
1.2300000e+000 3.1400000e+000 6.2800000e+000
-5.1000000e+000 7.0000000e+000 0.0000000e+000
```

This data is in a format that can be read by spreadsheets or by programs written in other computer languages, so it makes it easy to share data between MATLAB programs and other applications.

### Good Programming Practice

If data must be exchanged between MATLAB and other programs, save the MATLAB data in ASCII format. If the data will only be used in MATLAB, save the data in MAT-file format.

MATLAB doesn't care what file extent is used for ASCII files. However, it is better for the user if a consistent naming convention is used, and an extent of ".dat" is a common choice for ASCII files.

### \* Good Programming Practice

Save ASCII data files with a ".dat" file extent to distinguish them from MAT-files, which have a ".mat" file extent.

The **load** command is the opposite of the **save** command. It loads data from a disk file into the current MATLAB workspace. The most common form of this command is

```
load filename
```

where *filename* is the name of the file to be loaded. If the file is a MAT-file, then all of the variables in the file will be restored, with the names and types the same as before. If a list of variables is included in the command, then only those variables will be restored. If the given *filename* has no extent, or if the file extent is `.mat`, then the **load** command will treat the file as a MAT-file.

MATLAB can load data created by other programs in space-separated ASCII format. If the given *filename* has any file extent other than `.mat`, then the **load** command will treat the file as an ASCII file. The contents of an ASCII file will be converted into a MATLAB array having the same name as the file (without the file extent) that the data was loaded from. For example, suppose that an ASCII data file named `x.dat` contains the following data:

```
1.23  3.14  6.28
-5.1  7.00  0
```

Then the command `load x.dat` will create a  $2 \times 3$  array named `x` in the current workspace, containing these data values.

The `load` statement can be forced to treat a file as a MAT-file by specifying the `-mat` option. For example, the statement

```
load -mat x.dat
```

would treat file `x.dat` as a MAT-file even though its file extent is not `.mat`. Similarly, the `load` statement can be forced to treat a file as an ASCII file by specifying the `-ascii` option. These options allow the user to load a file properly even if its file extent doesn't match the MATLAB conventions.

### Quiz 2.3

This quiz provides a quick check to see if you have understood the concepts introduced in Sections 2.6 and 2.7. If you have trouble with the quiz, reread the sections, ask your instructor, or discuss the material with a fellow student. The answers to this quiz are found in the back of the book.

- How would you tell MATLAB to display all real values in exponential format with 15 significant digits?
- What do the following sets of statements do? What is the output from them?

(a) 

```
radius = input('Enter circle radius:\n');
area = pi * radius^2;
str = ['The area is ' num2str(area)];
disp(str);
```

(b) 

```
value = int2str(pi);
disp(['The value is ' value '!']);
```

- What do the following sets of statements do? What is the output from them?

```
value = 123.4567e2;
fprintf('value = %e\n', value);
fprintf('value = %f\n', value);
fprintf('value = %g\n', value);
fprintf('value = %12.4f\n', value);
```

## 2.8 Scalar and Array Operations

Calculations are specified in MATLAB with an assignment statement, whose general form is

```
variable_name = expression;
```

The assignment statement calculates the value of the expression to the right of the equal sign, and *assigns* that value to the variable named on the left of the equal sign.

**Table 2.5 Arithmetic Operations Between Two Scalars**

Operation	Algebraic Form	MATLAB Form
Addition	$a + b$	$a + b$
Subtraction	$a - b$	$a - b$
Multiplication	$a \times b$	$a * b$
Division	$\frac{a}{b}$	$a / b$
Exponentiation	$a^b$	$a ^ b$

Note that the equal sign does not mean equality in the usual sense of the word. Instead, it means: *store the value of expression into location variable\_name*. For this reason, the equal sign is called the **assignment operator**. A statement like

```
ii = ii + 1;
```

is complete nonsense in ordinary algebra, but makes perfect sense in MATLAB. It means: take the current value stored in variable `ii`, add one to it, and store the result back into variable `ii`.

## Scalar Operations

The expression to the right of the assignment operator can be any valid combination of scalars, arrays, parentheses, and arithmetic operators. The standard arithmetic operations between two scalars are given in Table 2.5.

Parentheses may be used to group terms whenever desired. When parentheses are used, the expressions inside the parentheses are evaluated before the expressions outside the parentheses. For example, the expression  $2 ^ ((8+2)/5)$  is evaluated as shown below

$$\begin{aligned} 2 ^ ((8+2)/5) &= 2 ^ (10/5) \\ &= 2 ^ 2 \\ &= 4 \end{aligned}$$

## Array and Matrix Operations

MATLAB supports two types of operations between arrays, known as *array operations* and *matrix operations*. **Array operations** are operations performed between arrays on an **element-by-element basis**. That is, the operation is performed on corresponding elements in the two arrays. For example, if  $a = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$  and  $b = \begin{bmatrix} -1 & 3 \\ -2 & 3 \end{bmatrix}$ , then  $a + b = \begin{bmatrix} 0 & 5 \\ 1 & 5 \end{bmatrix}$ . Note that for these operations to work, *the number of rows and columns in both arrays must be the same*. If not, MATLAB will generate an error message.

Array operations may also occur between an array and a scalar. If the operation is performed between an array and a scalar, the value of the scalar is applied to every element of the array. For example, if  $a = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$ , then  $a + 4 = \begin{bmatrix} 5 & 6 \\ 7 & 8 \end{bmatrix}$ .

In contrast, **matrix operations** follow the normal rules of linear algebra, such as matrix multiplication. In linear algebra, the product  $c = a \times b$  is defined by the equation

$$c(i, j) = \sum_{k=1}^n a(i, k)b(k, j)$$

For example, if  $a = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$  and  $b = \begin{bmatrix} -1 & 3 \\ -2 & 1 \end{bmatrix}$ , then  $a \times b = \begin{bmatrix} -5 & 5 \\ -11 & 13 \end{bmatrix}$ . Note that for matrix multiplication to work, *the number of columns in matrix a must be equal to the number of rows in matrix b*.

MATLAB uses a special symbol to distinguish array operations from matrix operations. In the cases where array operations and matrix operations have a different definition, MATLAB uses a period before the symbol to indicate an array operation (for example, `.*`). A list of common array and matrix operations is given in Table 2.6 on page 49.

New users often confuse array operations and matrix operations. In some cases, substituting one for the other will produce an illegal operation, and MATLAB will report an error. In other cases, both operations are legal, and MATLAB will perform the wrong operation and come up with a wrong answer. The most common problem happens when working with square matrices. Both array multiplication and matrix multiplication are legal for two square matrices of the same size, but the resulting answers are totally different. Be careful to specify exactly what you want!

### Programming Pitfalls

Be careful to distinguish between array operations and matrix operations in your MATLAB code. It is especially common to confuse array multiplication with matrix multiplication.

#### Example 2.1

Assume that  $a$ ,  $b$ ,  $c$ , and  $d$  are defined as follows:

$$a = \begin{bmatrix} 1 & 0 \\ 2 & 1 \end{bmatrix} \quad b = \begin{bmatrix} -1 & 2 \\ 0 & 1 \end{bmatrix}$$

$$c = \begin{bmatrix} 3 \\ 2 \end{bmatrix} \quad d = 5$$

What is the result of each of the following expressions?

- |              |              |
|--------------|--------------|
| (a) $a + b$  | (e) $a + c$  |
| (b) $a .* b$ | (f) $a + d$  |
| (c) $a * b$  | (g) $a .* d$ |
| (d) $a * c$  | (h) $a * d$  |

SOLUTION

(a) This is array or matrix addition:  $a + b = \begin{bmatrix} 0 & 2 \\ 2 & 2 \end{bmatrix}$

(b) This is element-by-element array multiplication:  $a .* b = \begin{bmatrix} -1 & 0 \\ 0 & 1 \end{bmatrix}$

(c) This is matrix multiplication:  $a * b = \begin{bmatrix} -1 & 2 \\ -2 & 5 \end{bmatrix}$

(d) This is matrix multiplication:  $a * c = \begin{bmatrix} 3 \\ 8 \end{bmatrix}$

(e) This operation is illegal, since  $a$  and  $c$  have different numbers of columns.

(f) This is addition of an array to a scalar:  $a + d = \begin{bmatrix} 6 & 5 \\ 7 & 6 \end{bmatrix}$

(g) This is array multiplication:  $a .* d = \begin{bmatrix} 5 & 0 \\ 10 & 5 \end{bmatrix}$

(h) This is matrix multiplication:  $a * b = \begin{bmatrix} 5 & 0 \\ 10 & 5 \end{bmatrix}$

The matrix left division operation has a special significance that we must understand. A  $3 \times 3$  set of simultaneous linear equations takes the form

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + a_{13}x_3 &= b_1 \\ a_{21}x_1 + a_{22}x_2 + a_{23}x_3 &= b_2 \\ a_{31}x_1 + a_{32}x_2 + a_{33}x_3 &= b_3 \end{aligned} \quad (2-1)$$

which can be expressed as

$$Ax = B \quad (2-2)$$

where  $A = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix}$ ,  $B = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix}$ , and  $x = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}$ .

Equation (2-2) can be solved for  $x$  using linear algebra. The result is

$$x = A^{-1}B \quad (2-3)$$



Since the left division operator  $A \setminus B$  is defined to be  $\text{inv}(A) \times B$ , the left division operator solves a system of simultaneous equations in a single statement!

### ★ Good Programming Practice

Use the left division operator to solve systems of simultaneous equations.

**Table 2.6 Common Array and Matrix Operations**

Operation	MATLAB Form	Comments
Array Addition	$a + b$	Array addition and matrix addition are identical.
Array Subtraction	$a - b$	Array subtraction and matrix subtraction are identical.
Array Multiplication	$a .* b$	Element-by-element multiplication of $a$ and $b$ . Both arrays must be the same shape, or one of them must be a scalar.
Matrix Multiplication	$a * b$	Matrix multiplication of $a$ and $b$ . The number of columns in $a$ must equal the number of rows in $b$ .
Array Right Division	$a ./ b$	Element-by-element division of $a$ and $b$ : $a(i, j) / b(i, j)$ . Both arrays must be the same shape, or one of them must be a scalar.
Array Left Division	$a .\setminus b$	Element-by-element division of $a$ and $b$ , but with $b$ in the numerator: $b(i, j) / a(i, j)$ . Both arrays must be the same shape, or one of them must be a scalar.
Matrix Right Division	$a / b$	Matrix division defined by $a * \text{inv}(b)$ , where $\text{inv}(b)$ is the inverse of matrix $b$ .
Matrix Left Division	$a \setminus b$	Matrix division defined by $\text{inv}(a) * b$ , where $\text{inv}(a)$ is the inverse of matrix $a$ .
Array Exponentiation	$a .^ b$	Element-by-element exponentiation of $a$ and $b$ : $a(i, j) ^ b(i, j)$ . Both arrays must be the same shape, or one of them must be a scalar.

## 2.9 Hierarchy of Operations

Often, many arithmetic operations are combined into a single expression. For example, consider the equation for the distance traveled by an object starting from rest and subjected to a constant acceleration:

$$\text{distance} = 0.5 * \text{accel} * \text{time} ^ 2$$

There are two multiplications and an exponentiation in this expression. In such an expression, it is important to know the order in which the operations are evaluated. If exponentiation is evaluated before multiplication, this expression is equivalent to

$$\text{distance} = 0.5 * \text{accel} * (\text{time} ^ 2)$$

But if multiplication is evaluated before exponentiation, this expression is equivalent to

$$\text{distance} = (0.5 * \text{accel} * \text{time}) ^ 2$$

These two equations have different results, and we must be able to unambiguously distinguish between them.

To make the evaluation of expressions unambiguous, MATLAB has established a series of rules governing the hierarchy or order in which operations are evaluated within an expression. The rules generally follow the normal rules of algebra. The order in which the arithmetic operations are evaluated is given in Table 2.7.

**Table 2.7 Hierarchy of Arithmetic Operations**

Precedence	Operation
1	The contents of all parentheses are evaluated, starting from the innermost parentheses and working outward.
2	All exponentials are evaluated, working from left to right.
3	All multiplications and divisions are evaluated, working from left to right.
4	All additions and subtractions are evaluated, working from left to right.

**Example 2.2**

Variables  $a$ ,  $b$ ,  $c$ , and  $d$  have been initialized to the following values:

$$a = 3; \quad b = 2; \quad c = 5; \quad d = 3;$$

Evaluate the following MATLAB assignment statements:

- (a)  $\text{output} = a*b+c*d;$
- (b)  $\text{output} = a*(b+c)*d;$
- (c)  $\text{output} = (a*b)+(c*d);$
- (d)  $\text{output} = a^b*d;$
- (e)  $\text{output} = a^(b*d);$

**SOLUTION**

- |  |   |
|--|---|
| (a) Expression to evaluate:  | $\text{output} = a*b+c*d;$                          |
| Fill in numbers:   | $\text{output} = 3*2+5*3;$                          |
| First, evaluate multiplications<br>and divisions from left to right: | $\text{output} = 6+5*3;$<br>$\text{output} = 6+15;$ |
| Now evaluate additions:  | $\text{output} = 21$                                |
| (b) Expression to evaluate:  | $\text{output} = a*(b+c)*d;$                        |
| Fill in numbers:   | $\text{output} = 3*(2+5)*3;$                        |
| First, evaluate parentheses:   | $\text{output} = 3*7*3;$                            |
| Now, evaluate multiplications<br>and divisions from left to right:   | $\text{output} = 21*3;$<br>$\text{output} = 63;$    |
| (c) Expression to evaluate:  | $\text{output} = (a*b)+(c*d);$                      |
| Fill in numbers:   | $\text{output} = (3*2)+(5*3);$                      |
| First, evaluate parentheses:   | $\text{output} = 6+15;$                             |
| Now evaluate additions:  | $\text{output} = 21$                                |
| (d) Expression to evaluate:  | $\text{output} = a^b*d;$                            |
| Fill in numbers:   | $\text{output} = 3^2*3;$                            |
| Evaluate exponentials<br>from left to right:                         | $\text{output} = 9*3;$<br>$\text{output} = 27;$     |
| (e) Expression to evaluate:  | $\text{output} = a^(b*d);$                          |
| Fill in numbers:   | $\text{output} = 3^(2*3);$                          |
| First, evaluate parentheses:   | $\text{output} = 3^6;$                              |
| Now, evaluate exponential:   | $\text{output} = 729;$                              |

As we see in the foregoing example, the order in which operations are performed has a major effect on the final result of an algebraic expression.

It is important that every expression in a program be made as clear as possible. Any program of value must not only be written but also must be maintained and modified when necessary. You should always ask yourself: "Will I easily understand this expression if I come back to it in six months? Can another programmer look at my code and easily understand what I am doing?" If there is any doubt in your mind, use extra parentheses in the expression to make it as clear as possible.

### \* Good Programming Practice

Use parentheses as necessary to make your equations clear and easy to understand.

If parentheses are used within an expression, then the parentheses must be balanced. That is, there must be an equal number of open parentheses and close parentheses within the expression. It is an error to have more of one type than the other. Errors of this sort are usually typographical, and they are caught by the MATLAB interpreter when the command is executed. For example, the expression

$$(2 + 4) / 2)$$

produces an error when the expression is executed.

### Quiz 2.4

This quiz provides a quick check to see if you have understood the concepts introduced in Sections 2.8 and 2.9. If you have trouble with the quiz, reread the sections, ask your instructor, or discuss the material with a fellow student. The answers to this quiz are found in the back of the book.

1. Assume that  $a$ ,  $b$ ,  $c$ , and  $d$  are defined as follows, and calculate the results of the following operations if they are legal. If an operation is, explain why it is illegal.

$$a = \begin{bmatrix} 2 & 1 \\ -1 & 2 \end{bmatrix} \quad b = \begin{bmatrix} 0 & -1 \\ 3 & 1 \end{bmatrix}$$

$$c = \begin{bmatrix} 1 \\ 2 \end{bmatrix} \quad d = -3$$

- (a) `result = a .* c;`
- (b) `result = a * [c c];`
- (c) `result = a .* [c c];`

(d) `result = a + b * c;`

(e) `result = a + b .* c;`

2. Solve for  $x$  in the equation  $Ax = B$ , where  $A = \begin{bmatrix} 1 & 2 & 1 \\ 2 & 3 & 2 \\ -1 & 0 & 1 \end{bmatrix}$  and

$$B = \begin{bmatrix} 1 \\ 1 \\ 0 \end{bmatrix}.$$

## 2.10 Built-in MATLAB Functions

In mathematics, a **function** is an expression that accepts one or more input values and calculates a single result from them. Scientific and technical calculations usually require functions that are more complex than the simple addition, subtraction, multiplication, division, and exponentiation operations that we have discussed so far. Some of these functions are very common, and are used in many different technical disciplines. Others are rarer and specific to a single problem or a small number of problems. Examples of very common functions are the trigonometric functions, logarithms, and square roots. Examples of rarer functions include the hyperbolic functions, Bessel functions, and so forth. One of MATLAB's greatest strengths is that it comes with an incredible variety of built-in functions ready for use.

### Optional Results

Unlike mathematical functions, MATLAB functions can return *more than one result* to the calling program. The function `max` is an example of such a function. This function normally returns the maximum value of an input vector, but it can also return a second argument containing the location in the input vector where the maximum value was found. For example, the statement

```
maxval = max ([1 -5 6 -3])
```

returns the result `maxval = 6`. However, if two variables are provided to store results in, the function returns *both* the maximum value *and* the location of the maximum value.

```
[maxval index] = max ([1 -5 6 -3])
```

produces the results `maxval = 6` and `index = 3`.

### Using MATLAB Functions with Array Inputs

Many MATLAB functions are defined for one or more scalar inputs, and produce a scalar output. For example, the statement `y = sin(x)` calculates the sine of  $x$  and stores the result in  $y$ . If these functions receive an array of input values, then

they will calculate an array of output values on an element-by-element basis. For example, if  $x = [0 \text{ pi}/2 \text{ pi } 3*\text{pi}/2 \text{ } 2*\text{pi}]$ , then the statement

```
y = sin(x)
```

will produce the result  $y = [0 \ 1 \ 0 \ -1 \ 0]$ .

## Common MATLAB Functions

A few of the most common and useful MATLAB functions are shown in Table 2.8. These functions will be used in many examples and homework problems. If you need to locate a specific function not on this list, you can search for the function alphabetically or by subject using the MATLAB Help Browser.

Note that unlike most computer languages, many MATLAB functions work correctly for both real and complex inputs. MATLAB functions automatically calculate the correct answer, even if the result is imaginary or complex. For example, the function `sqrt(-2)` will produce a runtime error in languages such as C or Fortran. In contrast, MATLAB correctly calculates the imaginary answer:

```
>> sqrt(-2)
ans =
    0 + 1.4142i
```

## 2.11 Introduction to Plotting

MATLAB's extensive, device-independent plotting capabilities are one of its most powerful features. They make it very easy to plot any data at any time. To plot a data set, just create two vectors containing the  $x$  and  $y$  values to be plotted, and use the `plot` function.

For example, suppose that we wish to plot the function  $y = x^2 - 10x + 15$  for values of  $x$  between 0 and 10. It takes only three statements to create this plot. The first statement creates a vector of  $x$  values between 0 and 10 using the colon operator. The second statement calculates the  $y$  values from the equation (note that we are using array operators here so that this equation is applied to each  $x$  value on an element-by-element basis). Finally, the third statement creates the plot.

```
x = 0:1:10;
y = x.^2 - 10.*x + 15;
plot(x,y);
```

When the `plot` function is executed, MATLAB opens a Figure Window and displays the plot in that window. The plot produced by these statements is shown in Figure 2.4 on page 56.

Table 2.8 Common MATLAB Functions

Function	Description
<b>Mathematical Functions</b>	
<code>abs(x)</code>	Calculates $ x $ .
<code>acos(x)</code>	Calculates $\cos^{-1}x$ .
<code>angle(x)</code>	Returns the phase angle of the complex value $x$ , in radians.
<code>asin(x)</code>	Calculates $\sin^{-1}x$ .
<code>atan(x)</code>	Calculates $\tan^{-1}x$ .
<code>atan2(y,x)</code>	Calculates $\tan^{-1} \frac{y}{x}$ over all four quadrants of the circle (results in radians in the range $-\pi \leq \tan^{-1} \frac{y}{x} \leq \pi$ ).
<code>cos(x)</code>	Calculates $\cos x$ , with $x$ in radians.
<code>exp(x)</code>	Calculate $e^x$ .
<code>log(x)</code>	Calculates the natural logarithm $\log_e x$
<code>[value,index] = max(x)</code>	Returns the maximum value in vector $x$ , and optionally the location of that value.
<code>[value,index] = min(x)</code>	Returns the minimum value in vector $x$ , and optionally the location of that value.
<code>mod(x,y)</code>	Remainder or modulo function.
<code>sin(x)</code>	Calculates $\sin x$ , with $x$ in radians
<code>sqrt(x)</code>	Calculates the square root of $x$ .
<code>tan(x)</code>	Calculates $\tan x$ , with $x$ in radians
<b>Rounding Functions</b>	
<code>ceil(x)</code>	Rounds $x$ to the nearest integer towards positive infinity: <code>ceil(3.1) = 4</code> and <code>ceil(-3.1) = -3</code> .
<code>fix(x)</code>	Rounds $x$ to the nearest integer towards zero: <code>fix(3.1) = 3</code> and <code>fix(-3.1) = -3</code> .
<code>floor(x)</code>	Rounds $x$ to the nearest integer towards minus infinity: <code>floor(3.1) = 3</code> and <code>floor(-3.1) = -4</code> .
<code>round(x)</code>	Rounds $x$ to the nearest integer.
<b>String Conversion Functions</b>	
<code>char(x)</code>	Converts a matrix of numbers into a character string. For ASCII characters the matrix should contain numbers $\leq 127$
<code>double(x)</code>	Converts a character string into a matrix of numbers.
<code>int2str(x)</code>	Converts $x$ into an integer character string.
<code>num2str(x)</code>	Converts $x$ into a character string.
<code>str2num(s)</code>	Converts character string $s$ into a numeric array.

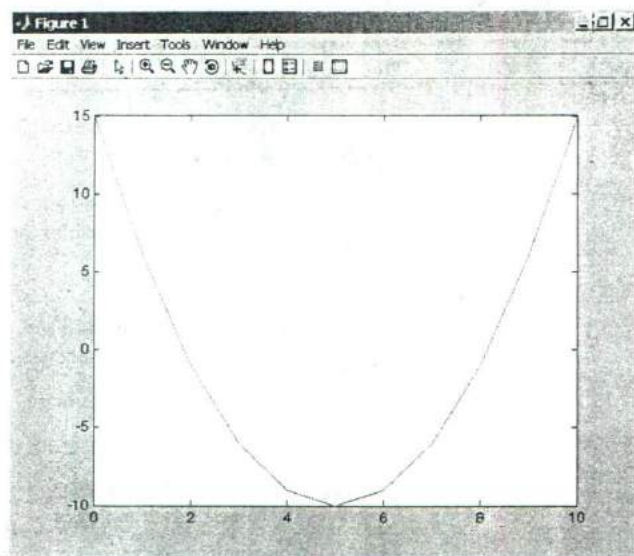


Figure 2.4 Plot of  $y = x^2 - 10x + 15$  from 0 to 10.

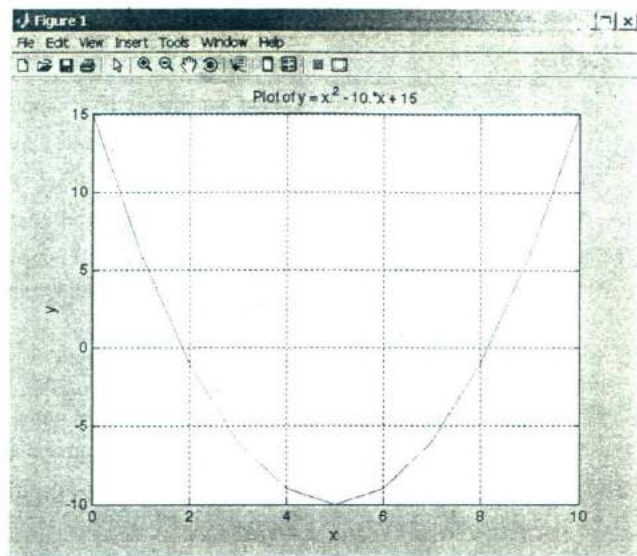
## Using Simple xy Plots

As we saw above, plotting is *very* easy in MATLAB. Any pair of vectors can be plotted versus each other as long as both vectors have the same length. However, the result is not a finished product, since there are no titles, axis labels, or grid lines on the plot.

Titles and axis labels can be added to a plot with the `title`, `xlabel`, and `ylabel` functions. Each function is called with a string containing the title or label to be applied to the plot. Grid lines can be added or removed from the plot with the `grid` command: `grid on` turns on grid lines, and `grid off` turns off grid lines. For example, the statements below generate a plot of the function  $y = x^2 - 10x + 15$  with titles, labels, and gridlines. The resulting plot is shown in Figure 2.5.

```
x = 0:1:10;
y = x.^2 - 10.*x + 15;
plot(x,y);
title('Plot of y = x.^2 - 10.*x + 15');
xlabel('x');
ylabel('y');
grid on;
```





**Figure 2.5** Plot of  $y = x^2 - 10x + 15$  with a title, axis labels, and gridlines.

## Printing a Plot

Once created, a plot may be printed on a printer with the `print` command, by clicking on the “print” icon in the Figure Window, or by selecting the “File/Print” menu option in the Figure Window.

The `print` command is especially useful because it can be included in a MATLAB program, allowing the program to automatically print graphical images. The form of the `print` command is:

```
print <options> <filename>
```

If no filename is included, this command prints a copy of the current figure on the system printer. If a filename is specified, the command prints a copy of the current figure to the specified file.

## Exporting a Plot as a Graphical Image

The `print` command can be used to save a plot as a graphical image by specifying appropriate options and a file name.

```
print <options> <filename>
```

There are many different options that specify the format of the output sent to a file. One very important option is `-dtiff`. This option specifies that the output

**Table 2.9** `print` Options to Create Graphics Files

Option	Description
<code>-deps</code>	Creates a monochrome encapsulated postscript image.
<code>-depvc</code>	Creates a color encapsulated postscript image.
<code>-djpeg</code>	Creates a JPEG image.
<code>-dpng</code>	Creates a Portable Network Graphic color image.
<code>-dtiff</code>	Creates a compressed TIFF image.

will be to a file in Tagged Image File Format (TIFF). Since this format can be imported into all of the important word processors on PC, Mac, Unix, and Linux platforms, it is a great way to include MATLAB plots in a document. The following command will create a TIFF image of the current figure and store it in a file called `my_image.tif`:

```
print -dtiff my_image.tif
```

Other options allow image files to be created in other formats. Some of the most important image file formats are given in Table 2.9.

In addition, the “File/Export” menu option on the Figure Window can be used to save a plot as a graphical image. In this case, the user selects the file name and the type of image from a standard dialog box (see Figure 2.6).

## Multiple Plots

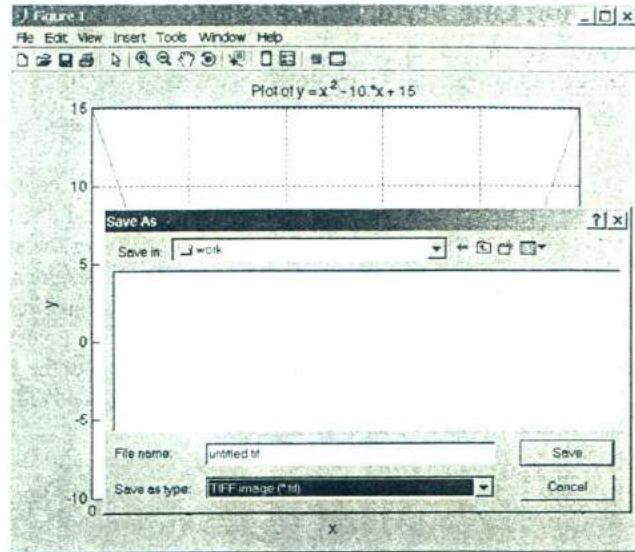
It is possible to plot multiple functions on the same graph by simply including more than one set of  $(x, y)$  values in the plot function. For example, suppose that we wanted to plot the function  $f(x) = \sin 2x$  and its derivative on the same plot. The derivative of  $f(x) = \sin 2x$  is:

$$\frac{d}{dt} \sin 2x = 2 \cos 2x \quad (2-4)$$

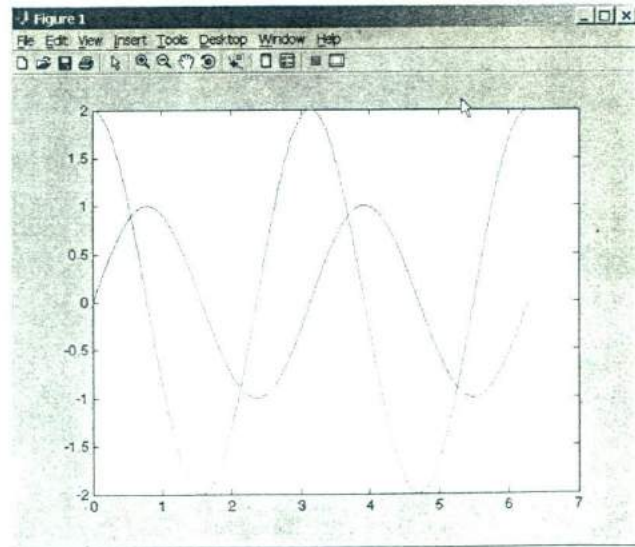
To plot both functions on the same axes, we must generate a set of  $x$  values and the corresponding  $y$  values for each function. Then to plot the functions, we would simply list both sets of  $(x, y)$  values in the plot function as shown below.

```
x = 0:pi/100:2*pi;
y1 = sin(2*x);
y2 = 2*cos(2*x);
plot(x,y1,x,y2);
```

The resulting plot is shown in Figure 2.7.



**Figure 2.6** Exporting a plot as an image file using the File/Export menu item.



**Figure 2.7** Plot of  $f(x) = \sin 2x$  and  $f(x) = 2 \cos 2x$  on the same axes.

## Line Color, Line Style, Marker Style, and Legends

MATLAB allows a programmer to select the color of a line to be plotted, the style of the line to be plotted, and the type of marker to be used for data points on the line. These traits may be selected using an attribute character string after the  $x$  and  $y$  vectors in the plot function.

The attribute character string can have up to three characters, with the first character specifying the color of the line, the second character specifying the style of the marker, and the last character specifying the style of the line. The characters for various colors, markers, and line styles are shown in Table 2.10.

The attribute characters may be mixed in any combination, and more than one attribute string may be specified if more than one pair of  $(x, y)$  vectors are included in a single `plot` function call. For example, the following statements will plot the function  $y = x^2 - 10x + 15$  with a dashed red line, and include the actual data points as blue circles.

```
x = 0:1:10;
y = x.^2 - 10.*x + 15;
plot(x,y, 'r--', x,y, 'bo');
```

Legends may be created with the `legend` function. The basic form of this function is

```
legend('string1', 'string2', ..., pos)
```

**Table 2.10** Table of Plot Colors, Marker Styles, and Line Styles

Color	Marker Style	Line Style	
y yellow	.	point	
m magenta	o	circle	
c cyan	x	x-mark	
r red	+	plus	
g green	*	star	
b blue	s	square	
w white	d	diamond	
k black	v	triangle (down)	
	^	triangle (up)	
	<	triangle (left)	
	>	triangle (right)	
	p	pentagram	
	h	hexagram	
	<none>	no marker	
		-	solid
		:	dotted
		-.	dash-dot
		--	dashed
		<none>	no line

**Table 2.11 Values of `pos` in the Legend Command**

Value	Legend Location
'NW'	Above and to the left
'NL'	Above top left corner
'NC'	Above center of top edge
'NR'	Above top right corner
'NE'	Above and to right
'TW'	At top and to left
'TL'	Top left corner
'TC'	At top center
'TR'	Top right corner
'TE'	At top and to right
'MW'	At middle and to left
'ML'	Middle left edge
'MC'	Middle and center
'MR'	Middle right edge
'ME'	At middle and to right
'BW'	At bottom and to left
'BL'	Bottom left corner
'BC'	At bottom center
'BR'	Bottom right corner
'BE'	At bottom and to right
'SW'	Below and to left
'SL'	Below bottom left corner
'SC'	Below center of bottom edge
'SR'	Below bottom right corner
'SE'	Below and to right

where `string1`, `string2`, etc. are the labels associated with the lines plotted, and `pos` is a string specifying where to place the legend. The possible values for `pos` are given in Table 2.11, and are shown graphically in Figure 2.8.

The command `legend off` will remove an existing legend<sup>2</sup>.

<sup>2</sup>Before MATLAB 7.0, the `pos` parameter took a number in the range 0–4 to specify the location of a legend. This usage is now obsolete but is still supported for backwards compatibility.

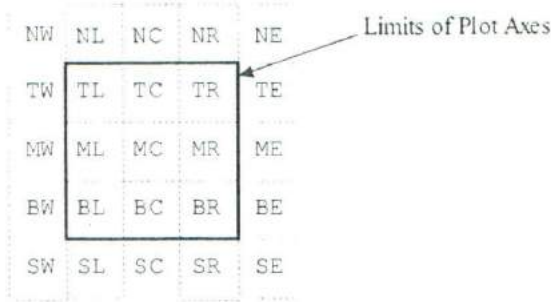


Figure 2.8 Possible locations for a plot legend.

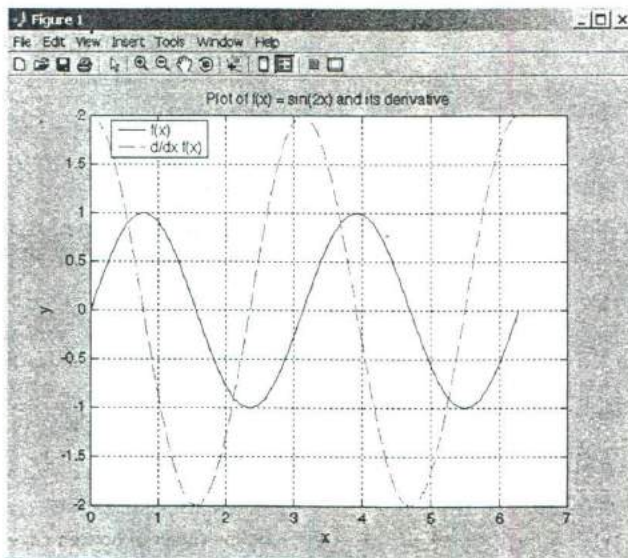


Figure 2.9 A complete plot with title, axis labels, legend, grid, and multiple line styles.

An example of a complete plot is shown in Figure 2.9, and the statements to produce that plot are shown below. They plot the function  $f(x) = \sin 2x$  and its derivative on the same axes, with a solid black line for  $f(x)$  and a dashed red line for its derivative. The plot includes a title, axis labels, a legend in the top left corner of the plot, and grid lines.

```
x = 0:pi/100:2*pi;
y1 = sin(2*x);
y2 = 2*cos(2*x);
```

```

plot(x,y1,'k-',x,y2,'b--');
title ('Plot of f(x) = sin(2x) and its deriva-
tive');
xlabel ('x');
ylabel ('y');
legend ('f(x)', 'd/dx f(x)', 't1')
grid on;

```

## Logarithmic Scales

It is possible to plot data on logarithmic scales as well as linear scales. There are four possible combinations of linear and logarithmic scales on the  $x$  and  $y$  axes, and each combination is produced by a separate function.

1. The `plot` function plots both  $x$  and  $y$  data on linear axes.
2. The `semilogx` function plots  $x$  data on logarithmic axes and  $y$  data on linear axes.
3. The `semilogy` function plots  $x$  data on linear axes and  $y$  data on logarithmic axes.
4. The `loglog` function plots both  $x$  and  $y$  data on logarithmic axes.

All of these functions have identical calling sequences—the only difference is the type of axis used to plot the data. Examples of each plot are shown in Figure 2.10.

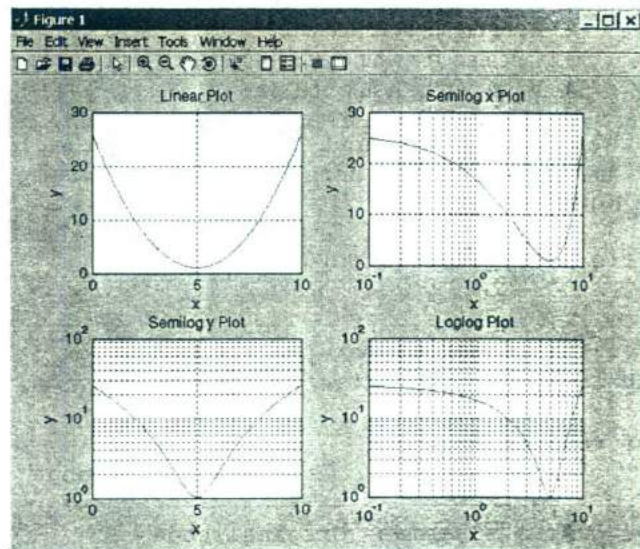


Figure 2.10 Comparison of linear, semilog  $x$ , semilog  $y$ , and log-log plots.

## 2.12 Examples

The following examples illustrate problem-solving with MATLAB.

### Example 2.3—Temperature Conversion

Design a MATLAB program that reads an input temperature in degrees Fahrenheit, converts it to an absolute temperature in kelvins, and writes out the result.

**SOLUTION** The relationship between temperature in degrees Fahrenheit ( $^{\circ}\text{F}$ ) and temperature in kelvins ( $\text{K}$ ) can be found in any physics textbook. It is

$$T \text{ (in kelvins)} = \left[ \frac{5}{9} T \text{ (in } ^{\circ}\text{F)} - 32.0 \right] + 273.15 \quad (2-5)$$

The physics books also give us sample values on both temperature scales, which we can use to check the operation of our program. Two such values are:

The boiling point of water	212 $^{\circ}$ F	373.15 K
The sublimation point of dry ice	-110 $^{\circ}$ F	194.26 K

Our program must perform the following steps:

1. Prompt the user to enter an input temperature in  $^{\circ}\text{F}$ .
2. Read the input temperature.
3. Calculate the temperature in kelvins from Equation (2-5).
4. Write out the result, and stop.

We will use function input to get the temperature in degrees Fahrenheit and function fprintf to print the answer. The resulting program is shown below.

```
% Script file: temp_conversion
%
% Purpose:
%   To convert an input temperature from degrees
%   Fahrenheit to an output temperature in kelvins.
%
% Record of revisions:
%   Date           Programmer      Description of change
%   ====          =====
%   01/03/04      S. J. Chapman    Original code
%
```



```

% Define variables:
%   temp_f   -- Temperature in degrees Fahrenheit
%   temp_k   -- Temperature in kelvins

% Prompt the user for the input temperature.
temp_f = input('Enter the temperature in degrees Fahrenheit:');
% Convert to kelvins.
temp_k = (5/9) * (temp_f - 32) + 273.15;
% Write out the result.
fprintf('%6.2f degrees Fahrenheit = %6.2f
        kelvins.\n', ... temp_f,temp_k);

```

To test the completed program, we will run it with the known input values given above. Note that user inputs appear in bold face below.

```

» temp_conversion
Enter the temperature in degrees Fahrenheit: 212
212.00 degrees Fahrenheit = 373.15 kelvins.
» temp_conversion
Enter the temperature in degrees Fahrenheit: -110
-110.00 degrees Fahrenheit = 194.26 kelvins.

```

The results of the program match the values from the physics book.

In the previous program, we echoed the input values and printed the output values together with their units. The results of this program make sense only if the units (degrees Fahrenheit and kelvins) are included together with their values. As a general rule, the units associated with any input value should always be printed along with the prompt that requests the value, and the units associated with any output value should always be printed along with that value.

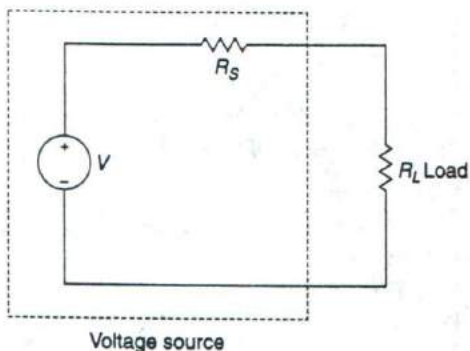
### ★ Good Programming Practice

Always include the appropriate units with any values that you read or write in a program.

The above program exhibits many of the good programming practices that we have described in this chapter. It includes a data dictionary defining the meanings of all of the variables in the program. It also uses descriptive variable names, and appropriate units are attached to all printed values.

**Example 2.4—Electrical Engineering: Maximum Power Transfer to a Load**

Figure 2.11 shows a voltage source  $V = 120\text{ V}$  with an internal resistance  $R_S$  of  $50\ \Omega$  supplying a load of resistance  $R_L$ . Find the value of load resistance  $R_L$  that will result in the maximum possible power being supplied by the source to the load. How much power be supplied in this case? Also, plot the power supplied to the load as a function of the load resistance  $R_L$ .



**Figure 2.11** A voltage source with a voltage  $V$  and an internal resistance  $R_S$  supplying a load of resistance  $R_L$ .

**SOLUTION**

In this program, we need to vary the load resistance  $R_L$  and compute the power supplied to the load at each value of  $R_L$ . The power supplied to the load resistance is given by the equation

$$P_L = I^2 R_L \quad (2-6)$$

where  $I$  is the current supplied to the load. The current supplied to the load can be calculated by Ohm's Law:

$$I = \frac{V}{R_{\text{TOT}}} = \frac{V}{R_S + R_L} \quad (2-7)$$

The program must perform the following steps:

1. Create an array of possible values for the load resistance  $R_L$ . The array will vary  $R_L$  from  $1\ \Omega$  to  $100\ \Omega$  in  $1\ \Omega$  steps.
2. Calculate the current for each value of  $R_L$ .

3. Calculate the power supplied to the load for each value of  $R_L$ .
4. Plot the power supplied to the load for each value of  $R_L$ , and determine the value of load resistance resulting in the maximum power.

The final MATLAB program is shown below.

```
% Script file: calc_power.m
%
% Purpose:
%   To calculate and plot the power supplied to a load as
%   as a function of the load resistance.
%
% Record of revisions:
%   Date           Programmer           Description of change
%   ====          =====
%   01/03/04      S. J. Chapman           Original code
%
% Define variables:
%   amps  -- Current flow to load (amps)
%   pl    -- Power supplied to load (watts)
%   rl    -- Resistance of the load (ohms)
%   rs    -- Internal resistance of the power source (ohms)
%   volts -- Voltage of the power source (volts)

% Set the values of source voltage and internal
resistance volts = 120;
rs = 50;

% Create an array of load resistances
rl = 1:1:100;

% Calculate the current flow for each resistance
amps = volts ./ ( rs + rl );

% Calculate the power supplied to the load
pl = (amps .^ 2) .* rl;

% Plot the power versus load resistance
plot(rl,pl);
title('Plot of power versus load resistance');
xlabel('Load resistance (ohms)');
ylabel('Power (watts)');
grid on;
```

When this program is executed, the resulting plot is shown in Figure 2.12. From this plot, we can see that the maximum power is supplied to the load when the load's resistance is  $50 \Omega$ . The power supplied to the load at this resistance is 72 watts.

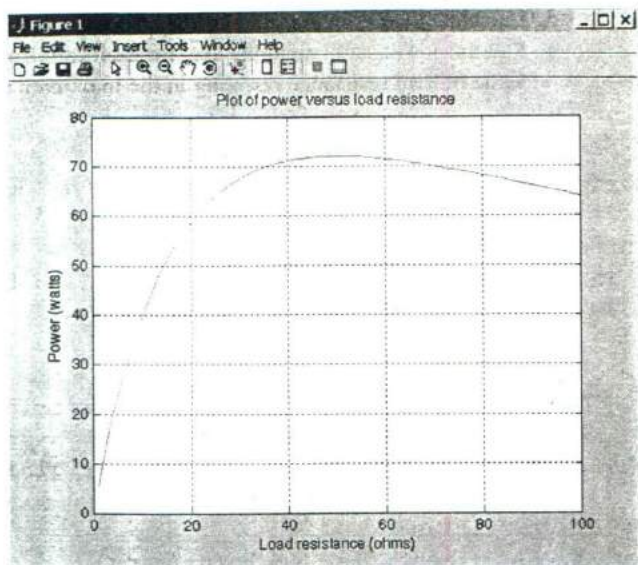


Figure 2.12 Plot of power supplied to load versus load resistance.

Note the use of the array operators `.*`, `.^`, and `./` in the above program. These operators cause the arrays `amps` and `p1` to be calculated on an element-by-element basis.

### Example 2.5—Carbon 14 Dating

A radioactive isotope of an element is a form of the element that is not stable. Instead, it spontaneously decays into another element over a period of time. Radioactive decay is an exponential process. If  $Q_0$  is the initial quantity of a radioactive substance at time  $t = 0$ , then the amount of that substance which will be present at any time  $t$  in the future is given by

$$Q(t) = Q_0 e^{-\lambda t} \quad (2-8)$$

where  $\lambda$  is the radioactive decay constant.

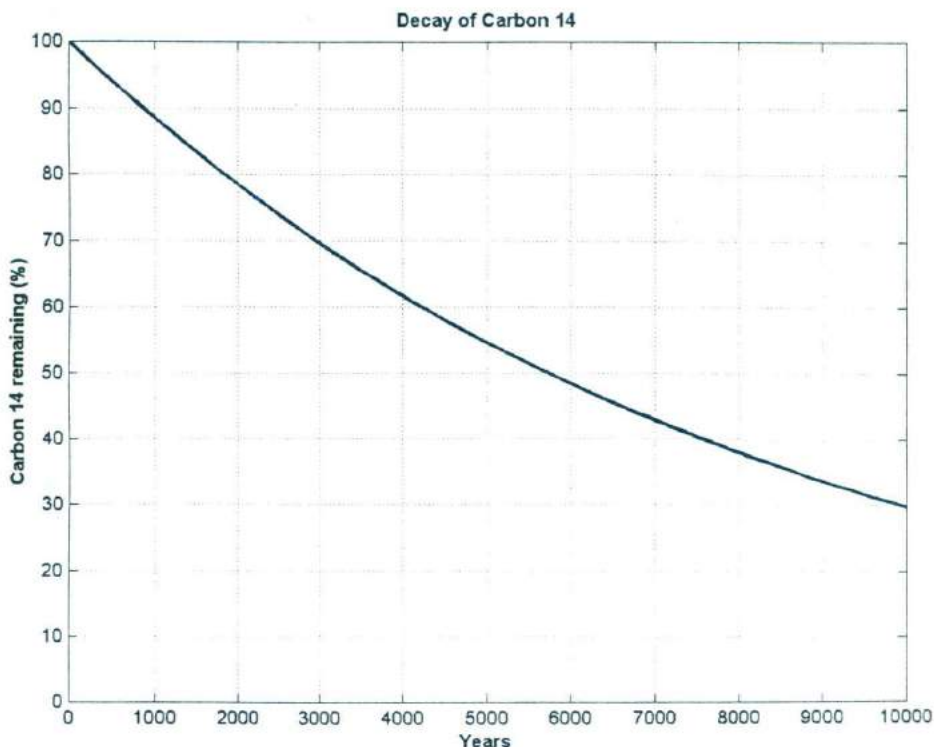
Because radioactive decay occurs at a known rate, it can be used as a clock to measure the time since the decay started. If we know the initial amount of the radioactive material  $Q_0$  present in a sample, and the amount of the material  $Q$  left

at the current time, we can solve for  $t$  in Equation (2-8) to determine how long the decay has been going on. The resulting equation is

$$t_{\text{decay}} = -\frac{1}{\lambda} \log_e \frac{Q}{Q_0} \quad (2-9)$$

Equation (2-9) has practical applications in many areas of science. For example, archaeologists use a radioactive clock based on carbon 14 to determine the time that has passed since a once-living thing died. Carbon 14 is continually taken into the body while a plant or animal is living, so the amount of it present in the body at the time of death is assumed to be known. The decay constant  $\lambda$  of carbon 14 is well known to be 0.00012097/year, so if the amount of carbon 14 remaining now can be accurately measured, then Equation (2-9) can be used to determine how long ago the living thing died. The amount of carbon 14 remaining as a function of time is shown in Figure 2.13.

Write a program that reads the percentage of carbon 14 remaining in a sample, calculates the age of the sample from it, and prints out the result with proper units.



**Figure 2.13** The radioactive decay of carbon 14 as a function of time. Notice that 50 percent of the original carbon 14 is left after about 5730 years have elapsed.

SOLUTION Our program must perform the following steps:

1. Prompt the user to enter the percentage of carbon 14 remaining in the sample.
2. Read in the percentage.
3. Convert the percentage into the fraction  $\frac{Q}{Q_0}$ .
4. Calculate the age of the sample in years using Equation (2-9).
5. Write out the result, and stop.

The resulting code is shown below.

```
% Script file: c14_date.m
%
% Purpose:
% To calculate the age of an organic sample from the
% percentage of the original carbon 14 remaining in
% the sample.
%
% Record of revisions:
%      Date          Programmer          Description of change
%      ====          =====          =====
%      01/03/04      S. J. Chapman          Original code
%
% Define variables:
% age      -- The age of the sample in years
% lamda    -- The radioactive decay constant for
%           carbon-14, in units of 1/years.
% percent  -- The percentage of carbon 14 remaining
%           at the time of the measurement
% ratio    -- The ratio of the carbon 14 remaining at
%           the time of the measurement to the
%           original amount of carbon 14.
%
% Set decay constant for carbon-14
lamda = 0.00012097;

% Prompt the user for the percentage of C-14 remaining.
percent = input('Enter the percentage of carbon 14 remaining:\n');

% Perform calculations
ratio = percent / 100;          % Convert to fractional ratio
age = (-1.0 / lamda) * log(ratio); % Get age in years

% Tell the user about the age of the sample.
string = ['The age of the sample is' num2str(age) ' years.'];
disp(string);
```

To test the completed program, we will calculate the time it takes for half of the carbon 14 to disappear. This time is known as the *half-life* of carbon 14.

```
>> c14_date
```

```
Enter the percentage of carbon 14 remaining:
```

```
50
```

```
The age of the sample is 5729.9097 years.
```

The *CRC Handbook of Chemistry and Physics* states that the half-life of carbon 14 is 5730 years, so output of the program agrees with the reference book.

## 2.13 Debugging MATLAB Programs

There is an old saying that the only sure things in life are death and taxes. We can add one more certainty to that list: if you write a program of any significant size, it won't work the first time you try it! Errors in programs are known as **bugs**, and the process of locating and eliminating them is known as **debugging**. Given that we have written a program and it is not working, how do we debug it?

Three types of errors are found in MATLAB programs. The first type of error is a **syntax error**. Syntax errors are errors in the MATLAB statement itself, such as spelling errors or punctuation errors. These errors are detected by the MATLAB compiler the first time an M-file is executed. For example, the statement

```
x = (y + 3) / 2);
```

contains a syntax error because it has unbalanced parentheses. If this statement appears in an M-file named `test.m`, the following message appears when `test` is executed.

```
>> test
```

```
??? x = (y + 3) / 2)
```

```
Missing operator, comma, or semi-colon.
```

```
Error in ==> d:\book\matlab\chap1\test.m
```

```
On line 2 ==>
```

The second type of error is the **run-time error**. A run-time error occurs when an illegal mathematical operation is attempted during program execution (for example, attempting to divide by 0). These errors cause the program to return `Inf` or `NaN`, which is then used in further calculations. The results of a program that contains calculations using `Inf` or `NaN` are usually invalid.

The third type of error is a **logical error**. Logical errors occur when the program compiles and runs successfully but produces the wrong answer.

The most common mistakes made during programming are *typographical errors*. Some typographical errors create invalid MATLAB statements. These errors produce syntax errors that are caught by the compiler. Other typographical errors occur in variable names. For example, the letters in some variable names might have been transposed, or an incorrect letter might be typed. The result will be a new variable, and MATLAB simply creates the new variable the first time that it is referenced. MATLAB cannot detect this type of error. Typographical errors can also produce logical errors. For example, if variables `ve11` and `ve12` are both used for velocities in the program, then one of them might be inadvertently used instead of the other one at some point. You must check for that sort of error by manually inspecting the code.

Sometimes a program will start to execute, but run-time errors or logical errors occur during execution. In this case, there is either something wrong with the input data or something wrong with the logical structure of the program. The first step in locating this sort of bug should be to *check the input data to the program*. Either remove semicolons from input statements or add extra output statements to verify that the input values are what you expect them to be.

If the variable names seem to be correct and the input data is correct, then you are probably dealing with a logical error. You should check each of your assignment statements.

1. If an assignment statement is very long, break it into several smaller assignment statements. Smaller statements are easier to verify.
2. Check the placement of parentheses in your assignment statements. It is a very common error to have the operations in an assignment statement evaluated in the wrong order. If you have any doubts as to the order in which the variables are being evaluated, add extra sets of parentheses to make your intentions clear.
3. Make sure that you have initialized all of your variables properly.
4. Be sure that any functions you use are in the correct units. For example, the input to trigonometric functions must be in units of radians, not degrees.

If you are still getting the wrong answer, add output statements at various points in your program to see the results of intermediate calculations. If you can locate the point where the calculations go bad, then you know just where to look for the problem, which is 95 percent of the battle.

If you still cannot find the problem after taking all of these steps, explain what you are doing to another student or to your instructor, and let them look at the code. It is very common for people to see just what they expect to see when they look at their own code. Another person can often quickly spot an error that you have overlooked time after time.



### \* Good Programming Practice

To reduce your debugging effort, make sure that during your program design you:

1. Initialize all variables.
2. Use parentheses to make the functions of assignment statements clear.

MATLAB includes a special debugging tool called a *symbolic debugger*. A symbolic debugger is a tool that allows you to walk through the execution of your program one statement at a time, and to examine the values of any variables at each step along the way. Symbolic debuggers allow you to see all of the intermediate results without having to insert a lot of output statements into your code. We will learn how to use MATLAB's symbolic debugger in Chapter 3.

## 2.14 Summary

In this chapter, we have presented many of the fundamental concepts required to write functional MATLAB programs. We learned about the basic types of MATLAB windows, the workspace, and how to get on-line help.

We introduced two data types: `double` and `char`. We also introduced assignment statements, arithmetic calculations, intrinsic functions, input/output statements, and data files.

The order in which MATLAB expressions are evaluated follows a fixed hierarchy, with operations at a higher level evaluated before operations at lower levels. The hierarchy of operations is summarized in Table 2.12.

The MATLAB language includes an extremely large number of built-in functions to help us solve problems. This list of functions is *much* richer than the list of functions found in other languages like Fortran or C, and it includes device-independent plotting capabilities. A few of the common intrinsic functions are

**Table 2.12 Hierarchy of Operations**

Precedence	Operation
1	The contents of all parentheses are evaluated, starting from the innermost parentheses and working outward.
2	All exponentials are evaluated, working from left to right.
3	All multiplications and divisions are evaluated, working from left to right.
4	All additions and subtractions are evaluated, working from left to right.

summarized in Table 2.8, and many others will be introduced throughout the remainder of the book. A complete list of all MATLAB functions is available through the on-line Help Browser.

## Summary of Good Programming Practice

Every MATLAB program should be designed so that another person who is familiar with MATLAB can easily understand it. This is very important, since a good program may be used for a long period of time. Over that time, conditions will change, and the program will need to be modified to reflect the changes. The program modifications may be done by someone other than the original programmer. The programmer making the modifications must understand the original program well before attempting to change it.

It is much harder to design clear, understandable, and maintainable programs than it is to simply write programs. To do so, a programmer must develop the discipline to properly document his or her work. In addition, the programmer must be careful to avoid known pitfalls along the path to good programs. The following guidelines will help you to develop good programs:

1. Use meaningful variable names whenever possible. Use names that can be understood at a glance, like `day`, `month`, and `year`.
2. Create a data dictionary for each program to make program maintenance easier.
3. Use only lower-case letters in variable names, so that there won't be errors due to capitalization differences in different occurrences of a variable name.
4. Use a semicolon at the end of all MATLAB assignment statements to suppress echoing of assigned values in the command window. If you need to examine the results of a statement during program debugging, you may remove the semicolon from that statement only.
5. If data must be exchanged between MATLAB and other programs, save the MATLAB data in ASCII format. If the data will be used only in MATLAB, save the data in MAT-file format.
6. Save ASCII data files with a "dat" file extent to distinguish them from MAT-files, which have a "mat" file extent.
7. Use parentheses as necessary to make your equations clear and easy to understand.
8. Always include the appropriate units with any values that you read or write in a program.

## MATLAB Summary

The following summary lists all of the MATLAB special symbols, commands, and functions described in this chapter, along with a brief description of each one.

---

**Special Symbols**


---

[ ]	Array constructor
( )	Forms subscripts
' '	Marks the limits of a character string
,	Comma, separates subscripts or matrix elements
;	1. Suppresses echoing in command window 2. Separates matrix rows 3. Separates assignment statements on a line
%	Marks the beginning of a comment
:	Colon operator, used to create shorthand lists
+	Array and matrix addition
-	Array and matrix subtraction
.*	Array multiplication
*	Matrix multiplication
./	Array right division
\	Array left division
/	Matrix right division
\	Matrix left division
.^	Array exponentiation
'	Transpose operator

---



---

**Commands and Functions**


---

...	Continues a MATLAB statement on the following line.
abs(x)	Calculates the absolute value of $x$
ans	Default variable used to store the result of expressions not assigned to another variable
acos(x)	Calculates the inverse cosine of $x$ . The resulting angle is in radians between 0 and $\pi$ .
asin(x)	Calculates the inverse sine of $x$ . The resulting angle is in radians between $-\pi/2$ and $\pi/2$ .
atan(x)	Calculates the inverse tangent of $x$ . The resulting angle is in radians between $-\pi/2$ and $\pi/2$ .
atan2(y, x)	Calculates the inverse tangent of $y/x$ , valid over the entire circle. The resulting angle is in radians between $-\pi$ and $\pi$ .
ceil(x)	Rounds $x$ to the nearest integer towards positive infinity: $\text{floor}(3.1) = 4$ and $\text{floor}(-3.1) = -3$ .

(continued)

**Commands and Functions (Continued)**

<code>char</code>	Converts a matrix of numbers into a character string. For ASCII characters the matrix should contain numbers $\leq 127$ .
<code>clock</code>	Current time
<code>cos(x)</code>	Calculates cosine of $x$ , where $x$ is in radians.
<code>date</code>	Current date
<code>disp</code>	Displays data in command window
<code>doc</code>	Open HTML Help Browser directly at a particular function description.
<code>double</code>	Converts a character string into a matrix of numbers.
<code>eps</code>	Represents machine precision
<code>exp(x)</code>	Calculates $e^x$ .
<code>eye(n,m)</code>	Generates an identity matrix
<code>fix(x)</code>	Rounds $x$ to the nearest integer towards zero: <code>fix(3.1) = 3</code> and <code>fix(-3.1) = -3</code> .
<code>floor(x)</code>	Rounds $x$ to the nearest integer towards minus infinity: <code>floor(3.1) = 3</code> and <code>floor(-3.1) = -4</code> .
<code>format +</code>	Print + and - signs only
<code>format bank</code>	Print in "dollars and cents" format
<code>format compact</code>	Suppress extra linefeeds in output
<code>format hex</code>	Print hexadecimal display of bits
<code>format long</code>	Print with 14 digits after the decimal
<code>format long e</code>	Print with 15 digits plus exponent
<code>format long g</code>	Print with 15 digits with or without exponent
<code>format loose</code>	Print with extra linefeeds in output
<code>format rat</code>	Print as an approximate ratio of small integers
<code>format short</code>	Print with 4 digits after the decimal
<code>format short e</code>	Print with 5 digits plus exponent
<code>format short g</code>	Print with 5 digits with or without exponent
<code>fprintf</code>	Print formatted information
<code>grid</code>	Add / remove a grid from a plot
<code>i</code>	$\sqrt{-1}$
<code>Inf</code>	Represents machine infinity ( $\infty$ )
<code>input</code>	Writes a prompt and reads a value from the keyboard
<code>int2str</code>	Converts $x$ into an integer character string.
<code>j</code>	$\sqrt{-1}$
<code>legend</code>	Adds a legend to a plot
<code>length(arr)</code>	Returns the length of a vector, or the longest dimension of a 2-D array.

**Commands and Functions (Continued)**


---

load	Load data from a file
log(x)	Calculates the natural logarithm of $x$ .
loglog	Generates a log-log plot
lookfor	Look for a matching term in the one-line MATLAB function descriptions.
max(x)	Returns the maximum value in vector $x$ , and optionally the location of that value.
min(x)	Returns the minimum value in vector $x$ , and optionally the location of that value.
mod(n,m)	Remainder or modulo function.
NaN	Represents not-a-number
num2str(x)	Converts $x$ into a character string.
ones(n,m)	Generates an array of ones
pi	Represents the number $\pi$
plot	Generates a linear $xy$ plot
print	Prints a Figure window
round(x)	Rounds $x$ to the nearest integer
save	Saves data from workspace into a file
semilogx	Generates a log-linear plot
semilogy	Generates a linear-log plot
sin(x)	Calculates sine of $x$ , where $x$ is in radians.
size	Get number of rows and columns in an array
sqrt	Calculates the square root of a number
str2num	Converts a character string into a number
tan(x)	Calculates tangent of $x$ , where $x$ is in radians.
title	Adds a title to a plot
zeros	Generate an array of zeros

---

**2.15 Exercises**

2.1 Answer the following questions for the array shown below.

$$\text{array1} = \begin{bmatrix} 1.1 & 0.0 & 2.1 & -3.5 & 6.0 \\ 0.0 & 1.1 & -6.6 & 2.8 & 3.4 \\ 2.1 & 0.1 & 0.3 & -0.4 & 1.3 \\ -1.4 & 5.1 & 0.0 & 1.1 & 0.0 \end{bmatrix}$$

- (a) What is the size of `array1`?  
 (b) What is the value of `array1(4,1)`?

- (c) What is the size and value of `array1(:, 1:2)`?  
 (d) What is the size and value of `array1([1 3], end)`?

**2.2** Are the following MATLAB variable names legal or illegal? Why?

- (a) `dog1`  
 (b) `ldog`  
 (c) `Do_you_know_the_way_to_san_jose`  
 (d) `_help`  
 (e) `What's_up?`

**2.3** Determine the size and contents of the following arrays. Note that the later arrays may depend on the definitions of arrays defined earlier in this exercise.

- (a) `a = 1:2:5;`  
 (b) `b = [a' a' a'];`  
 (c) `c = b(1:2:3, 1:2:3);`  
 (d) `d = a + b(2, :);`  
 (e) `w = [zeros(1,3) ones(3,1)' 3:5'];`  
 (f) `b([1 3], 2) = b([3 1], 2);`

**2.4** Assume that array `array1` is defined as shown, and determine the contents of the following sub-arrays:

$$\text{array1} = \begin{bmatrix} 1.1 & 0.0 & 2.1 & -3.5 & 6.0 \\ 0.0 & 1.1 & -6.6 & 2.8 & 3.4 \\ 2.1 & 0.1 & 0.3 & -0.4 & 1.3 \\ -1.4 & 5.1 & 0.0 & 1.1 & 0.0 \end{bmatrix}$$

- (a) `array1(3, :)`  
 (b) `array1(:, 3)`  
 (c) `array1(1:2:3, [3 3 4])`  
 (d) `array1([1 1], :)`

**2.5** Assume that `value` has been initialized to  $10\pi$ , and determine what is printed out by each of the following statements.

```
disp(['value = ' num2str(value)]);
disp(['value = ' int2str(value)]);
fprintf('value = %e\n', value);
fprintf('value = %f\n', value);
fprintf('value = %g\n', value);
fprintf('value = %12.4f\n', value);
```

**2.6** Assume that `a`, `b`, `c`, and `d` are defined as follows, and calculate the results of the following operations if they are legal. If an operation is, explain why it is illegal.

$$a = \begin{bmatrix} 2 & -2 \\ -1 & 2 \end{bmatrix}$$

$$b = \begin{bmatrix} 1 & -1 \\ 0 & 2 \end{bmatrix}$$

$$c = \begin{bmatrix} 1 \\ -2 \end{bmatrix}$$

$$d = \text{eye}(2)$$

- (a) result = a + b;  
 (b) result = a \* d;  
 (c) result = a .\* d;  
 (d) result = a \* c;  
 (e) result = a .\* c;  
 (f) result = a \ b;  
 (g) result = a .\ b;  
 (h) result = a .^ b;

**2.7** Evaluate each of the following expressions.

- (a) 11 / 5 + 6  
 (b) (11 / 5) + 6  
 (c) 11 / (5 + 6)  
 (d) 3 ^ 2 ^ 3  
 (e) 3 ^ (2 ^ 3)  
 (f) (3 ^ 2) ^ 3  
 (g) round(-11/5) + 6  
 (h) ceil(-11/5) + 6  
 (i) floor(-11/5) + 6

**2.8** Use MATLAB to evaluate each of the following expressions.

- (a) (3 - 5i)(-4 + 6i)  
 (b) cos<sup>-1</sup>(1.2)

**2.9** Solve the following system of simultaneous equations for x:

$$\begin{aligned} -2.0 x_1 + 5.0 x_2 + 1.0 x_3 + 3.0 x_4 + 4.0 x_5 - 1.0 x_6 &= 0.0 \\ 2.0 x_1 - 1.0 x_2 - 5.0 x_3 - 2.0 x_4 + 6.0 x_5 + 4.0 x_6 &= 1.0 \\ -1.0 x_1 + 6.0 x_2 - 4.0 x_3 - 5.0 x_4 + 3.0 x_5 - 1.0 x_6 &= -6.0 \\ 4.0 x_1 + 3.0 x_2 - 6.0 x_3 - 5.0 x_4 - 2.0 x_5 - 2.0 x_6 &= 10.0 \\ -3.0 x_1 + 6.0 x_2 + 4.0 x_3 + 2.0 x_4 - 6.0 x_5 + 4.0 x_6 &= -6.0 \\ 2.0 x_1 + 4.0 x_2 + 4.0 x_3 + 4.0 x_4 + 5.0 x_5 - 4.0 x_6 &= -2.0 \end{aligned}$$

**2.10 Position and Velocity of a Ball** If a stationary ball is released at a height  $h_0$  above the surface of the Earth with a vertical velocity  $v_0$ , the position and velocity of the ball as a function of time will be given by the equations

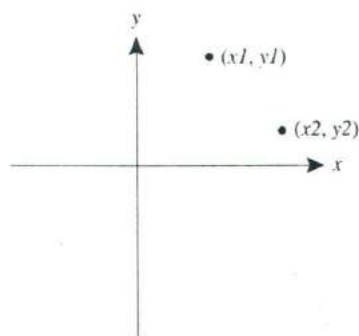
$$h(t) = \frac{1}{2}gt^2 + v_0t + h_0 \quad (2-10)$$

$$v(t) = gt + v_0 \quad (2-11)$$

where  $g$  is the acceleration due to gravity ( $-9.81 \text{ m/s}^2$ ),  $h$  is the height above the surface of the Earth (assuming no air friction), and  $v$  is the vertical component of velocity. Write a MATLAB program that prompts a user for the initial height of the ball in meters and velocity of the ball in meters per second, and plots the height and velocity as a function of time. Be sure to include proper labels in your plots.

- 2.11** The distance between two points  $(x_1, y_1)$  and  $(x_2, y_2)$  on a Cartesian coordinate plane is given by the equation

$$d = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2} \quad (2-12)$$



**Figure 2.14** Distance between two points on a Cartesian plane.

(See Figure 2.14.) Write a program to calculate the distance between any two points  $(x_1, y_1)$  and  $(x_2, y_2)$  specified by the user. Use good programming practices in your program. Use the program to calculate the distance between the points  $(2, 3)$  and  $(8, -5)$ .

- 2.12 Decibels** Engineers often measure the ratio of two power measurements in *decibels*, or dB. The equation for the ratio of two power measurements in decibels is

$$\text{dB} = 10 \log_{10} \frac{P_2}{P_1} \quad (2-13)$$

where  $P_2$  is the power level being measured, and  $P_1$  is some reference power level.

- (a) Assume that the reference power level  $P_1$  is 1 milliwatt, and write a program that accepts an input power  $P_2$  and converts it into dB with respect to the 1 mW reference level. (Engineers have a special unit for dB power levels with respect to a 1 mW reference: dBm.) Use good programming practices in your program.



- (b) Write a program that creates a plot of power in watts versus power in dBm with respect to a 1 mW reference level. Create both a linear  $xy$  plot and a log-linear  $xy$  plot.

**2.13 Hyperbolic cosine** The hyperbolic cosine function is defined by the equation

$$\cosh x = \frac{e^x + e^{-x}}{2} \quad (2-14)$$

Write a program to calculate the hyperbolic cosine of a user-supplied value  $x$ . Use the program to calculate the hyperbolic cosine of 3.0. Compare the answer that your program produces to the answer produced by the MATLAB intrinsic function `cosh(x)`. Also, use MATLAB to plot the function `cosh(x)`. What is the smallest value that this function can have? At what value of  $x$  does it occur?

**2.14 Energy Stored in a Spring** The force required to compress a linear spring is given by the equation

$$F = kx \quad (2-15)$$

where  $F$  is the force in newtons and  $k$  is the spring constant in newtons per meter. The potential energy stored in the compressed spring is given by the equation

	Spring 1	Spring 2	Spring 3	Spring 4
Force (N)	20	24	22	20
Spring constant $k$ (N/m)	500	600	700	800

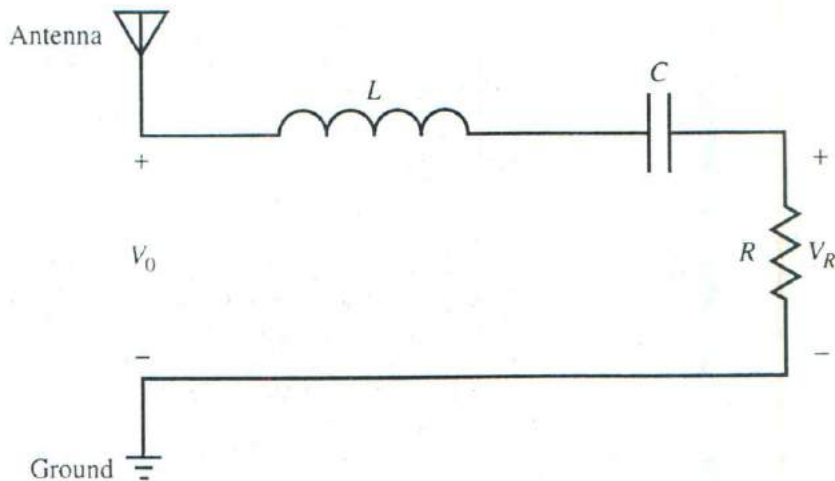
$$E = \frac{1}{2} kx^2 \quad (2-16)$$

where  $E$  is the energy in joules. The following information is available for four springs:

Determine the compression of each spring, and the potential energy stored in each spring. Which spring has the most energy stored in it?

**2.15 Radio Receiver** A simplified version of the front end of an AM radio receiver is shown in Figure 2.15. This receiver consists of an  $RLC$  tuned circuit containing a resistor, capacitor, and an inductor connected in series. The  $RLC$  circuit is connected to an external antenna and ground as shown in the picture.

The tuned circuit allows the radio to select a specific station out of all the stations transmitting on the AM band. At the resonant frequency of the circuit, essentially all of the signal  $V_0$  appearing at the antenna appears across the resistor, which represents the rest of the radio. In other words, the radio receives its strongest signal at the resonant frequency. The resonant frequency of the LC circuit is given by the equation



**Figure 2.15** A simplified version of the front end of an AM radio receiver.

$$f_0 = \frac{1}{2\pi\sqrt{LC}} \quad (2-17)$$

where  $L$  is inductance in henrys (H) and  $C$  is capacitance in farads (F). Write a program that calculates the resonant frequency of this radio set given specific values of  $L$  and  $C$ . Test your program by calculating the frequency of the radio when  $L = 0.1$  mH and  $C = 0.25$  nF.

- 2.16 Radio Receiver** The voltage across the resistive load in Figure 2.15 varies as a function of frequency according to Equation (2-18).

$$V_R = \frac{R}{\sqrt{R^2 + \left(\omega L - \frac{1}{\omega C}\right)^2}} \quad (2-18)$$

where  $\omega = 2\pi f$  and  $f$  is the frequency in hertz. Assume that  $L = 0.1$  mH,  $C = 0.25$  nF,  $R = 50 \Omega$ , and  $V_0 = 10$  mV.

- Plot the voltage on the resistive load as a function of frequency. At what frequency does the voltage on the resistive load peak? What is the voltage on the load at this frequency? This frequency is called the resonant frequency  $f_0$  of the circuit.
- If the frequency is changed to 10 percent greater than the resonant frequency, what is the voltage on the load? How selective is this radio receiver?
- At what frequencies will the voltage on the load drop to half of the voltage at the resonant frequency?



**Figure 2.16** An object moving in uniform circular motion due to the centripetal acceleration  $a$ .

- 2.17** Suppose that two signals were received at the antenna of the radio receiver described in the previous problem. One signal has a strength of 1 V at a frequency of 1000 kHz, and the other signal has a strength of 1 V at 950 kHz. How much power will the first signal supply to the resistive load  $R$ ? How much power will the second signal supply to the resistive load  $R$ ? Express the ratio of the power supplied by signal 1 to the power supplied by signal 2 in decibels. How much is the second signal enhanced or suppressed compared to the first signal?
- 2.18 Aircraft Turning Radius** An object moving in a circular path at a constant tangential velocity  $v$  is shown in Figure 2.16. The radial acceleration required for the object to move in the circular path is given by the Equation (2-19)

$$a = \frac{v^2}{r} \quad (2-19)$$

where  $a$  is the centripetal acceleration of the object in  $\text{m/s}^2$ ,  $v$  is the tangential velocity of the object in  $\text{m/s}$ , and  $r$  is the turning radius in meters. Suppose that the object is an aircraft, and answer the following questions about it:

- Suppose that the aircraft is moving at Mach 0.85, or 85% of the speed of sound. If the centripetal acceleration is 2 g, what is the turning radius of the aircraft? (Note: For this problem, you may assume that Mach 1 is equal to 340 m/s, and that  $1 \text{ g} = 9.81 \text{ m/s}^2$ ).
- Suppose that the speed of the aircraft increases to Mach 1.5. What is the turning radius of the aircraft now?

- (c) Plot the turning radius as a function of aircraft speed for speeds between Mach 0.5 and Mach 2.0, assuming that the acceleration remains 2 g.
- (d) Suppose that the maximum acceleration that the pilot can stand is 7 g. What is the minimum possible turning radius of the aircraft at Mach 1.5?
- (e) Plot the turning radius as a function of centripetal acceleration for accelerations between 2 g and 8 g, assuming a constant speed of Mach 0.85.

## Branching Statements and Program Design

---

In the previous chapter, we developed several complete working MATLAB programs. However, all of the programs were very simple, consisting of a series of MATLAB statements that were executed one after another in a fixed order. Such programs are called *sequential* programs. They read input data, process it to produce a desired answer, print out the answer, and quit. There is no way to repeat sections of the program more than once, and there is no way to selectively execute only certain portions of the program depending on values of the input data.

In the next two chapters, we will introduce a number of MATLAB statements that allow us to control the order in which statements are executed in a program. There are two broad categories of control statements: **branches**, which select specific sections of the code to execute, and **loops**, which cause specific sections of the code to be repeated. Branches will be discussed in this chapter, and loops will be discussed in Chapter 4.

With the introduction of branches and loops, our programs are going to become more complex, and it will get easier to make mistakes. To help avoid programming errors, we will introduce a formal program design procedure based on the technique known as top-down design. We will also introduce a common algorithm development tool known as pseudocode.

### 3.1 Introduction to Top-Down Design Techniques

---

Suppose that you are an engineer working in industry, and that you need to write a program to solve some problem. How do you begin?

When given a new problem, there is a natural tendency to sit down at a keyboard and start programming without “wasting” a lot of time thinking about the

problem first. It is often possible to get away with this “on the fly” approach to programming for very small problems, such as many of the examples in this book. In the real world, however, problems are larger, and a programmer attempting this approach will become hopelessly bogged down. For larger problems, it pays to completely think out the problem and the approach you are going to take to it before writing a single line of code.

We will introduce a formal program design process in this section, and then apply that process to every major application developed in the remainder of the book. For some of the simple examples that we will be doing, the design process will seem like overkill. However, as the problems that we solve get larger and larger, the process becomes more and more essential to successful programming.

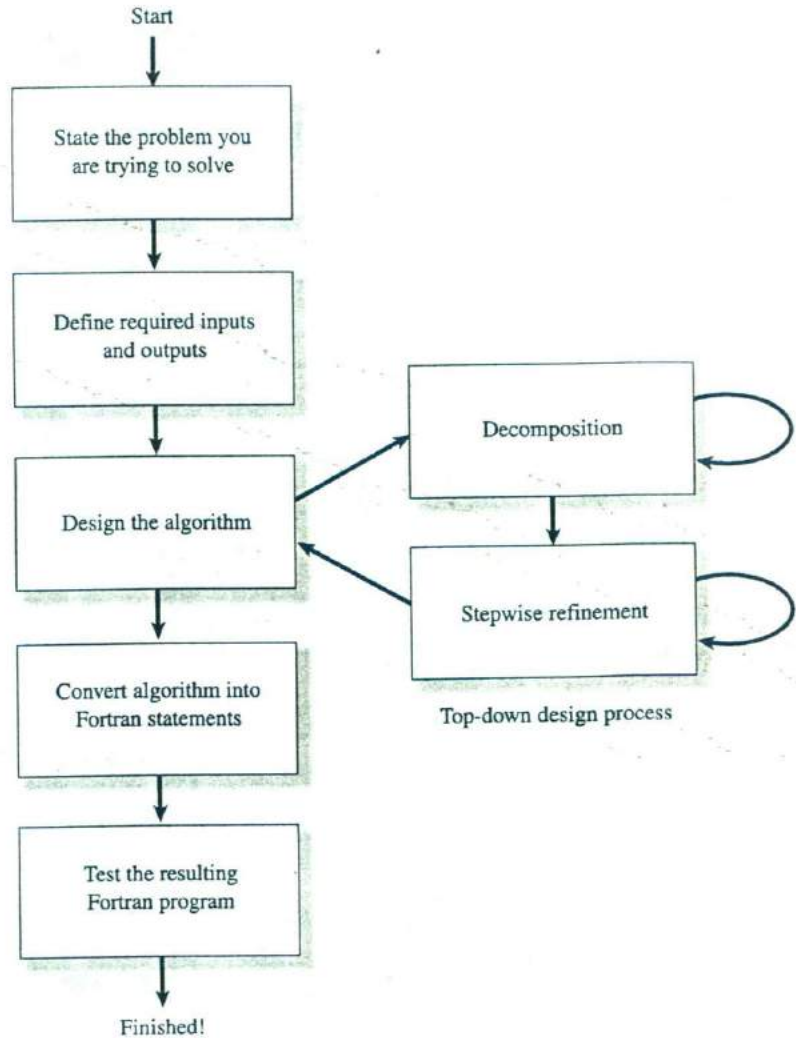
When I was an undergraduate, one of my professors was fond of saying, “Programming is easy. It’s knowing what to program that’s hard.” His point was forcefully driven home to me after I left university and began working in industry on larger-scale software projects. I found that the most difficult part of my job was to *understand the problem* I was trying to solve. Once I really understood the problem, it became easy to break the problem apart into smaller, more easily manageable pieces with well-defined functions, and then to tackle those pieces one at a time.

**Top-down design** is the process of starting with a large task and breaking it down into smaller, more easily understandable pieces (sub-tasks) which perform a portion of the desired task. Each sub-task may in turn be subdivided into smaller sub-tasks if necessary. Once the program is divided into small pieces, each piece can be coded and tested independently. We do not attempt to combine the sub-tasks into a complete task until each of the sub-tasks has been verified to work properly by itself.

The concept of top-down design is the basis of our formal program design process. We will now introduce the details of the process, which is illustrated in Figure 3.1. The steps involved are:

1. *Clearly state the problem that you are trying to solve.*

Programs are usually written to fill some perceived need, but that need may not be articulated clearly by the person requesting the program. For example, a user may ask for a program to solve a system of simultaneous linear equations. This request is not clear enough to allow a programmer to design a program to meet the need; he or she must first know much more about the problem to be solved. Is the system of equations to be solved real or complex? What is the maximum number of equations and unknowns that the program must handle? Are there any symmetries in the equations which might be exploited to make the task easier? The program designer will have to talk with the user requesting the program, and the two of them will have to come up with a clear statement of exactly what they are trying to accomplish. A clear statement of the problem will prevent misunderstandings, and it will also help the program designer to properly organize his or her thoughts. In the example we were describing, a proper statement of the problem might have been:



**Figure 3.1** The program design process used in this book.

Design and write a program to solve a system of simultaneous linear equations having real coefficients and with up to 20 equations in 20 unknowns.

2. Define the inputs required by the program and the outputs to be produced by the program.

The inputs to the program and the outputs produced by the program must be specified so that the new program will properly fit into the overall

processing scheme. In the above example, the coefficients of the equations to be solved are probably in some pre-existing order, and our new program needs to be able to read them in that order. Similarly, it needs to produce the answers required by the programs that may follow it in the overall processing scheme, and to write out those answers in the format needed by the programs following it.

3. *Design the algorithm that you intend to implement in the program.*

An **algorithm** is a step-by-step procedure for finding the solution to a problem. It is at this stage in the process that top-down design techniques come into play. The designer looks for logical divisions within the problem, and divides it up into sub-tasks along those lines. This process is called *decomposition*. If the sub-tasks are themselves large, the designer can break them up into even smaller sub-sub-tasks. This process continues until the problem has been divided into many small pieces, each of which does a simple, clearly understandable job.

After the problem has been decomposed into small pieces, each piece is further refined through a process called *stepwise refinement*. In stepwise refinement, a designer starts with a general description of what the piece of code should do, and then defines the functions of the piece in greater and greater detail until they are specific enough to be turned into MATLAB statements. Stepwise refinement is usually done with **pseudocode**, which will be described in the next section.

It is often helpful to solve a simple example of the problem by hand during the algorithm development process. If the designer understands the steps that he or she went through in solving the problem by hand, then he or she will be in better able to apply decomposition and stepwise refinement to the problem.

4. *Turn the algorithm into MATLAB statements.*

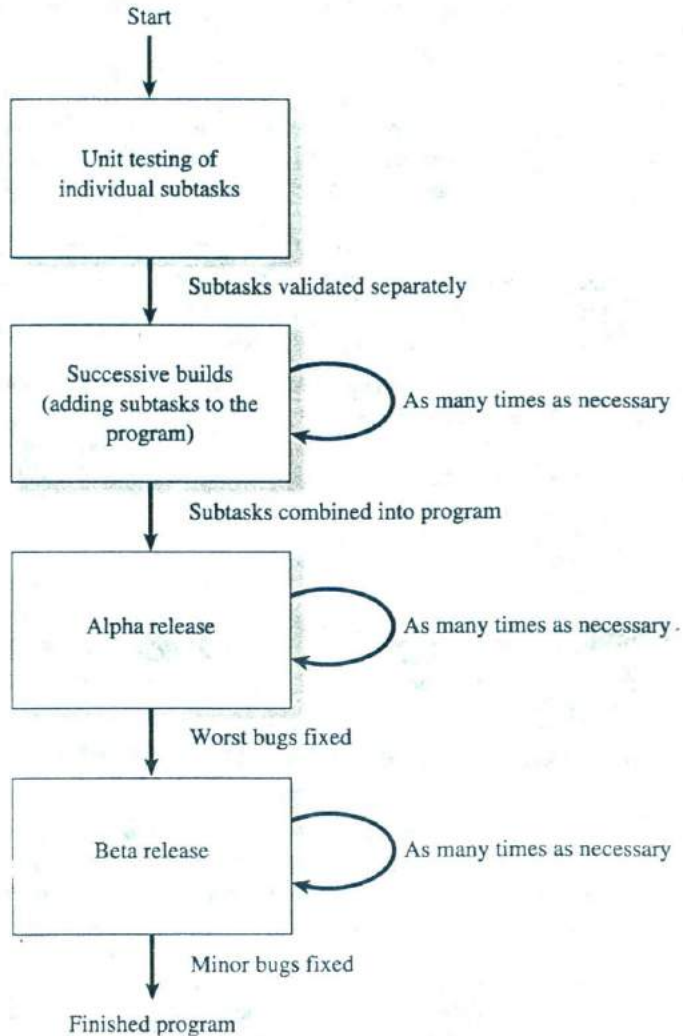
If the decomposition and refinement process was carried out properly, this step will be very simple. All the programmer will have to do is to replace pseudocode with the corresponding MATLAB statements on a one-for-one basis.

5. *Test the resulting MATLAB program.*

This step is the real killer. The components of the program must first be tested individually, if possible, and then the program as a whole must be tested. When testing a program, we must verify that it works correctly for *all legal input data sets*. It is very common for a program to be written, tested with some standard data set, and released for use, only to find that it produces the wrong answers (or crashes) with a different input data set. If the algorithm implemented in a program includes different branches, we must test all of the possible branches to confirm that the program operates correctly under every possible circumstance.



Large programs typically go through a series of tests before they are released for general use (see Figure 3.2). The first stage of testing is sometimes called **unit testing**. During unit testing, the individual sub-tasks of the program are tested separately to confirm that they work correctly. After the unit testing is completed, the program goes through a series of *builds* during which the individual sub-tasks are combined to produce the final program. The first build of the program



**Figure 3.2** A typical testing process for a large program.

typically includes only a few of the sub-tasks. It is used to check the interactions among those sub-tasks and the functions performed by the combinations of the sub-tasks. In successive builds, more and more sub-tasks are added, until the entire program is complete. Testing is performed on each build, and any errors (bugs) that are detected are corrected before moving on to the next build.

Testing continues even after the program is complete. The first complete version of the program is usually called the **alpha release**. It is exercised by the programmers and others very close to them in as many different ways as possible, and the bugs discovered during the testing are corrected. When the most serious bugs have been removed from the program, a new version called the **beta release** is prepared. The beta release is normally given to “friendly” outside users who have a need for the program in their normal day-to-day jobs. These users put the program through its paces under many different conditions and with many different input data sets, and they report any bugs that they find to the programmers. When those bugs have been corrected, the program is ready to be released for general use.

Because the programs in this book are fairly small, we will not go through the sort of extensive testing described above. However, we will follow the basic principles in testing all of our programs.

The program design process may be summarized as follows:

1. Clearly state the problem that you are trying to solve.
2. Define the inputs required by the program and the outputs to be produced by the program.
3. Design the algorithm that you intend to implement in the program.
4. Turn the algorithm into MATLAB statements.
5. Test the MATLAB program.

### \* Good Programming Practice

Follow the steps of the program design process to produce reliable, understandable MATLAB programs.

In a large programming project, the time actually spent programming is surprisingly small. In his book *The Mythical Man-Month*<sup>1</sup>, Frederick P. Brooks, Jr. suggests that in a typical large software project, 1/3 of the time is spent planning what to do (steps 1 through 3), 1/6 of the time is spent actually writing the program (step 4), and fully 1/2 of the time is spent in testing and debugging the program! Clearly, anything that we can do to reduce the testing and debugging time will be very helpful. We can best reduce the testing and debugging time by doing a very

<sup>1</sup>*The Mythical Man-Month, Anniversary Edition*, by Frederick P. Brooks Jr., Addison-Wesley, 1995.

careful job in the planning phase, and by using good programming practices. Good programming practices will reduce the number of bugs in the program, and will make the ones that do creep in easier to find.

## 3.2 Use of Pseudocode

As a part of the design process, it is necessary to describe the algorithm that you intend to implement. The description of the algorithm should be in a standard form that is easy for both you and other people to understand, and the description should aid you in turning your concept into MATLAB code. The standard forms that we use to describe algorithms are called **constructs** (or sometimes structures), and an algorithm described using these constructs is called a structured algorithm. When the algorithm is implemented in a MATLAB program, the resulting program is called a **structured program**.

The constructs used to build algorithms can be described in a special way called pseudocode. **Pseudocode** is a hybrid mixture of MATLAB and English. It is structured like MATLAB, with a separate line for each distinct idea or segment of code, but the descriptions on each line are in English. Each line of the pseudocode should describe its idea in plain, easily understandable English. Pseudocode is very useful for developing algorithms, since it is flexible and easy to modify. It is especially useful since pseudocode can be written and modified with the same editor or word processor used to write the MATLAB program—no special graphical capabilities are required.

For example, the pseudocode for the algorithm in Example 2.3 is:

```
Prompt user to enter temperature in degrees Fahrenheit
Read temperature in degrees Fahrenheit (temp_f)
temp_k in kelvins <- (5/9) * (temp_f - 32) + 273.15
Write temperature in kelvins
```

Notice that a left arrow (<-) is used instead of an equal sign (=) to indicate that a value is stored in a variable, since this avoids any confusion between assignment and equality. Pseudocode is intended to aid you in organizing your thoughts before converting them into MATLAB code.

## 3.3 The Logical Data Type

The logical data type<sup>2</sup> is a special type of data that can have one of only two possible values: true or false. These values are produced by the two special functions true and false. They are also produced by two types of MATLAB operators: relational operators and logic operators.

<sup>2</sup>The logical data type was introduced in MATLAB 6.5.

Logical values are stored in a single byte of memory, so they take up much less space than numbers, which usually occupy 8 bytes.

The operation of many MATLAB branching constructs is controlled by logical variables or expressions. If the result of a variable or expression is true, then one section of code is executed. If not, then a different section of code is executed.

To create a logical variable, just assign a logical value to it in an assignment statement. For example, the statement

```
a1 = true;
```

creates a logical variable `a1` containing the logical value `true`. If this variable is examined with the `whos` command, we can see that it has the logical data type:

```
» whos a1
Name      Size      Bytes   Class
a1        1x1         1    logical array
```

Unlike programming languages such as Java, C++, and Fortran, it is legal in MATLAB to mix numerical and logical data in expressions. If a logical value is used in a place where a numerical value is expected, `true` values are converted to 1 and `false` values are converted to 0, and then used as numbers. If a numerical value is used in a place where a logical value is expected, non-zero values are converted to `true` and 0 values are converted to `false`, and then used as logical values.

It is also possible to explicitly convert numerical values to logical values, and vice versa. The `logical` function converts numerical data to logical data, and the `real` function converts logical data to numerical data.

## Relational Operators

**Relational operators** are operators with two numerical or string operands that yield a logical result, depending on the relationship between the two operands. The general form of a relational operator is

$$a_1 \text{ op } a_2$$

where  $a_1$  and  $a_2$  are arithmetic expressions, variables, or strings, and `op` is one of the following relational operators:

**Table 3.1 Relational Operators**

Operator	Operation
<code>==</code>	Equal to
<code>~=</code>	Not equal to
<code>&gt;</code>	Greater than
<code>&gt;=</code>	Greater than or equal to
<code>&lt;</code>	Less than
<code>&lt;=</code>	Less than or equal to

If the relationship between  $a_1$  and  $a_2$  expressed by the operator is true, then the operation returns a true value; otherwise, the operation returns false.

Some relational operations and their results are given below:

Operation	Result
<code>3 &lt; 4</code>	true (1)
<code>3 &lt;= 4</code>	true (1)
<code>3 == 4</code>	false (0)
<code>3 &gt; 4</code>	false (0)
<code>4 &lt;= 4</code>	true (1)
<code>'A' &lt; 'B'</code>	true (1)

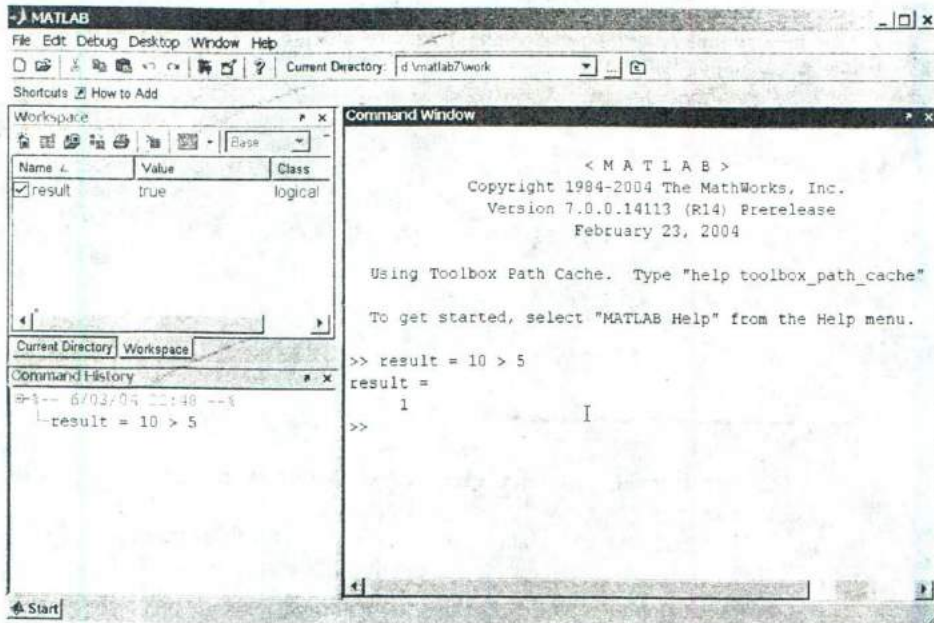
The last relational operation is true because characters are evaluated in alphabetical order.

Note that both `true` and `1` are shown as the result of true operations, and both `false` and `0` are shown as the result of false operations. MATLAB is a bit schizophrenic about how the results of logical operations are displayed. When a relational operator is evaluated in the Command Window, the result of the operation will be displayed as a 0 or 1. When it is displayed in the Workspace Brower, the same value will be show as `false` or `true` (see Figure 3.3).

Relational operators may be used to compare a scalar value with an array. For example, if  $a = \begin{bmatrix} 1 & 0 \\ -2 & 1 \end{bmatrix}$  and  $b = 0$ , then the expression  $a > b$  will yield the logical array  $\begin{bmatrix} \text{true} & \text{false} \\ \text{false} & \text{true} \end{bmatrix}$  (shown as  $\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$  in the Command Window). Relational operators may also be used to compare two arrays, as long as both arrays have the same size. For example, if  $a = \begin{bmatrix} 1 & 0 \\ -2 & 1 \end{bmatrix}$  and  $b = \begin{bmatrix} 0 & 2 \\ -2 & -1 \end{bmatrix}$ , then the expression  $a >= b$  will yield the logical array  $\begin{bmatrix} \text{true} & \text{false} \\ \text{true} & \text{true} \end{bmatrix}$  (shown as  $\begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix}$  in the Command Window). If the arrays have different sizes, a runtime error will result.

Note that since strings are really arrays of characters, *relational operators can only compare two strings if they are of equal lengths*. If they are of unequal lengths, the comparison operation will produce an error. We will learn of a more general way to compare strings in Chapter 6.

The equivalence relational operator is written with two equal signs, while the assignment operator is written with a single equal sign. These are very different operators that beginning programmers often confuse. The `==` symbol is a *comparison* operation that returns a logical result, while the `=` symbol *assigns* the value of the



**Figure 3.3** The result of a relational operator is a true or false value that can be stored in a logical variable. In the example shown here, the result of the operator  $10 > 5$  is displayed as a 1 on the Command Window and as a true in the Workspace Browser.

expression to the right of the equal sign to the variable on the left of the equal sign. It is a very common mistake for beginning programmers to use a single equal sign when trying to do a comparison.

### Programming Pitfalls

Be careful not to confuse the equivalence relational operator (`==`) with the assignment operator (`=`).

In the hierarchy of operations, relational operators are evaluated after all arithmetic operators have been evaluated. Therefore, the following two expressions are equivalent (both are true).

$$7 + 3 < 2 + 11$$

$$(7 + 3) < (2 + 11)$$

## A Caution About the == and ~= Operators

The equivalence operator (==) returns a `true` value (1) when the two values being compared are equal, and a `false` (0) when the two values being compared are different. Similarly, non-equivalence operator (~=) returns a `false` (0) when the two values being compared are equal, and a `true` (1) when the two values being compared are different. These operators are generally safe to use for comparing strings, but they can sometimes produce surprising results when two numeric values are compared. Due to **roundoff errors** during computer calculations, two theoretically equal numbers can differ slightly, causing an equality or inequality test to fail.

For example, consider the following two numbers, both of which should be equal to 0.0.

```
a = 0;
b = sin(pi);
```

Since these numbers are theoretically the same, the relational operation `a == b` *should* produce a 1. In fact, the results of this MATLAB calculation are

```
>> a = 0;
>> b = sin(pi);
>> a == b
ans =
    0
```

MATLAB reports that `a` and `b` are different because a slight roundoff error in the calculation of `sin(pi)` makes the result be  $1.2246 \times 10^{-16}$  instead of exactly zero. The two theoretically equal values differ slightly due to roundoff error!

Instead of comparing two numbers for *exact* equality, you should set up your tests to determine if the two numbers *nearly* equal to each other within some accuracy that takes into account the roundoff error expected for the numbers being compared. The test

```
>> abs(a - b) < 1.0E-14
ans =
    1
```

produces the correct answer, despite the roundoff errors in calculating `a` and `b`.

### \* Good Programming Practice

Be cautious about testing for equality with numeric values, since roundoff errors may cause two variables that should be equal to fail a test for equality. Instead, test to see if the variables are *nearly* equal within the roundoff error to be expected on the computer you are working with.

**Table 3.2 Logic Operators**

Operator	Operation
&	Logical AND
&&	Logical AND with shortcut evaluation
	Logical Inclusive OR
	Logical Inclusive OR with shortcut evaluation
xor	Logical Exclusive OR
~	Logical NOT

## Logic Operators

Logic operators are operators with one or two logical operands that yield a logical result. There are five binary logic operators: AND (& and &&), inclusive OR (| and ||), and exclusive OR (xor), and one unary operator: NOT (~). The general form of a binary logic operation is

$$l_1 \text{ op } l_2$$

and the general form of a unary logic operation is

$$\text{op } l_1$$

where  $l_1$  and  $l_2$  are expressions or variables, and op is one of the logic operators shown in Table 3.2.

If the relationship between  $l_1$  and  $l_2$  expressed by the operator is true, then the operation returns a value of `true` (displayed as 1 in the Command Window); otherwise, the operation returns a value of `false` (0 in the Command Window).

The results of the operators are summarized in **truth tables**, which show the result of each operation for all possible combinations of  $l_1$  and  $l_2$ . Table 3.3 shows the truth tables for all logic operators.

### Logical ANDs

The result of an AND operator is `true` if and only if both input operands are `true`. If either or both operands are `false`, the result is `false`, as shown in Table 3.3.

Note that there are two logical AND operators: && and &. Why are there two AND operators, and what is the difference between them? The basic difference between && and & is that && supports *short-circuit evaluations* (or *partial evaluations*), while & doesn't. That is, && will evaluate expression  $l_1$  and immediately return a `false` value if  $l_1$  is `false`. If  $l_1$  is `false`, the operator never evaluates  $l_2$ , because the result of the operator will be `false` regardless of the value of  $l_2$ . In contrast, the & operator always evaluates both  $l_1$  and  $l_2$  before returning an answer.

A second difference between && and & is that && only works between scalar values, while & works with either scalar or array values, as long as the sizes of the arrays are compatible.



Table 3.3 Truth Tables for Logic Operators

Inputs		and		or		xor	not
$l_1$	$l_2$	$l_1 \& l_2$	$l_1 \&\& l_2$	$l_1   l_2$	$l_1    l_2$	$\text{xor}(l_1, l_2)$	$\sim l_1$
false	false	false	false	false	false	false	true
false	true	false	false	true	true	true	true
true	false	false	false	true	true	true	false
true	true	true	true	true	true	false	false

When should you use `&&` and when should you use `&` in a program? Most of the time, it doesn't matter which AND operation is used. If you are comparing scalars, and it is not necessary to always evaluate  $l_2$ , then use the `&&` operator. The partial evaluation will make the operation faster in the cases where the first operand is false.

Sometimes it is important to use shortcut expressions. For example, suppose that we wanted to test for the situation where the ratio of two variables  $a$  and  $b$  is greater than 10. The code to perform this test is:

```
x = a / b > 10.0
```

This code normally works fine, but what about the case where  $b$  is zero? In that case, we would be dividing by zero, which produces an `Inf` instead of a number. The test could be modified to avoid this problem as follows:

```
x = (b ~= 0) && (a/b > 10.0)
```

This expression uses partial evaluation, so if  $b = 0$ , the expression  $a/b > 10.0$  will never be evaluated, and no `Inf` will occur.

### \* Good Programming Practice

Use the `&` AND operator if it is necessary to ensure that both operands are evaluated in an expression, or if the comparison is between arrays. Otherwise, use the `&&` AND operator, since the partial evaluation will make the operation faster in the cases where the first operand is false. The `&` operator is preferred in most practical cases.

### Logical Inclusive ORs

The result of an inclusive OR operator is `true` if either of the input operands are `true`. If both operands are `false`, the result is `false`, as shown in Table 3.3.

Note that there are two inclusive OR operators: `||` and `|`. Why are there two inclusive OR operators, and what is the difference between them? The basic

difference between `||` and `|` is that `||` supports partial evaluations, while `|` doesn't. That is, `||` will evaluate expression  $l_1$  and immediately return a `true` value if  $l_1$  is `true`. If  $l_1$  is `true`, the operator never evaluates  $l_2$ , because the result of the operator will be `true` regardless of the value of  $l_2$ . In contrast, the `|` operator always evaluates both  $l_1$  and  $l_2$  before returning an answer.

A second difference between `||` and `|` is that `||` only works between scalar values, while `|` works with either scalar or array values, as long as the sizes of the arrays are compatible.

When should you use `||` and when should you use `|` in a program? Most of the time, it doesn't matter which OR operation is used. If you are comparing scalars, and it is not necessary to always evaluate  $l_2$ , use the `||` operator. The partial evaluation will make the operation faster in the cases where the first operand is `true`.

### \* Good Programming Practice

Use the `|` inclusive OR operator if it is necessary to ensure that both operands are evaluated in an expression, or if the comparison is between arrays. Otherwise, use the `||` operator, since the partial evaluation will make the operation faster in the cases where the first operand is `true`. The `|` operator is preferred in most practical cases.

### Logical Exclusive OR

The result of an exclusive OR operator is `true` if and only if one operand is `true` and the other one is `false`. If both operands are `true` or both operands are `false`, then the result is `false`, as shown in Table 3.3. Note that both operands must always be evaluated in order to calculate the result of an exclusive OR.

The logical exclusive OR operation is implemented as a function. For example,

```
a = 10;
b = 0;
x = xor(a, b);
```

This result is `true`. The value of `a` is non-zero, so it will be converted to `true`. The value of `b` is zero, so it will be converted to `false`. Therefore, the result of the `xor` operation will be `true`.

### Logical NOT

The NOT operator is a unary operator, having only one operand. The result of a NOT operator is `true` if its operand is `false`, and `false` if its operand is `true`, as shown in Table 3.3.

### Using Numeric Data with Logic Operators

*Real numeric data can also be use with logic operators.* Since logic operators expect logical input values, MATLAB converts non-zero values to `true` and zero

values to false before performing the operation. Thus, the result of `~5` is false (0 in the Command Window) and the result of `~0` is true (1 in the Command Window).

Logic operators may be used to compare a scalar value with an array. For example, if  $a = \begin{bmatrix} \text{true} & \text{false} \\ \text{false} & \text{true} \end{bmatrix}$  and  $b = \text{false}$ , then the expression  $a \& b$  will yield the result  $\begin{bmatrix} \text{false} & \text{false} \\ \text{false} & \text{false} \end{bmatrix}$  (displayed as  $\begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix}$  in the Command Window). Logic operators may also be used to compare two arrays, as long as both arrays have the same size. For example, if  $a = \begin{bmatrix} \text{true} & \text{false} \\ \text{false} & \text{true} \end{bmatrix}$  and  $b = \begin{bmatrix} \text{true} & \text{true} \\ \text{false} & \text{false} \end{bmatrix}$ , then the expression  $a|b$  will yield the result  $\begin{bmatrix} \text{true} & \text{true} \\ \text{false} & \text{true} \end{bmatrix}$  (displayed as  $\begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix}$  in the Command Window). If the arrays have different sizes, a runtime error will result.

Logic operators may *not* be used with complex or imaginary numeric data. For example, an expression such as `"2i & 2i"` will produce an error when it is evaluated.

### Hierarchy of Operations

In the hierarchy of operations, logic operators are evaluated *after all arithmetic operations and all relational operators have been evaluated*. The order in which the operators in an expression are evaluated is:

1. All arithmetic operators are evaluated first in the order previously described.
2. All relational operators (`==`, `~=`, `>`, `>=`, `<`, `<=`) are evaluated, working from left to right.
3. All `~` operators are evaluated.
4. All `&` and `&&` operators are evaluated, working from left to right.
5. All `|`, `||`, and `xor` operators are evaluated, working from left to right.

As with arithmetic operations, parentheses can be used to change the default order of evaluation. Examples of some logic operators and their results are given below.

#### Example 3.1

Assume that the following variables are initialized with the values shown, and calculate the result of the specified expressions:

```
value1 = true
value2 = false
value3 = 1
```

```
value4 = -10
value5 = 0
value6 = [1 2; 0 1]
```

Expression	Result	Comment				
(a) ~value1	false					
(b) ~value3	false	The number 1 is converted to true before operation is applied				
(c) value1   value2	true					
(d) value1 & value2	false					
(e) value4 & value5	false	-10 is converted to true and 0 is converted to false before the operation is applied				
(f) ~(value4 & value5)	true	-10 is converted to true and 0 is converted to false before the operation is applied				
(g) value1 + value4	-9	value1 is converted to the number 1 before the addition is performed				
(h) value1 + (~value4)	1	The logical value1 is converted to the number 1 before the addition is performed. The number value4 is converted to true before the NOT is performed. Then ~value4 is evaluated to be false. This false value is converted to 0 before the addition, so the final result is 1 + 0 = 1.				
(i) value3 && value6	Illegal	The && operator must be used with scalar operands.				
(j) value3 & value6	<table border="1" style="display: inline-table; vertical-align: middle;"> <tr><td>true</td><td>true</td></tr> <tr><td>false</td><td>true</td></tr> </table>	true	true	false	true	AND between a scalar and an array operand.
true	true					
false	true					

The ~ operator is evaluated before other logic operators. Therefore, the parentheses in part (f) of the above example were required. If they had been absent, the expression in part (f) would have been evaluated in the order (~value3) & value5.

## Logical Functions

MATLAB includes a number of logical functions that return true whenever the condition they test for is true, and false whenever the condition they test for is false. These functions can be used with relational and logic operator to control the operation of branches and loops.

A few of the more important logical functions are given in Table 3.4.

**Table 3.4 Selected MATLAB Logical Functions**

Function	Purpose
<code>ischar(a)</code>	Returns true if <code>a</code> is a character array and false otherwise.
<code>isempty(a)</code>	Returns true if <code>a</code> is an empty array and false otherwise.
<code>isinf(a)</code>	Returns true if the value of <code>a</code> is infinite (Inf) and false otherwise.
<code>isnan(a)</code>	Returns true if the value of <code>a</code> is NaN (not a number) and false otherwise.
<code>isnumeric(a)</code>	Returns true if <code>a</code> is a numeric array and false otherwise.
<code>logical(a)</code>	Converts numerical values to logical values: if a value is non-zero, it is converted to true. If it is zero, it is converted to false.

**Quiz 3.1**

This quiz provides a quick check to see if you have understood the concepts introduced in Section 3.3. If you have trouble with the quiz, reread the sections, ask your instructor, or discuss the material with a fellow student. The answers to this quiz are found in the back of the book.

Assume that `a`, `b`, `c`, and `d` are as defined, and evaluate the following expressions.

```
a = 20;      b = -2;
c = 0;      d = 1;
```

- `a > b`
- `b > d`
- `a > b && c > d`
- `a == b`
- `a && b > c`
- `~b`

Assume that `a`, `b`, `c`, and `d` are as defined, and evaluate the following expressions.

```
a = 2;      b = [ 1 -2;
                  0 10];
c = [ 0 1;
      2 0];  d = [ -2 1 2;
                  0 1 0];
```

- `~(a > b)`
- `a > c && b > c`
- `c <= d`

10. `logical(d)`
11. `a * b > c`
12. `a * (b > c)`

Assume that `a`, `b`, `c`, and `d` are as defined. Explain the order in which each of the following expressions are evaluated, and specify the results in each case:

```
a = 2;          b = 3;
c = 10;        d = 0;
```

13. `a*b^2 > a*c`
14. `d || b > a`
15. `(d | b) > a`

Assume that `a`, `b`, `c`, and `d` are as defined, and evaluate the following expressions.

```
a = 20;        b = -2;
c = 0;        d = 'Test';
```

16. `isinf(a/b)`
17. `isinf(a/c)`
18. `a > b && ischar(d)`
19. `isempty(c)`
20. `(~a) & b`
21. `(~a) + b`

## 3.4 Branches

**Branches** are MATLAB statements that permit us to select and execute specific sections of code (called *blocks*) while skipping other sections of code. They are variations of the `if` construct, the `switch` construct, and the `try/catch` construct.

### The `if` Construct

The `if` construct has the form

```
if control_expr_1
    Statement 1
    Statement 2
    ...
elseif control_expr_2
    Statement 1
    Statement 2
    ...
```

} Block 1

} Block 2

```

else
    Statement 1
    Statement 2
    ...
end
} Block 3

```

where the control expressions are logical expressions that control the operation of the `if` construct. If `control_expr_1` is true (non-zero), then the program executes the statements in Block 1, and skips to the first executable statement following the `end`. Otherwise, the program checks for the status of `control_expr_2`. If `control_expr_2` is true (non-zero), then the program executes the statements in Block 2, and skips to the first executable statement following the `end`. If all control expressions are zero, then the program executes the statements in the block associated with the `else` clause.

There can be any number of `elseif` clauses (0 or more) in an `if` construct, but there can be at most one `else` clause. The control expression in each clause will be tested only if the control expressions in every clause above it are false (0). Once one of the expressions proves to be true and the corresponding code block is executed, the program skips to the first executable statement following the `end`. If all control expressions are false, then the program executes the statements in the block associated with the `else` clause. If there is no `else` clause, then execution continues after the `end` statement without executing any part of the `if` construct.

Note that the MATLAB keyword `end` in this construct is *completely different* from the MATLAB function `end` that we used in Chapter 2 to return the highest value of a given subscript. MATLAB tells the difference between these two uses of `end` from the context in which the word appears within an M-file.

In most circumstances, *the control expressions will be some combination of relational and logic operators*. As we learned earlier in this chapter, relational and logic operators produce a true (1) when the corresponding condition is true and a false (0) when the corresponding condition is false. When an operator is true, its result is non-zero, and the corresponding block of code will be executed.

As an example of an `if` construct, consider the solution of a quadratic equation of the form

$$ax^2 + bx + c = 0 \quad (3-1)$$

The solution to this equation is

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a} \quad (3-2)$$

The term  $b^2 - 4ac$  is known as the *discriminant* of the equation. If  $b^2 - 4ac > 0$ , then there are two distinct real roots to the quadratic equation. If  $b^2 - 4ac = 0$ , then there is a single repeated root to the equation, and if  $b^2 - 4ac < 0$ , then there are two complex roots to the quadratic equation.

Suppose that we wanted to examine the discriminant of a quadratic equation and to tell a user whether the equation has two complex roots, two identical real

roots, or two distinct real roots. In pseudocode, this construct would take the form

```

if (b^2 - 4*a*c) < 0
    Write msg that equation has two complex roots.
elseif (b^2 - 4*a*c) == 0
    Write msg that equation has two identical real roots.
else
    Write msg that equation has two distinct real roots.
end

```

The MATLAB statements to do this are

```

if (b^2 - 4*a*c) < 0
    disp('This equation has two complex roots. ');
elseif (b^2 - 4*a*c) == 0
    disp('This equation has two identical real roots. ');
else
    disp('This equation has two distinct real roots. ');
end

```

For readability, the blocks of code within an `if` construct are usually indented by 2 or 3 spaces, but this is not actually required.

### \* Good Programming Practice

Always indent the body of an `if` construct by 2 or more spaces to improve the readability of the code.

It is possible to write a complete `if` construct on a single line by separating the parts of the construct by commas or semicolons. Thus the following two constructs are identical:

```

if x < 0
    y = abs(x);
end

and

if x < 0; y = abs(x); end

```

However, this should only be done for very simple constructs.

### Examples Using `if` Constructs

We will now look at two examples that illustrate the use of `if` constructs.



### Example 3.2—The Quadratic Equation

Write a program to solve for the roots of a quadratic equation, regardless of type.

**SOLUTION** We will follow the design steps outlined earlier in the chapter.

**1. State the problem.**

The problem statement for this example is very simple. We want to write a program that will solve for the roots of a quadratic equation, whether they are distinct real roots, repeated real roots, or complex roots.

**2. Define the inputs and outputs.**

The inputs required by this program are the coefficients  $a$ ,  $b$ , and  $c$  of the quadratic equation

$$ax^2 + bx + c = 0 \quad (3-1)$$

The output from the program will be the roots of the quadratic equation, whether they are distinct real roots, repeated real roots, or complex roots.

**3. Design the algorithm.**

This task can be broken down into three major sections, whose functions are input, processing, and output:

```

Read the input data
Calculate the roots
Write out the roots

```

We will now break each of the above major sections into smaller, more detailed pieces. There are three possible ways to calculate the roots, depending on the value of the discriminant, so it is logical to implement this algorithm with a three-branched if construct. The resulting pseudocode is:

```

Prompt the user for the coefficients a, b, and c.
Read a, b, and c
discriminant <- b^2 - 4 * a * c
if discriminant > 0
    x1 <- ( -b + sqrt(discriminant) ) / ( 2 * a )
    x2 <- ( -b - sqrt(discriminant) ) / ( 2 * a )
    Write msg that equation has two distinct real roots.
    Write out the two roots.
elseif discriminant == 0
    x1 <- -b / ( 2 * a )
    Write msg that equation has two identical real roots.
    Write out the repeated root.
else
    real_part <- -b / ( 2 * a )

```

```

    imag_part <- sqrt ( abs ( discriminant ) ) / ( 2 * a )
    Write msg that equation has two complex roots.
    Write out the two roots.
end

```

#### 4. Turn the algorithm into MATLAB statements.

The final MATLAB code is shown in below:

```

% Script file: calc_roots.m
%
% Purpose:
%   This program solves for the roots of a quadratic equation
%   of the form  $a*x^2 + b*x + c = 0$ . It calculates the answers
%   regardless of the type of roots that the equation possesses.
%
% Record of revisions:
%
%   Date           Programmer           Description of change
%   ----           -
%   01/02/04      S. J. Chapman           Original code
%
% Define variables:
%   a              -- Coefficient of  $x^2$  term of equation
%   b              -- Coefficient of  $x$  term of equation
%   c              -- Constant term of equation
%   discriminant   -- Discriminant of the equation
%   imag_part      -- Imag part of equation (for complex roots)
%   real_part      -- Real part of equation (for complex roots)
%   x1             -- First solution of equation (for real roots)
%   x2             -- Second solution of equation (for real roots)
%
% Prompt the user for the coefficients of the equation
disp ('This program solves for the roots of a quadratic ');
disp ('equation of the form  $A*X^2 + B*X + C = 0$ . ');
a = input ('Enter the coefficient A: ');
b = input ('Enter the coefficient B: ');
c = input ('Enter the coefficient C: ');
% Calculate discriminant
discriminant = b^2 - 4 * a * c;

% Solve for the roots, depending on the value of the discriminant
if discriminant > 0 % there are two real roots, so...
    x1 = ( -b + sqrt(discriminant) ) / ( 2 * a );
    x2 = ( -b - sqrt(discriminant) ) / ( 2 * a );
    disp ('This equation has two real roots:');
    fprintf ('x1 = %f\n', x1);
    fprintf ('x2 = %f\n', x2);

```

```

elseif discriminant == 0 % there is one repeated root, so...
    x1 = ( -b ) / ( 2 * a );
    disp ('This equation has two identical real roots:');
    fprintf ('x1 = x2 = %f\n', x1);
else % there are complex roots, so ...
    real_part = ( -b ) / ( 2 * a );
    imag_part = sqrt ( abs ( discriminant ) ) / ( 2 * a );
    disp ('This equation has complex roots:');
    fprintf('x1 = %f +i %f\n', real_part, imag_part );
    fprintf('x1 = %f -i %f\n', real_part, imag_part );
end

```

### 5. Test the program.

Next, we must test the program using real input data. Since there are three possible paths through the program, we must test all three paths before we can be certain that the program is working properly. From Equation (3-2), it is possible to verify the solutions to the equations given below:

$$\begin{array}{ll}
 x^2 + 5x + 6 = 0 & x = -2, \text{ and } x = -3 \\
 x^2 + 4x + 4 = 0 & x = -2 \\
 x^2 + 2x + 5 = 0 & x = -1 \pm i2
 \end{array}$$

If this program is executed three times with the above coefficients, the results are as shown below (user inputs are shown in bold face):

#### » calc\_roots

This program solves for the roots of a quadratic equation of the form  $A \cdot X^2 + B \cdot X + C = 0$ .

Enter the coefficient A: **1**

Enter the coefficient B: **5**

Enter the coefficient C: **6**

This equation has two real roots:

x1 = -2.000000

x2 = -3.000000

#### » calc\_roots

This program solves for the roots of a quadratic equation of the form  $A \cdot X^2 + B \cdot X + C = 0$ .

Enter the coefficient A: **1**

Enter the coefficient B: **4**

Enter the coefficient C: **4**

This equation has two identical real roots:

x1 = x2 = -2.000000

#### » calc\_roots

This program solves for the roots of a quadratic

```

equation of the form A*X^2 + B*X + C = 0.
Enter the coefficient A: 1
Enter the coefficient B: 2
Enter the coefficient C: 5
This equation has complex roots:
x1 = -1.000000 +i 2.000000i
x2 = -1.000000 -i 2.000000i

```

The program gives the correct answers for our test data in all three possible cases.

### Example 3.3—Evaluating a Function of Two Variables

Write a MATLAB program to evaluate a function  $f(x, y)$  for any two user-specified values  $x$  and  $y$ . The function  $f(x, y)$  is defined as follows.

$$f(x, y) = \begin{cases} x + y & x \geq 0 \text{ and } y \geq 0 \\ x + y^2 & x \geq 0 \text{ and } y < 0 \\ x^2 + y & x < 0 \text{ and } y \geq 0 \\ x^2 + y^2 & x < 0 \text{ and } y < 0 \end{cases}$$

**SOLUTION** The function  $f(x, y)$  is evaluated differently depending on the signs of the two independent variables  $x$  and  $y$ . To determine the proper equation to apply, it will be necessary to check for the signs of the  $x$  and  $y$  values supplied by the user.

**1. State the problem.**

This problem statement is very simple: Evaluate the function  $f(x, y)$  for any user-supplied values of  $x$  and  $y$ .

**2. Define the inputs and outputs.**

The inputs required by this program are the values of the independent variables  $x$  and  $y$ . The output from the program will be the value of the function  $f(x, y)$ .

**3. Design the algorithm.**

This task can be broken down into three major sections, whose functions are input, processing, and output:

```

Read the input values x and y
Calculate f(x,y)
Write out f(x,y)

```

We will now break each of the above major sections into smaller, more detailed pieces. There are four possible ways to calculate the function  $f(x, y)$ , depending upon the values of  $x$  and  $y$ , so it is logical to implement this algorithm with a four-branched `if` construct. The resulting pseudocode is:



```

elseif x >= 0 && y < 0
    fun = x + y^2;
elseif x = 0 && y >= 0
    fun = x^2 + y;
else
    fun = x^2 + y^2;
end

% Write the value of the function.
disp(['The value of the function is ' num2str(fun)]);

```

### 5. Test the program.

Next, we must test the program using real input data. Since there are four possible paths through the program, we must test all four paths before we can be certain that the program is working properly. To test all four possible paths, we will execute the program with the four sets of input values  $(x, y) = (2, 3), (2, -3), (-2, 3),$  and  $(-2, -3)$ . Calculating by hand, we see that

$$\begin{aligned}
 f(2, 3) &= 2 + 3 = 5 \\
 f(2, -3) &= 2 + (-3)^2 = 11 \\
 f(-2, 3) &= (-2)^2 + 3 = 7 \\
 f(-2, -3) &= (-2)^2 + (-3)^2 = 13
 \end{aligned}$$

If this program is compiled, and then run four times with the above values, the results are:

```

» funxy
Enter the x coefficient: 2
Enter the y coefficient: 3
The value of the function is 5
» funxy
Enter the x coefficient: 2
Enter the y coefficient: -3
The value of the function is 11
» funxy
Enter the x coefficient: -2
Enter the y coefficient: 3
The value of the function is 7
» funxy
Enter the x coefficient: -2
Enter the y coefficient: -3
The value of the function is 13

```

The program gives the correct answers for our test values in all four possible cases.

## Notes Concerning the Use of `if` Constructs

The `if` construct is very flexible. It must have one `if` statement and one `end` statement. In between, it can have any number of `elseif` clauses, and may also have one `else` clause. With this combination of features, it is possible to implement any desired branching construct.

In addition, `if` constructs may be **nested**. Two `if` constructs are said to be nested if one of them lies entirely within a single code block of the other one. The following two `if` constructs are properly nested.

```

if x > 0
    ...
    if y < 0
        ...
    end
    ...
end

```

The MATLAB interpreter always associates a given `end` statement with the most recent `if` statement, so the first `end` above closes the `if y < 0` statement, while the second `end` closes the `if x > 0` statement. This works well for a properly written program, but can cause the interpreter to produce confusing error messages in cases where the programmer makes a coding error. For example, suppose that we have a large program containing a construct like the one shown below.

```

...
if (test1)
    ...
    if (test2)
        ...
        if (test3)
            ...
        end
        ...
    end
    ...
end
...
end

```

This program contains three nested `if` constructs that may span hundreds of lines of code. Now suppose that the first `end` statement is accidentally deleted during an editing session. When that happens, the MATLAB interpreter will automatically associate the second `end` with the innermost `if (test3)` construct, and the third `end` with the middle `if (test2)`. When the interpreter reaches the end of the file, it will notice that the first `if (test1)` construct was never ended, and it will generate an error message saying that there is a missing `end`. Unfortunately, it can't tell *where* the problem occurred, so we will have to go back and manually search the entire program to locate the problem.

It is sometimes possible to implement an algorithm using either multiple `elseif` clauses or nested `if` statements. In that case, a programmer may choose whichever style he or she prefers.

### Example 3.4—Assigning Letter Grades

Suppose that we are writing a program that reads in a numerical grade and assigns a letter grade to it according to the following table:

$95 < \text{grade}$	A
$86 < \text{grade} \leq 95$	B
$76 < \text{grade} \leq 86$	C
$66 < \text{grade} \leq 76$	D
$0 < \text{grade} \leq 66$	F

Write an `if` construct that will assign the grades as described above using (a) multiple `elseif` clauses and (b) nested `if` constructs.

SOLUTION

(a) One possible structure using `elseif` clauses is

```
if grade > 95.0
    disp('The grade is A.');
```

```
elseif grade > 86.0
    disp('The grade is B.');
```

```
elseif grade > 76.0
    disp('The grade is C.');
```

```
elseif grade > 66.0
    disp('The grade is D.');
```

```
else
    disp('The grade is F.');
```

```
end
```

(b) One possible structure using nested `if` constructs is

```
if grade > 95.0
    disp('The grade is A.');
```

```
else
    if grade > 86.0
        disp('The grade is B.');
```

```
    else
        if grade > 76.0
            disp('The grade is C.');
```

```
        else
            if grade > 66.0
                disp('The grade is D.');
```

```
            else
                disp('The grade is F.');
```



```

        end
    end
end
end

```

It should be clear from the above example that if there are a lot of mutually exclusive options, a single `if` construct with multiple `elseif` clauses will be simpler than a nested `if` construct.

### Good Programming Practice

For branches in which there are many mutually exclusive options, use a single `if` construct with multiple `elseif` clauses in preference to nested `if` constructs.

## The `switch` Construct

The `switch` construct is another form of branching construct. It permits a programmer to select a particular code block to execute based on the value of a single integer, character, or logical expression. The general form of a `switch` construct is:

```

switch (switch_expr)
case case_expr_1,
    Statement 1
    Statement 2
    ...
case case_expr_2,
    Statement 1
    Statement 2
    ...
...
otherwise,
    Statement 1
    Statement 2
    ...
end

```

} Block 1

} Block 2

} Block n

If the value of `switch_expr` is equal to `case_expr_1`, then the first code block will be executed, and the program will jump to the first statement following the end of the `switch` construct. Similarly, if the value of `switch_expr` is equal to `case_expr_2`, then the second code block will be executed, and the program will jump to the first statement following the end of the `switch` construct. The same idea applies for any other cases in the construct. The `otherwise` code block

is optional. If it is present, it will be executed whenever the value of `switch_expr` is outside the range of all of the case selectors. If it is not present and the value of `switch_expr` is outside the range of all of the case selectors, then none of the code blocks will be executed. The pseudocode for the case construct looks just like its MATLAB implementation.

If many values of the `switch_expr` should cause the same code to execute, all of those values may be included in a single block by enclosing them in brackets, as shown below. If the switch expression matches any of the case expressions in the list, then the block will be executed.

```
switch (switch_expr)
case {case_expr_1, case_expr_2, case_expr_3},
    Statement 1
    Statement 2
    ...
otherwise,
    Statement 1
    Statement 2
    ...
end
```

} Block 1

} Block n

The `switch_expr` and each `case_expr` may be either numerical or string values.

Note that at most one code block can be executed. After a code block is executed, execution skips to the first executable statement after the `end` statement. Thus if the switch expression matches more than one case expression, *only the first one of them will be executed*.

Let's look at a simple example of a switch construct. The following statements determine whether an integer between 1 and 10 is even or odd, and print out an appropriate message. It illustrates the use of a list of values as case selectors, and also the use of the `otherwise` block.

```
switch (value)
case {1,3,5,7,9},
    disp('The value is odd.');
```

```
case {2,4,6,8,10},
    disp('The value is even.');
```

```
otherwise,
    disp('The value is out of range.');
```

```
end
```

## The try/catch Construct

The `try/catch` construct is a special form branching construct designed to trap errors. Ordinarily, when a MATLAB program encounters an error while running, the program aborts. The `try/catch` construct modifies this default behavior. If an error occurs in a statement in the `try` block of this construct, then instead of

aborting, the code in the `catch` block is executed and the program keeps running. This allows a programmer to handle errors within the program without causing the program to stop.

The general form of a `try/catch` construct is:

```

try
    Statement 1
    Statement 2
    ...
catch
    Statement 1
    Statement 2
    ...
end

```

} Try Block

} Catch Block

When a `try/catch` construct is reached, the statements in the `try` block of a will be executed. If no error occurs, the statements in the `catch` block will be skipped, and execution will continue at the first statement following the end of the construct. On the other hand, if an error *does* occur in the `try` block, the program will stop executing the statements in the `try` block, and immediately execute the statements in the `catch` block.

An example program containing a `try/catch` construct follows. This program creates an array, and asks the user to specify an element of the array to display. The user will supply a subscript number, and the program displays the corresponding array element. The statements in the `try` block will always be executed in this program, while the statements in the `catch` block will only be executed of an error occurs in the `try` block.

```

% Initialize array
a = [ 1 -3 2 5];
try
    % Try to display an element
    index = input('Enter subscript of element to display: ');
    disp( ['a(' int2str(index) ') = ' num2str(a(index))] );
catch
    % If we get here an error occurred
    disp( ['Illegal subscript: ' int2str(index)] );
end

```

When this program is executed, the results are: z

```

> try_catch
Enter subscript of element to display: 3
a(3) = 2
> try_catch
Enter subscript of element to display: 8
Illegal subscript: 8

```

**Quiz 3.2**

This quiz provides a quick check to see if you have understood the concepts introduced in Section 3.4. If you have trouble with the quiz, reread the section, ask your instructor, or discuss the material with a fellow student. The answers to this quiz are found in the back of the book.

Write MATLAB statements that perform the functions described below.

1. If  $x$  is greater than or equal to zero, then assign the square root of  $x$  to variable `sqrt_x` and print out the result. Otherwise, print out an error message about the argument of the square root function, and set `sqrt_x` to zero.
2. A variable `fun` is calculated as `numerator/denominator`. If the absolute value of `denominator` is less than  $1.0E-300$ , write "Divide by 0 error." Otherwise, calculate and print out `fun`.
3. The cost per mile for a rented vehicle is \$1.00 for the first 100 miles, \$0.80 for the next 200 miles, and \$0.70 for all miles in excess of 300 miles. Write MATLAB statements that determine the total cost and the average cost per mile for a given number of miles (stored in variable `distance`).

Examine the following MATLAB statements. Are they correct or incorrect? If they are correct, what do they output? If they are incorrect, what is wrong with them?

4. 

```
if volts > 125
    disp('WARNING: High voltage on line.');
```

```
if volts < 105
    disp('WARNING: Low voltage on line.');
```

```
else
    disp('Line voltage is within tolerances.');
```

```
end
```
5. 

```
color = 'yellow';
switch ( color )
case 'red',
    disp('Stop now!');
```

```
case 'yellow',
    disp('Prepare to stop.');
```

```
case 'green',
    disp('Proceed through intersection.');
```

```
otherwise,
    disp('Illegal color encountered.');
```

```
end
```

```

6. if temperature > 37
    disp('Human body temperature exceeded.');
```

---

```

elseif temperature > 100
    disp('Boiling point of water exceeded.');
```

---

```

end

```

## 3.5 Additional Plotting Features

This section describes additional features of the simple two-dimensional plots introduced in Chapter 2. These features permit us to control the range of  $x$  and  $y$  values displayed on a plot, to lay multiple plots on top of each other, to create multiple figures, to create multiple subplots within a figure, and to provide greater control of the plotted lines and text strings. In addition, we will learn how to create polar plots.

### Controlling $x$ - and $y$ -axis Plotting Limits

By default, a plot is displayed with  $x$ - and  $y$ -axis ranges wide enough to show every point in an input data set. However, it is sometimes useful to display only the subset of the data that is of particular interest. This can be done using the **axis** command/function (see the Sidebar on the next page about the relationship between MATLAB commands and functions).

Some of the forms of the **axis** command/function are shown in Table 3.5 below. The two most important forms are shown in bold type—they let a

**Table 3.5** Forms of the **axis** Function/Command

Command	Description
<b>v = axis;</b>	This function returns a 4-element row vector containing [xmin xmax ymin ymax], where xmin, xmax, ymin, and ymax are the current limits of the plot.
<b>axis ([xmin xmax ymin ymax]);</b>	This function sets the $x$ and $y$ limits of the plot to the specified values.
axis equal	This command sets the axis increments to be equal on both axes.
axis square	This command makes the current axis box square.
axis normal	This command cancels the effect of <b>axis equal</b> and <b>axis square</b> .
axis off	This command turns off all axis labeling, tick marks, and background.
axis on	This command turns on all axis labeling, tick marks, and background (default case).

## Command/Function Duality

Some MATLAB commands seem to be unable to make up their minds whether they are commands or functions. For example, sometimes `axis` seems to be a command and sometimes it seems to be a function. Sometimes we treat it as a command: `axis on`, and other times we might treat it as a function: `axis([0 20 0 35])`. How is this possible?

The short answer is that MATLAB commands are really implemented by functions, and the MATLAB interpreter is smart enough to substitute the function call whenever it encounters the command. It is always possible to call the command directly as a function instead of using the command syntax. Thus the following two statements are identical:

```
axis on;
axis ('on');
```

Whenever MATLAB encounters a command, it forms a function from the command by treating each command argument as a character string and calling the equivalent function with those character strings as arguments. Thus MATLAB interprets the command

```
garbage 1 2 3
```

as the following function call:

```
garbage('1','2','3')
```

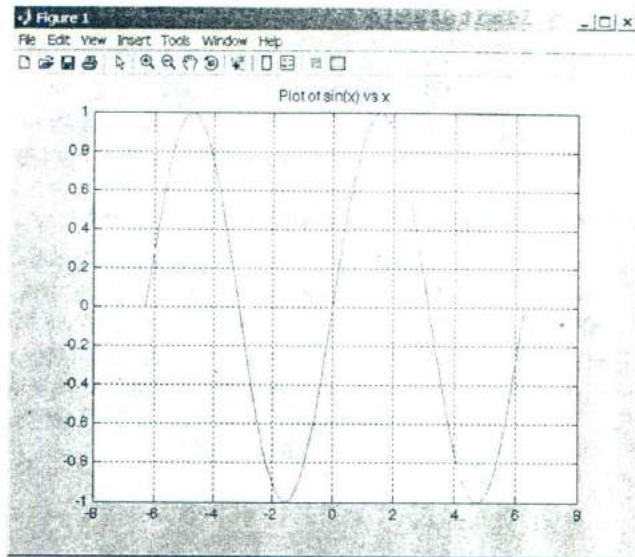
Note that *only functions with character arguments can be treated as commands*. Functions with numerical arguments must be used in function form only. This fact explains why `axis` is sometimes treated as a command and sometimes treated as a function.

programmer get the current limits of a plot and modify them. A complete list of all options can be found in the MATLAB on-line documentation.

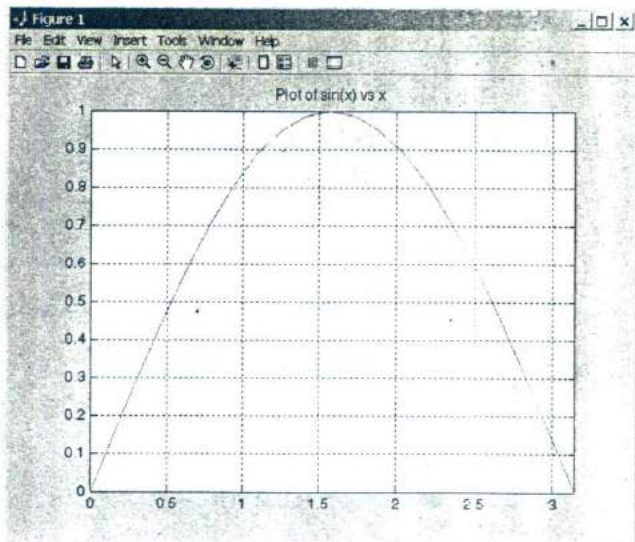
To illustrate the use of `axis`, we will plot the function  $f(x) = \sin x$  from  $-2\pi$  to  $2\pi$ , and then restrict the axes to the region to  $0 \leq x \leq \pi$  and  $0 \leq y \leq 1$ . The statements to create this plot are shown below, and the resulting plot is shown in Figure 3.4a.

```
x = -2*pi:pi/20:2*pi;
y = sin(x);
plot(x,y);
title ('Plot of sin(x) vs x');
grid on;
```

The current limits of this plot can be determined from the basic `axis` function.



(a)



(b)

Figure 3.4 (a) Plot of  $\sin x$  versus  $x$ . (b) Closeup of the region  $[0, \pi]$ .

```

>> limits=axis
limits =
    -8  8  -1  1

```

These limits can be modified with the function call `axis([0 pi 0 1])`. After that function is executed, the resulting plot is shown in Figure 3.4b.

### Plotting Multiple Plots on the Same Axes

Normally, a new plot is created each time that a `plot` command is issued, and the previous data is lost. This behavior can be modified with the **hold** command. After a `hold on` command is issued, all additional plots will be laid on top of the previously existing plots. A `hold off` command switches plotting behavior back to the default situation, in which a new plot replaces the previous one.

For example, the following commands plot  $\sin x$  and  $\cos x$  on the same axes. The resulting plot is shown in Figure 3.5.

```

x = -pi:pi/20:pi;
y1 = sin(x);
y2 = cos(x);
plot(x,y1,'b-');
hold on;
plot(x,y2,'k--');
hold off;
legend('sin x','cos x');

```

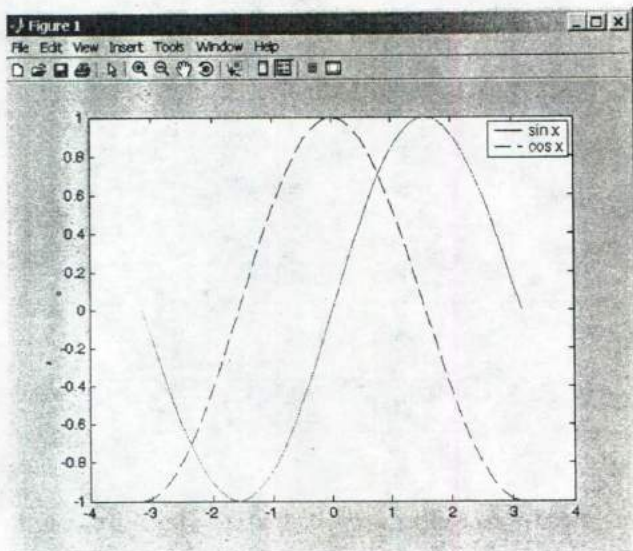


Figure 3.5 Multiple curves plotted on a single set of axes using the `hold` command.



## Creating Multiple Figures

MATLAB can create multiple Figure Windows, with different data displayed in each window. Each Figure Window is identified by a *figure number*, which is a small positive integer. The first Figure Window is Figure 1, the second is Figure 2, etc. One of the Figure Windows will be the **current figure**, and all new plotting commands will be displayed in that window.

The current figure is selected with the **figure** function. This function takes the form “`figure(n)`”, where  $n$  is a figure number. When this command is executed, Figure  $n$  becomes the current figure and is used for all plotting commands. The figure is automatically created if it does not already exist. The current figure may also be selected by clicking on it with the mouse.

The function `gcf` returns the number of the current figure. This function can be used by an M-file if it needs to know the current figure.

The following commands illustrate the use of the figure function. They create two figures, displaying  $e^x$  in the first figure and  $e^{-x}$  in the second one.

```
figure(1)
x = 0:0.05:2;
y1 = exp(x);
plot(x,y1);
figure(2)
y2 = exp(-x);
plot(x,y2);
```

## Subplots

It is possible to place more than one set of axes on a single figure, creating multiple **subplots**. Subplots are created with a `subplot` command of the form

```
subplot(m,n,p)
```

This command divides the current figure into  $m \times n$  equal-sized regions, arranged in  $m$  rows and  $n$  columns, and creates a set of axes at position  $p$  to receive all current plotting commands. The subplots are numbered from left to right and from top to bottom. For example, the command `subplot(2,3,4)` would divide the current figure into six regions arranged in two rows and three columns, and create an axis in position 4 (the lower left one) to accept new plot data (see Figure 3.6).

If a `subplot` command creates a new set of axes that conflict with a previously existing set, then the older axes are automatically deleted.

The commands below create two subplots within a single window, and display the separate graphs in each subplot. The resulting figure is shown in Figure 3.7.

```
figure(1)
subplot(2,1,1)
x = -pi:pi/20:pi;
y = sin(x);
```

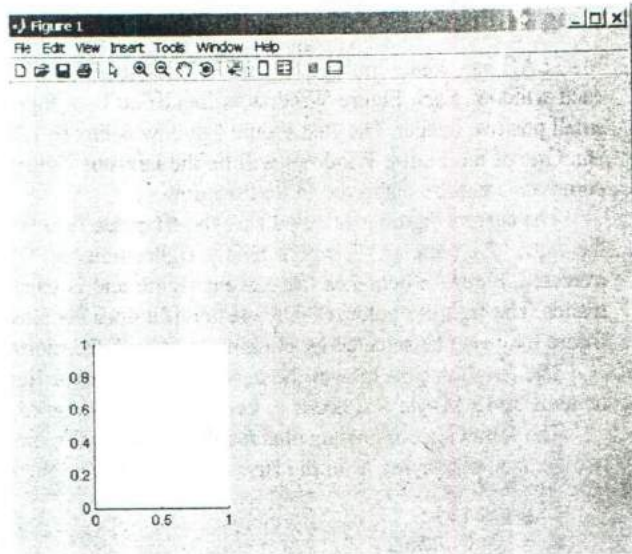


Figure 3.6 The axis created by the subplot (2, 3, 4) command.

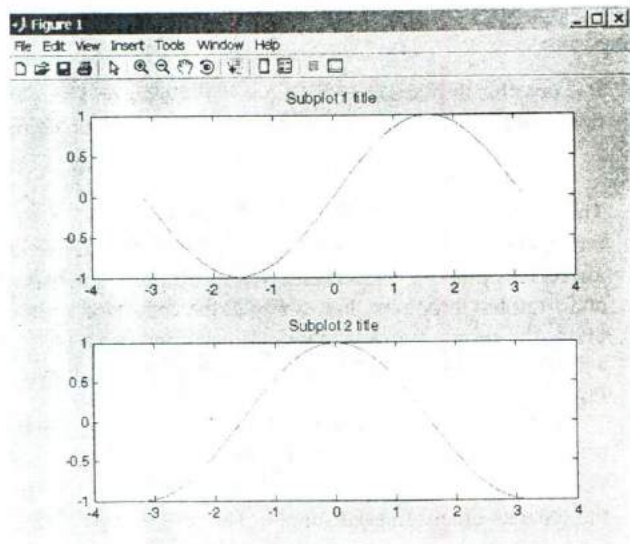


Figure 3.7 A figure containing two subplots.

```

plot(x,y);
title('Subplot 1 title');
subplot(2,1,2)
x = -pi:pi/20:pi;
y = cos(x);
plot(x,y);
title('Subplot 2 title');

```

## Enhanced Control of Plotted Lines

In Chapter 1 we learned how to set the color, style, and marker type for a line. It is also possible to set four additional properties associated with each line:

- **Width**—specifies the width of each line in points
- **MarkerEdgeColor**—specifies the color of the marker or the edge color for filled markers
- **MarkerFaceColor**—specifies the color of the face of filled markers.
- **MarkerSize**—specifies the size of the marker in points.

These properties are specified in the `plot` command after the data to be plotted in the following fashion:

```
plot(x,y, 'PropertyName', value, ...)
```

For example, the following command plots a 3-point wide solid black line with 6-point wide circular markers at the data points. Each marker has a red edge and a green center, as shown in Figure 3.8.

```

x = 0:pi/15:4*pi;
y = exp(2*sin(x));
plot(x,y, '-ko', 'LineWidth', 3.0, 'MarkerSize', 6, ...
      'MarkerEdgeColor', 'r', 'MarkerFaceColor', 'g')

```

## Enhanced Control of Text Strings

It is possible to enhance plotted text strings (titles, axis labels, etc.) with formatting such as bold face, italics, etc., and with special characters such as Greek and mathematical symbols.

The font used to display the text can be modified by **stream modifiers**. A stream modifier is a special sequence of characters that tells the MATLAB interpreter to change its behavior. The most common stream modifiers are:

- `\bf`—Bold face
- `\it`—Italics
- `\rm`—Restore normal font
- `\fontname{fontname}`—Specify the font name to use
- `\fontsize{fontsize}`—Specify font size
- `_ {xxx}`—The characters inside the braces are subscripts
- `^ {xxx}`—The characters inside the braces are superscripts

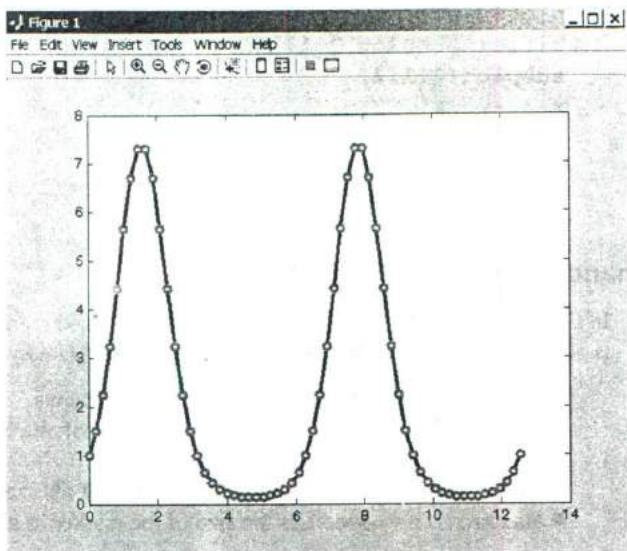


Figure 3.8 A plot illustrating the use of the `LineWidth` and `Marker` properties.

Once a stream modifier has been inserted into a text string, it will remain in effect until the end of the string or until cancelled. Any stream modifier can be followed by braces `{}`. If a modifier is followed by braces, only the text within the braces is effected.

Special Greek and mathematical symbols may also be used in text strings. They are created by embedding *escape sequences* into the text string. These escape sequences are the same as those defined in the TeX language. A sample of the possible escape sequences is shown in Table 3.6; the full set of possibilities is included in the MATLAB on-line documentation.

If one of the special escape characters `\`, `(`, `)`, `_`, or `^` must be printed, precede it by a backslash character.

The following examples illustrate the use of stream modifiers and special characters.

String	Result
<code>\tau_{ind} versus \omega_{itm}</code>	$\tau_{ind}$ versus $\omega_m$
<code>\theta varies from 0\circ to 90\circ</code>	$\theta$ varies from $0^\circ$ to $90^\circ$
<code>\bf{B}_{\itS}</code>	<b>B<sub>S</sub></b>

**Table 3.6 Selected Greek and Mathematical Symbols**

Character Sequence	Symbol	Character Sequence	Symbol	Character Sequence	Symbol
\alpha	$\alpha$			\int	$\int$
\beta	$\beta$			\cong	$\cong$
\gamma	$\gamma$	\Gamma	$\Gamma$	\sim	$\sim$
\delta	$\delta$	\Delta	$\Delta$	\infty	$\infty$
\epsilon	$\epsilon$			\pm	$\pm$
\eta	$\eta$			\leq	$\leq$
\theta	$\theta$			\geq	$\geq$
\lambda	$\lambda$	\Lambda	$\Lambda$	\neq	$\neq$
\mu	$\mu$			\propto	$\propto$
\nu	$\nu$			\div	$\div$
\pi	$\pi$	\Pi	$\Pi$	\circ	$\circ$
\phi	$\phi$			\leftrightarrow	$\leftrightarrow$
\rho	$\rho$			\leftarrow	$\leftarrow$
\sigma	$\sigma$	\Sigma	$\Sigma$	\rightarrow	$\rightarrow$
\tau	$\tau$			\uparrow	$\uparrow$
\omega	$\omega$	\Omega	$\Omega$	\downarrow	$\downarrow$

### Polar Plots

MATLAB includes a special function called `polar`, which plots data in polar coordinates. The basic form of this function is

```
polar(theta,r)
```

where `theta` is an array of angles in radians, and `r` is an array of distances. It is useful for plotting data that is intrinsically a function of angle.

#### Example 3.5—Cardioid Microphone

Most microphones designed for use on a stage are directional microphones, which are specifically built to enhance the signals received from the singer in the front of the microphone while suppressing the audience noise from behind the microphone. The gain of such a microphone varies as a function of angle according to the equation

$$\text{Gain} = 2g(1 + \cos\theta) \quad (3-3)$$

where  $g$  is a constant associated with a particular microphone, and  $\theta$  is the angle from the axis of the microphone to the sound source. Assume that  $g$  is 0.5 for a

particular microphone, and make a polar plot the gain of the microphone as a function of the direction of the sound source.

**SOLUTION** We must calculate the gain of the microphone versus angle and then plot it with a polar plot. The MATLAB code to do this is shown below.

```
% Script file: microphone.m
%
% Purpose:
%   This program plots the gain pattern of a cardioid
%   microphone.
%
% Record of revisions:
%   Date           Programmer           Description of change
%   ====          =====
%   01/05/04      S. J. Chapman           Original code
%
% Define variables:
%   g             -- Microphone gain constant
%   gaint         -- Gain as a function of angle
%   theta         -- Angle from microphone axis (radians)

% Calculate gain versus angle
g = 0.5;
theta = 0:pi/20:2*pi;
gain = 2*g*(1+cos(theta));

% Plot gain
polar (theta,gain,'r-');
title ('\bfGain versus angle \theta');
```

The resulting plot is shown in Figure 3.9 on page 127. Note that this type of microphone is called a “cardioid microphone” because its gain pattern is heart-shaped.

### Example 3.6—Electrical Engineering: Frequency Response of a Low-Pass Filter

A simple low-pass filter circuit is shown in Figure 3.10. This circuit consists of a resistor and capacitor in series, and the ratio of the output voltage  $V_o$  to the input voltage  $V_i$  is given by the equation

$$\frac{V_o}{V_i} = \frac{1}{1 + j2\pi fRC} \quad (3-4)$$

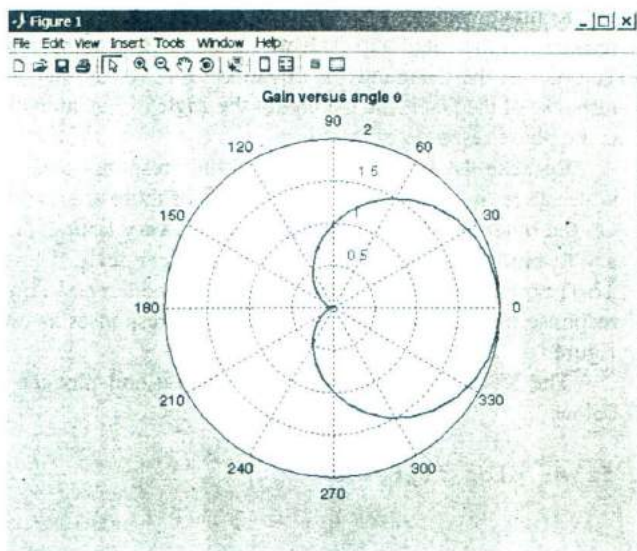


Figure 3.9 Gain of a cardioid microphone.

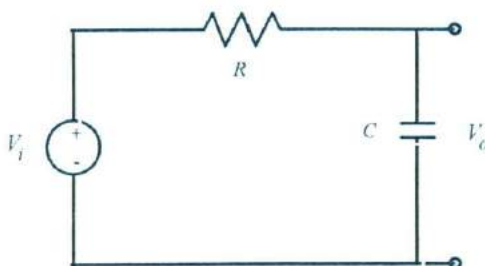


Figure 3.10 A simple low-pass filter circuit.

where  $V_i$  is a sinusoidal input voltage of frequency  $f$ ,  $R$  is the resistance in ohms,  $C$  is the capacitance in farads, and  $j$  is  $\sqrt{-1}$  (electrical engineers use  $j$  instead of  $i$  for  $\sqrt{-1}$ , because the letter  $i$  is traditionally reserved for the current in a circuit).

Assume that the resistance  $R = 16 \text{ k}\Omega$ , and capacitance  $C = 1 \text{ }\mu\text{F}$ , and plot the amplitude and frequency response of this filter.

**SOLUTION** The amplitude response of a filter is the ratio of the amplitude of the output voltage to the amplitude of the input voltage, and the phase response of the filter is the difference between the phase of the output voltage and the phase

of the input voltage. The simplest way to calculate the amplitude and phase response of the filter is to evaluate Equation (3-4) at many different frequencies. The plot of the magnitude of Equation (3-4) versus frequency is the amplitude response of the filter, and the plot of the angle of Equation (3-4) versus frequency is the phase response of the filter.

Because the frequency and amplitude response of a filter can vary over a wide range, it is customary to plot both of these values on logarithmic scales. On the other hand, the phase varies over a very limited range, so it is customary to plot the phase of the filter on a linear scale. Therefore, we will use a loglog plot for the amplitude response, and a semilogx plot for the phase response of the filter. We will display both responses as two sub-plots within a figure.

The MATLAB code required to create and plot the responses is shown below.

```
% Script file: plot_filter.m
%
% Purpose:
%   This program plots the amplitude and phase responses
%   of a low-pass RC filter.
%
% Record of revisions:
%   Date           Programmer           Description of change
%   ----           -
%   01/05/04      S. J. Chapman           Original code
%
% Define variables:
%   amp           -- Amplitude response
%   C             -- Capacitance (farads)
%   f             -- Frequency of input signal (Hz)
%   phase        -- Phase response
%   R             -- Resistance (ohms)
%   res          -- Vo/Vi

% Initialize R & C
R = 16000;           % 16 k ohms
C = 1.0E-6;         % 1 uF

% Create array of input frequencies
f = 1:2:1000;

% Calculate response
res = 1 ./ ( 1 + j*2*pi*f*R*C );
% Calculate amplitude response
amp = abs(res);
```



```

% Calculate phase response
phase = angle(res);

% Create plots
subplot(2,1,1);
loglog( f, amp );
title('Amplitude Response');
xlabel('Frequency (Hz)');
ylabel('Output/Input Ratio');
grid on;

subplot(2,1,2);
semilogx( f, phase );
title('Phase Response');
xlabel('Frequency (Hz)');
ylabel('Output-Input Phase (rad)');
grid on;

```

The resulting amplitude and phase responses are shown in Figure 3.11. Note that this circuit is called a low-pass filter because low frequencies are passed through with little attenuation, while high frequencies are strongly attenuated. ◀

### Example 3.7—Thermodynamics: The Ideal Gas Law

An ideal gas is one in which all collisions between molecules are perfectly elastic. It is possible to think of the molecules in an ideal gas as perfectly hard billiard balls that collide and bounce off of each other without losing kinetic energy.

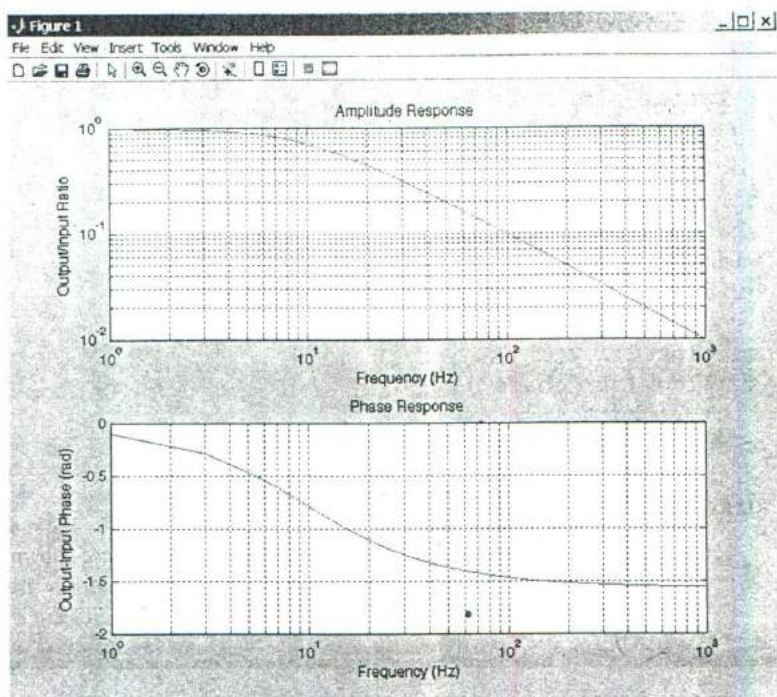
Such a gas can be characterized by three quantities: absolute pressure ( $P$ ), volume ( $V$ ) and absolute temperature ( $T$ ). The relationship among these quantities in an ideal gas is known as the Ideal Gas Law:

$$PV = nRT \quad (3-5)$$

where  $P$  is the pressure of the gas in kilopascals (kPa),  $V$  is the volume of the gas in liters (L),  $n$  is the number of molecules of the gas in units of moles (mol),  $R$  is the universal gas constant (8.314 L·kPa/mol·K), and  $T$  is the absolute temperature in kelvins (K). (Note: 1 mol =  $6.02 \times 10^{23}$  molecules)

Assume that a sample of an ideal gas contains 1 mole of molecules at a temperature of 273 K, and answer the following questions.

- (a) How does the volume of this gas vary as its pressure varies from 1 to 1000 kPa? Plot pressure versus volume for this gas on an appropriate set of axes. Use a solid red line, with a width of 2 pixels.



**Figure 3.11** The amplitude and phase response of the low-pass filter circuit.

- (b) Suppose that the temperature of the gas is increased to 373 K. How does the volume of this gas vary with pressure now? Plot pressure versus volume for this gas on an the same set of axes as part (a). Use a dashed blue line, with a width of 2 pixels.

Include a bold face title and  $x$ - and  $y$ -axis labels on the plot, as well as legends for each line.

**SOLUTION** The values that we wish to plot both vary by a factor of 1000, so an ordinary linear plot will not produce a useful plot. Therefore, we will plot the data on a log-log scale.

Note that we must plot two curves on the same set of axes, so we must issue the command `hold on` after the first one is plotted, and `hold off` after the plot is complete. It will also be necessary to specify the color, style, and width of each line, and to specify that labels be in bold face.

A program that calculates the volume of the gas as a function of pressure and creates the appropriate plot is shown below. Note that the special features controlling the style of the plot are shown in bold face.

```

% Script file: ideal_gas.m
%
% Purpose:
%   This program plots the pressure versus volume of an
%   ideal gas.
%
% Record of revisions:
%   Date           Programmer           Description of change
%   ====          =====
%   01/05/04      S. J. Chapman           Original code
%
% Define variables:
%   n             -- Number of atoms (mol)
%   P             -- Pressure (kPa)
%   R             -- Ideal gas constant (L kPa/mol K)
%   T             -- Temperature (K)
%   V             -- volume (L)

% Initialize nRT
n = 1;           % Moles of atoms
R = 8.314;       % Ideal gas constant
T = 273;        % Temperature (K)

% Create array of input pressures. Note that this
% array must be quite dense to catch the major
% changes in volume at low pressures.
P = 1:0.1:1000;

% Calculate volumes
V = (n * R * T) ./ P;

% Create first plot
figure(1);
loglog(P, V, 'r-', 'LineWidth', 2);
title('\bfVolume vs Pressure in an Ideal Gas');
xlabel('\bfPressure (kPa)');
ylabel('\bfVolume (L)');
grid on;
hold on;

% Now increase temperature
T = 373;        % Temperature (K)

% Calculate volumes
V = (n * R * T) ./ P;

```

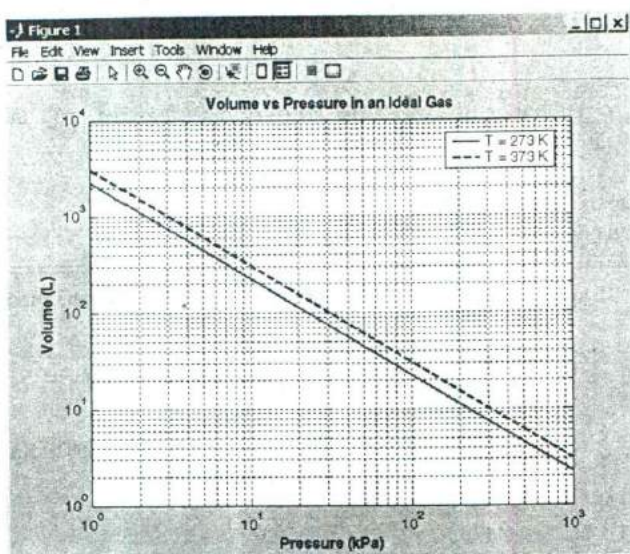



Figure 3.12 Pressure versus volume for an ideal gas.

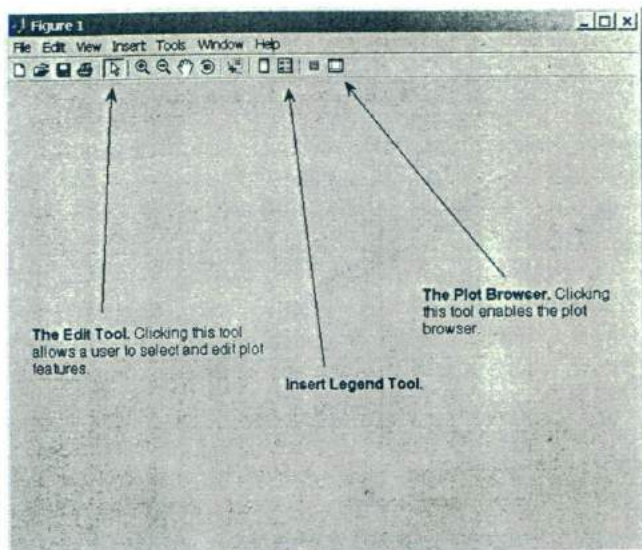
```
% Add second line to plot
figure(1);
loglog( P, V, 'b--', 'LineWidth', 2 );
hold off;

% Add legend
legend( 'T = 273 K', 'T = 373 K' );
```

The resulting volume versus pressure plot shown in Figure 3.12.

## Annotating and Saving Plots

Once a plot has been created by a MATLAB program, a user can edit and annotate the plot using the GUI-based tools available from the plot toolbar. Figure 3.13 shows the tools available, which allow the user to edit the properties of any objects on the plot, or to add annotations to the plot. When the editing button () is selected from the toolbar, the editing tools become available for use. When the button is depressed, clicking any line or text on the figure will cause it to be selected for editing, and double-clicking the line or text will open a Property Editor window that allows you to modify any or all of the characteristics of that object. Figure 3.14 shows Figure 3.12 after a user has clicked on the blue line to change it to a 3-pixel-wide dashed line.



**Figure 3.13** The editing tools on the figure toolbar.

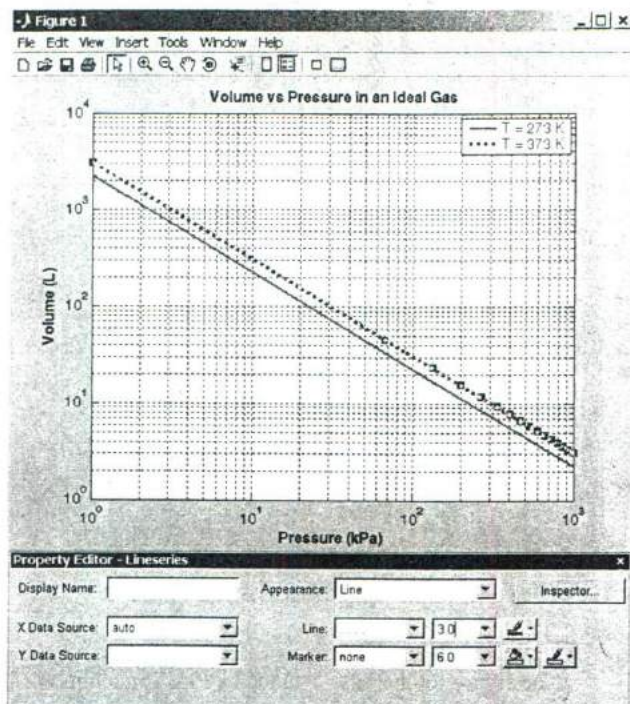
The figure toolbar also includes a Plot Tools button (□). When this button is depressed, the Plot Browser is displayed. This tool gives the user complete control over the figure. He or she can add axes, edit object properties, modify data values, and add annotations such as lines and text boxes. Figure 3.15 on page 135 shows Figure 3.12 after the user has added an arrow and annotation to the plot.

When the plot has been edited and annotated, you can save the entire plot in a modifiable form using the "File/Save As" menu item from the Figure Window. The resulting figure file (\*.fig) contains all the information required to re-create the figure plus annotations at any time in the future.

### Quiz 3.3

This quiz provides a quick check to see if you have understood the concepts introduced in Section 3.5. If you have trouble with the quiz, reread the section, ask your instructor, or discuss the material with a fellow student. The answers to this quiz are found in the back of the book.

1. Write the MATLAB statements required to plot  $\sin x$  versus  $\cos 2x$  from 0 to  $2\pi$  in steps of  $\pi/10$ . The points should be connected by a 2-pixel-wide red line, and each point should be marked with a 6-pixel-wide blue circular marker.
2. Use the Figure editing tools to change the markers on the previous plot into black squares. Add an arrow and annotation pointing to the location  $x = \pi$  on the plot.



**Figure 3.14** Figure 3.12 after the blue line has been modified using the editing tools built into the figure toolbar.

Write the MATLAB text string that will produce the following expressions:

3.  $f(x) = \sin \theta \cos 2\phi$
4. Plot of  $\sum x^2$  versus  $x$

Write the expression produced by the following text strings:

5. `'\tau\it_{m}'`
6. `'\bf\itx_{1}^{2} + x_{2}^{2} \rm(units: \bfm^{2} \rm)'`
7. How do you display the backslash (`\`) character in a text string?

## 3.6 More on Debugging MATLAB Programs

It is much easier to make a mistake when writing a program containing branches and loops than it is when writing simple sequential programs. Even after going through the full design process, a program of any size is almost guaranteed not to

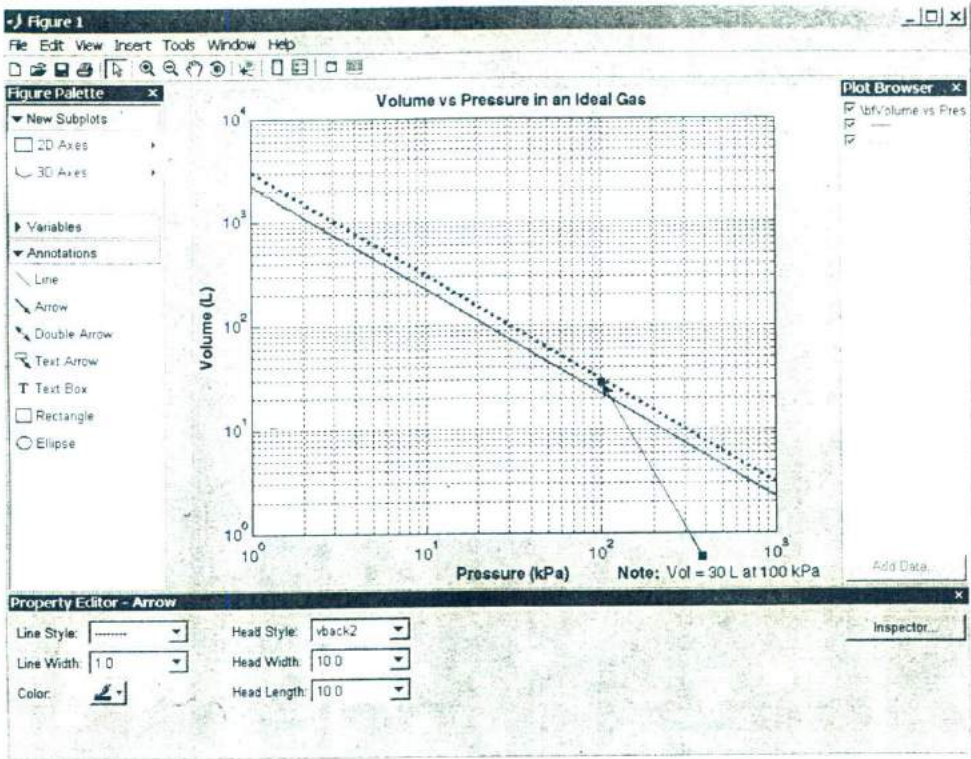


Figure 3.15 Figure 3.12 after the Plot Browser has been used to add an arrow and annotation.

be completely correct the first time it is used. Suppose that we have built the program and tested it, only to find that the output values are in error. How do we go about finding the bugs and fixing them?

Once programs start to include loops and branches, the best way to locate an error is to use the symbolic debugger supplied with MATLAB. This debugger is integrated with the MATLAB editor.

To use the debugger, first open the file that you would like to debug using the "File/Open" menu selection in the MATLAB Command Window. When the file is opened, it is loaded into the editor and the syntax is automatically color-coded. Comments in the file appear in green, variables and numbers appear in black, character strings appear in purple, and language keywords appear in blue. Figure 3.16 shows an example Edit/Debug window containing the file `calc_roots.m`.

Let's say that we would like to determine what happens when the program is executed. To do this, we can set one or more **breakpoints** by right-clicking the

```

7 %
8 % History of revisions:
9 %      Date      Programmer      Description of change
10 %      ----      -
11 %      01/02/94      R. J. Chapman      Original code
12 %
13 % Define Variables:
14 % a      -- Coefficient of x^2 term of equation
15 % b      -- Coefficient of x term of equation
16 % c      -- Constant term of equation
17 % discriminant -- Discriminant of the equation
18 % imag_part -- Imag part of equation (for complex roots)
19 % real_part  -- Real part of equation (for complex roots)
20 % x1      -- First solution of equation (for real roots)
21 % x2      -- Second solution of equation (for real roots)
22 %
23 % Prompt the user for the coefficients of the equation
24 disp ('This program solves for the roots of a quadratic ');
25 disp ('equation of the form A*x^2 + B*x + C = 0. ');
26 a = input ('Enter the coefficient A: ');
27 b = input ('Enter the coefficient B: ');
28 c = input ('Enter the coefficient C: ');
29 %
30 % Calculate Discriminant
31 discriminant = b^2 - 4 * a * c;
32 %
33 % Solve for the roots, depending on the value of the discriminant
34 if discriminant > 0 % there are two real roots, so...
35 %
36     x1 = (-b + sqrt(discriminant)) / (2 * a);
37     x2 = (-b - sqrt(discriminant)) / (2 * a);
38     disp ('This equation has two real roots:');
39     fprintf ('x1 = %f\n', x1);
40     fprintf ('x2 = %f\n', x2);

```

Figure 3.16 An Edit/Debug window with a MATLAB program loaded.

mouse on the lines of interest and choosing the “Set/Clear Breakpoint” option. When a breakpoint is set, a red dot appears to the left of that line containing the breakpoint, as shown in Figure 3.17.

Once the breakpoints have been set, execute the program as usual by typing `calc_roots` in the Command Window. The program will run until it reaches the first breakpoint and stop there. A green arrow will appear by the current line during the debugging process, as shown in Figure 3.18. When the breakpoint is reached, the programmer can examine and/or modify any variable in the workspace by typing its name in the Command Window. When the programmer is satisfied with the program at that point, he/she can either step through the program a line at a time by repeatedly pressing F10, or else run to the next breakpoint by



```

7 %
8 % Receipt of revisions:
9 %   Date      Programmer      Description of change
10 %   ----      -
11 %   01-01-04  S. J. Chapman      Original code.
12 %
13 % Define variables:
14 % a          -- Coefficient of x^2 term of equation
15 % b          -- Coefficient of x term of equation
16 % c          -- Constant term of equation
17 % discriminant -- Discriminant of the equation
18 % imag_part  -- Imag part of equation (for complex roots)
19 % real_part  -- Real part of equation (for complex roots)
20 % x1         -- First solution of equation (for real roots)
21 % x2         -- Second solution of equation (for real roots)
22 %
23 % Prompt the user for the coefficients of the equation
24 disp ('This program solves for the roots of a quadratic ');
25 disp ('equation of the form A*X^2 + B*X + C = 0. ');
26 a = input ('Enter the coefficient A: ');
27 b = input ('Enter the coefficient B: ');
28 c = input ('Enter the coefficient C: ');
29 %
30 % Calculate discriminant
31 • discriminant = b^2 - 4 * a * c;
32 %
33 % Solve for the roots, depending on the value of the discriminant
34 if discriminant > 0 % there are two real roots, so...
35 %
36     x1 = (-b + sqrt(discriminant)) / (2 * a);
37     x2 = (-b - sqrt(discriminant)) / (2 * a);
38     disp ('This equation has two real roots:');
39     fprintf ('x1 = %f\n', x1);
40     fprintf ('x2 = %f\n', x2);

```

**Figure 3.17** The window after a breakpoint has been set. Note the red dot to the left of the line with the breakpoint.

pressing F5. It is always possible to examine the values of any variable at any point in the program.

When a bug is found, the programmer can use the Editor to correct the MATLAB program and save the modified version to disk. Note that all breakpoints may be lost when the program is saved to disk, so they may have to be set again before debugging can continue. This process is repeated until the program appears to be bug-free.

Two other very important features of the debugger are found on the “Breakpoints” menu. The first feature is “Set/Modify Conditional Breakpoint.” A **conditional breakpoint** is a breakpoint where the code only stops if some condition is true. For example, a conditional breakpoint can be used to stop execution inside a `for` loop on its 200<sup>th</sup> execution. This can be very important if a

```

7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40

```

% Example 3.18: Solving a Quadratic Equation  
 % This program solves for the roots of a quadratic equation of the form  
 %  $AX^2 + BX + C = 0$ . The roots are calculated using the quadratic formula.  
 % The discriminant is calculated as  $b^2 - 4ac$ . If the discriminant is positive,  
 % there are two real roots. If it is zero, there is one real root. If it is  
 % negative, there are two complex roots.

% Coefficient A  
 % Coefficient B  
 % Coefficient C  
 % Discriminant  
 % Imaginary part  
 % Real part  
 % First solution of equation (for real roots)  
 % Second solution of equation (for real roots)

% Prompt the user for the coefficients of the equation  
 disp('This program solves for the roots of a quadratic. ');  
 disp('Equation of the form  $AX^2 + BX + C = 0$ . ');  
 a = input('Enter the coefficient A: ');  
 b = input('Enter the coefficient B: ');  
 c = input('Enter the coefficient C: ');

% Calculate discriminant  
 discriminant = b^2 - 4 \* a \* c;

% Solve for the roots, depending on the value of the discriminant  
 if discriminant > 0 % there are two real roots, so...  
 x1 = (-b + sqrt(discriminant)) / (2 \* a);  
 x2 = (-b - sqrt(discriminant)) / (2 \* a);  
 disp('This equation has two real roots:');  
 fprintf('x1 = %f\n', x1);  
 fprintf('x2 = %f\n', x2);

**Figure 3.18** A green arrow will appear by the current line during the debugging process.

bug only appears after a loop has been executed many times. The condition that causes the breakpoint to stop execution can be modified, and the breakpoint can be enabled or disabled during debugging.

The second feature is “Set Error Breakpoints for All Files.” If an error is occurring in a program that causes it to crash or generate warning messages, the programmer can turn this item on and execute the program. It will run to the point of the error and stop there, allowing the programmer to examine the values of variables and exactly what is causing the problem.

A final critical feature is found on the “Debug” menu. It is “Check Code with M-Lint.” M-Lint is a program that examines one or more M-files and reports any examples of improper or questionable usage. It is a *great* tool for locating errors, poor usage, or obsolete features in MATLAB code, including such

things as variables that are defined but never used. You should always run M-Lint over your programs when they are finished as a final check that everything has been done properly.

Take some time now to become familiar with the Editor/Debugger and its supporting tools—it is a very worthwhile investment.

## 3.7 Summary

In Chapter 3 we have presented the basic types of MATLAB branches and the relational and logic operations used to control them. The principal type of branch is the `if` construct. This construct is very flexible. It can have as many `elseif` clauses as needed to construct any desired test. Furthermore, `if` constructs can be nested to produce more complex tests. A second type of branch is the `switch` construct. It may be used to select among mutually exclusive alternatives specified by a control expression. A third type of branch is the `try/catch` construct. It is used to trap errors that might occur during execution.

Chapter 3 also included additional information about plots. The `axis` command allows a programmer to select the specific range of  $x$  and  $y$  data to be plotted. The `hold` command allows later plots to be plotted on top of earlier ones, so that elements can be added to a graph a piece at a time. The `figure` command allows the programmer to create and select among multiple Figure Windows, so that a program can create multiple plots in separate windows. The `subplot` command allows the programmer to create and select among multiple plots within a single Figure Window.

In addition, we learned how to control additional characteristics of our plots, such as the line width and marker color. These properties may be controlled by specifying 'PropertyName', value pairs in the plot command after the data to be plotted.

Text strings in plots may be enhanced with stream modifiers and escape sequences. Stream modifiers allow a programmer to specify features like bold face, italic, superscripts, subscripts, font size, and font name. Escape sequences allow the programmer to include special characters such as Greek and mathematical symbols in the text string.

The MATLAB symbolic debugger and related tools such as M-Lint make debugging MATLAB code much easier. You should invest some time to become familiar with these tools.

### Summary of Good Programming Practice

The following guidelines should be adhered to when programming with branch or loop constructs. By following them consistently, your code will contain fewer bugs, will be easier to debug, and will be more understandable to others who may need to work with it in the future.

1. Follow the steps of the program design process to produce reliable, understandable MATLAB programs.
2. Be cautious about testing for equality with numeric values, since roundoff errors may cause two variables that should be equal to fail a test for equality. Instead, test to see if the variables are *nearly* equal within the roundoff error to be expected on the computer you are working with.
3. Use the `&` AND operator if it is necessary to ensure that both operands are evaluated in an expression, or if the comparison is between arrays. Otherwise, use the `&&` AND operator, since the partial evaluation will make the operation faster in the cases where the first operand is `false`. The `&` operator is preferred in most practical cases.
4. Use the `|` inclusive OR operator if it is necessary to ensure that both operands are evaluated in an expression, or if the comparison is between arrays. Otherwise, use the `||` operator, since the partial evaluation will make the operation faster in the cases where the first operand is `true`. The `|` operator is preferred in most practical cases.
5. Always indent code blocks in `if`, `switch`, and `try/catch` constructs to make them more readable.
6. For branches in which there are many mutually exclusive options, use a single `if` construct with multiple `elseif` clauses in preference to nested `if` constructs.

## MATLAB Summary

The following summary lists all of the MATLAB commands and functions described in this chapter, along with a brief description of each one.

### Commands and Functions

---

<code>axis</code>	(a) Set the <i>x</i> and <i>y</i> limits of the data to be plotted. (b) Get the <i>x</i> and <i>y</i> limits of the data to be plotted. (c) Set other axis-related properties.
<code>figure</code>	Select a Figure Window to be the current Figure Window. If the selected Figure Window does not exist, it is automatically created.
<code>hold</code>	Allows multiple plot commands to write on top of each other.
<code>if</code> construct	Selects a block of statements to execute if a specified condition is satisfied.
<code>ischar(a)</code>	Returns a 1 if <i>a</i> is a character array and a 0 otherwise.
<code>isempty(a)</code>	Returns a 1 if <i>a</i> is an empty array and a 0 otherwise.
<code>isinf(a)</code>	Returns a 1 if the value of <i>a</i> is infinite ( <code>Inf</code> ) and a 0 otherwise.
<code>isnan(a)</code>	Returns a 1 if the value of <i>a</i> is NaN (not a number) and a 0 otherwise.
<code>isnumeric(a)</code>	Returns a 1 if the <i>a</i> is a numeric array and a 0 otherwise.
<code>logical</code>	Converts numeric data to logical data, with non-zero values becoming <code>true</code> and zero values becoming <code>false</code> .

## Commands and Functions

---

polar	Create a polar plot.
subplot	Select a subplot in the current Figure Window. If the selected subplot does not exist, it is automatically created. If the new subplot conflicts with a previously existing set of axes, they are automatically deleted.
switch construct	Selects a block of statements to execute from a set of mutually-exclusive choices based on the result of a single expression.
try/catch construct	A special construct used to trap errors. It executes construct the code in the try block. If an error occurs, execution stops immediately and transfers to the code in the catch construct.

---

## 3.8 Exercises

3.1 Evaluate the following MATLAB expressions.

- (a)  $5.5 \geq 5$
- (b)  $20 > 20$
- (c)  $\text{xor}(17 - \text{pi} < 15, \text{pi} < 3)$
- (d)  $\text{true} > \text{false}$
- (e)  $\sim(35/17) == (35/17)$
- (f)  $(7 \leq 8) == (3/2 == 1)$
- (g)  $17.5 \&\& (3.3 > 2.)$

3.2 The tangent function is defined as  $\tan\theta = \sin\theta/\cos\theta$ . This expression can be evaluated to solve for the tangent as long as the magnitude of  $\cos\theta$  is not too near to 0. (If  $\cos\theta$  is 0, evaluating the equation for  $\tan\theta$  will produce the non-numerical value *Inf*.) Assume that  $\theta$  is given in *degrees*, and write the MATLAB statements to evaluate  $\tan\theta$  as long as the magnitude of  $\cos\theta$  is greater than or equal to  $10^{-20}$ . If the magnitude of  $\cos\theta$  is less than  $10^{-20}$ , write out an error message instead.

3.3 The following statements are intended to alert a user to dangerously high oral thermometer readings (values are in degrees Fahrenheit). Are they correct or incorrect? If they are incorrect, explain why and correct them.

```
if temp < 97.5
    disp('Temperature below normal');
elseif temp > 97.5
    disp('Temperature normal');
    elseif temp > 99.5
        disp('Temperature slightly high');
elseif temp > 103.0
    disp('Temperature dangerously high');
end
```

- 3.4** The cost of sending a package by an express delivery service is \$12.00 for the first two pounds, and \$4.50 for each pound or fraction thereof over two pounds. If the package weighs more than 70 pounds, a \$15.00 excess weight surcharge is added to the cost. No package over 100 pounds will be accepted. Write a program that accepts the weight of a package in pounds and computes the cost of mailing the package. Be sure to handle the case of overweight packages.
- 3.5** In Example 3.3, we wrote a program to evaluate the function  $f(x, y)$  for any two user-specified values  $x$  and  $y$ , where the function  $f(x, y)$  was defined as follows.

$$f(x, y) = \begin{cases} x + y & x \geq 0 \text{ and } y \geq 0 \\ x + y^2 & x \geq 0 \text{ and } y < 0 \\ x^2 + y & x < 0 \text{ and } y \geq 0 \\ x^2 + y^2 & x < 0 \text{ and } y < 0 \end{cases}$$

The problem was solved by using a single `if` construct with four code blocks to calculate  $f(x, y)$  for all possible combinations of  $x$  and  $y$ . Rewrite program `funxy` to use nested `if` constructs, where the outer construct evaluates the value of  $x$  and the inner constructs evaluate the value of  $y$ .

- 3.6** Write a MATLAB program to evaluate the function

$$y(x) = \ln \frac{1}{1-x}$$

for any user-specified value of  $x$ , where  $x$  is a number  $< 1.0$  (note that  $\ln$  is the natural logarithm, the logarithm to the base  $e$ ). Use an `if` structure to verify that the value passed to the program is legal. If the value of  $x$  is legal, calculate  $y(x)$ . If not, write a suitable error message and quit.

- 3.7** Write a program that allows a user to enter a string containing a day of the week ('Sunday', 'Monday', 'Tuesday', etc.), and uses that a `switch` construct to convert the day to its corresponding number, where Sunday is considered the first day of the week and Saturday is considered the last day of the week. Print out the resulting day number. Also, be sure to handle the case of an illegal day name! (*Note:* Be sure to use the 's' option on function `input` so that the input is treated as a string.)
- 3.8** Suppose that a student has the option of enrolling for a single elective during a term. The student must select a course from a limited list of options: "English," "History," "Astronomy," or "Literature." Construct a fragment of MATLAB code that will prompt the student for his or her choice, read in the choice, and use the answer as the case expression for a `CASE` construct. Be sure to include a default case to handle invalid inputs.
- 3.9 Ideal Gas Law** The Ideal Gas Law was defined in Example 3.7. Assume that the volume of 1 mole of this gas is 10 L, and plot the pressure of the gas as a function of temperature as the temperature is changed from 250 to 400 kelvins. What sort of plot (linear, semilogx, etc.) is most appropriate for this data?

- 3.10 Antenna Gain Pattern** The gain  $G$  of a certain microwave dish antenna can be expressed as a function of angle by the equation

$$G(\theta) = |\operatorname{sinc} 4\theta| \quad \text{for } -\frac{\pi}{2} \leq \theta \leq \frac{\pi}{2} \quad (3-5)$$

where  $\theta$  is measured in radians from the boresite of the dish, and  $\operatorname{sinc} x = \sin x/x$ . Plot this gain function on a polar plot, with the title "Antenna Gain vs  $\theta$ " in bold face.

- 3.11** The author of this book now lives in Australia. Australia is a great place to live, but it is also a land of high taxes. In 2002, individual citizens and residents of Australia paid the following income taxes:

Taxable Income (in A\$)	Tax on This Income
\$0–\$6,000	Nil.
\$6,001–\$20,000	17¢ for each \$1 over \$6,000
\$20,001–\$50,000	\$2,380 plus 30¢ for each \$1 over \$20,000
\$50,001–\$60,000	\$11,380 plus 42¢ for each \$1 over \$50,000
Over \$60,000	\$15,580 plus 47¢ for each \$1 over \$60,000

In addition, a flat 1.5% Medicare levy is charged on all income. Write a program to calculate how much income tax a person will owe based on this information. The program should accept a total income figure from the user, and calculate the income tax, Medicare Levy, and total tax payable by the individual.

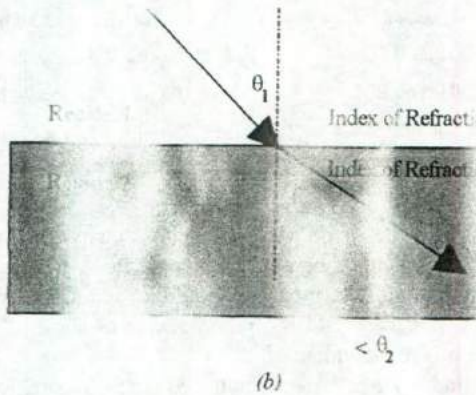
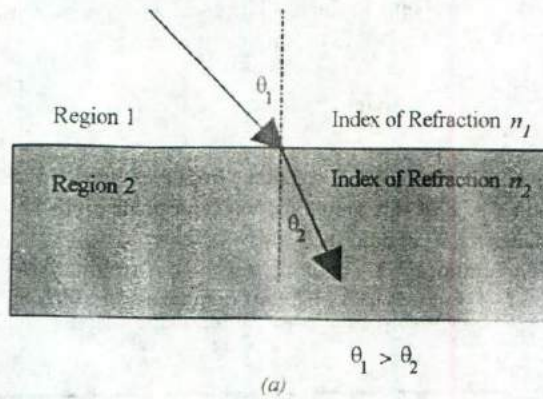
- 3.12 Refraction** When a ray of light passes from a region with an index of refraction  $n_1$  into a region with a different index of refraction  $n_2$ , the light ray is bent (see Figure 3.19). The angle at which the light is bent is given by *Snell's Law*

$$n_1 \sin \theta_1 = n_2 \sin \theta_2 \quad (3-6)$$

where  $\theta_1$  is the angle of incidence of the light in the first region, and  $\theta_2$  is the angle of incidence of the light in the second region. Using Snell's Law, it is possible to predict the angle of incidence of a light ray in Region 2 if the angle of incidence  $\theta_1$  in Region 1 and the indices of refraction  $n_1$  and  $n_2$  are known. The equation to perform this calculation is

$$\theta_2 = \sin^{-1}\left(\frac{n_1}{n_2} \sin \theta_1\right) \quad (3-7)$$

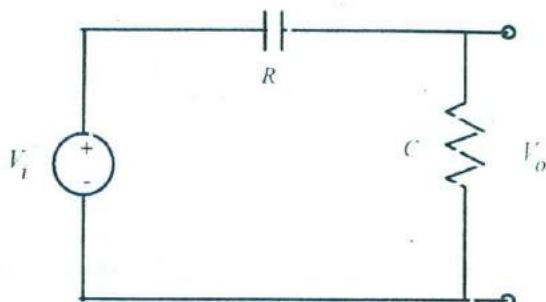
Write a program to calculate the angle of incidence (in degrees) of a light ray in Region 2 given the angle of incidence  $\theta_1$  in Region 1 and the indices of refraction  $n_1$  and  $n_2$ . (Note: If  $n_1 > n_2$ , then for some angles  $\theta_1$ , Equation (3-7) will have no real solution because the absolute value of the quantity



**Figure 3.19** A ray of light passes from a region with a higher index of refraction into a region with a lower index of refraction. (a) If the ray of light passes from a region with a higher index of refraction into a region with a lower index of refraction, the ray bends towards the vertical. (b) If the ray of light passes from a region with a lower index of refraction into a region with a higher index of refraction, the ray bends away from the vertical.

( ) will be greater than 1.0. When all light is reflected back into region 1, and no light passes into region 2, this is called total internal reflection. To properly handle this situation, your program should create a plot of the incident ray on the boundary, and the refracted ray on the boundary.





**Figure 3.20** A simple high-pass filter circuit.

Test your program by running it for the following two cases: (a)  $n_1 = 1.0$ ,  $n_2 = 1.7$ , and  $\theta_1 = 45^\circ$ . (b)  $n_1 = 1.7$ ,  $n_2 = 1.0$ , and  $\theta_1 = 45^\circ$ .

- 3.13** Assume that the complex function  $f(t)$  is defined by the equation

$$f(t) = (0.5 - 0.25i)t - 1.0$$

Plot the amplitude and phase of function  $f$  for  $0 \leq t \leq 4$ .

- 3.14 High-Pass Filter** Figure 3.20 shows a simple high-pass filter consisting of a resistor and a capacitor. The ratio of the output voltage  $V_o$  to the input voltage  $V_i$  is given by the equation

$$\frac{V_o}{V_i} = \frac{j2\pi fRC}{1 + j2\pi fRC} \quad (3-8)$$

Assume that  $R = 16 \text{ k}\Omega$  and  $C = 1 \text{ }\mu\text{F}$ . Calculate and plot the amplitude and phase response of this filter as a function of frequency.

- 3.15 The Spiral of Archimedes** The spiral of Archimedes is a curve described in polar coordinates by the equation

$$r = k\theta \quad (3-9)$$

where  $r$  is the distance of a point from the origin, and  $\theta$  is the angle of that point in radians with respect to the origin. Plot the spiral of Archimedes for  $0 \leq \theta \leq 6\pi$  when  $k = 0.5$ . Be sure to label you plot properly.

- 3.16 Output Power from a Motor** The output power produced by a rotating motor is given by the equation

$$P = \tau_{\text{IND}} \omega_m \quad (3-10)$$

where  $\tau_{\text{IND}}$  is the induced torque on the shaft in newton-meters,  $\omega_m$  is the rotational speed of the shaft in radians per second, and  $P$  is in watts. Assume that the rotational speed of a particular motor shaft is given by the equation

$$\omega_m = 188.5(1 - e^{-0.2t}) \text{ rad/s}$$

and the induced torque on the shaft is given by

$$\tau_{\text{IND}} = 10e^{-0.2t} \text{ N} \cdot \text{m}$$

Plot the torque, speed, and power supplied by this shaft versus time for  $0 \leq t \leq 10$  s. Be sure to label your plot properly with the symbols  $\tau_{\text{IND}}$  and  $\omega_m$  where appropriate. Create two plots, one with the power displayed on a linear scale, and one with the output power displayed on a logarithmic scale. Time should always be displayed on a linear scale.

- 3.17 Plotting Orbits** When a satellite orbits the Earth, the satellite's orbit will form an ellipse with the Earth located at one of the focal points of the ellipse. The satellite's orbit can be expressed in polar coordinates as

$$r = \frac{p}{1 - \epsilon \cos \theta} \quad (3-11)$$

where  $r$  and  $\theta$  are the distance and angle of the satellite from the center of the Earth,  $p$  is a parameter specifying the size of the orbit, and  $\epsilon$  is a parameter representing the eccentricity of the orbit. A circular orbit has an eccentricity  $\epsilon$  of 0. An elliptical orbit has an eccentricity of  $0 \leq \epsilon \leq 1$ . If  $\epsilon > 1$ , the satellite follows a hyperbolic path and escapes from the Earth's gravitational field.

Consider a satellite with a size parameter  $p = 1000$  km. Plot the orbit of this satellite if (a)  $\epsilon = 0$ ; (b)  $\epsilon = 0.25$ ; (c)  $\epsilon = 0.5$ . How close does each orbit come to the Earth? How far away does each orbit get from the Earth? Compare the three plots you created. Can you determine what the parameter  $p$  means from looking at the plots?

## Loops

---

**Loops** are MATLAB constructs that permit us to execute a sequence of statements more than once. There are two basic forms of loop constructs: **while loops** and **for loops**. The major difference between these two types of loops is in how the repetition is controlled. The code in a **while** loop is repeated an indefinite number of times until some user-specified condition is satisfied. By contrast, the code in a **for** loop is repeated a specified number of times, and the number of repetitions is known before the loops starts.

### 4.1 The while Loop

A **while loop** is a block of statements that are repeated indefinitely as long as some condition is satisfied. The general form of a while loop is

```
while expression
    ...
    ...
    ...
end
```

} Code block

The controlling expression produces a logical value. If the *expression* is **true**, the code block will be executed, and then control will return to the **while** statement. If the *expression* is still **true**, the statements will be executed again. This process will be repeated until the *expression* becomes **false**. When control returns to the **while** statement and the expression is **false**, the program will execute the first statement after the **end**.

The pseudocode corresponding to a while loop is

```
while expr
    ...
    ...
    ...
end
```

We will now show an example statistical analysis program that is implemented using a while loop.

### Example 4.1—Statistical Analysis

It is very common in science and engineering to work with large sets of numbers, each of which is a measurement of some particular property that we are interested in. A simple example would be the grades on the first test in this course. Each grade would be a measurement of how much a particular student has learned in the course to date.

Much of the time, we are not interested in looking closely at every single measurement that we make. Instead, we want to summarize the results of a set of measurements with a few numbers that tell us a lot about the overall data set. Two such numbers are the *average* (or *arithmetic mean*) and the *standard deviation* of the set of measurements. The average or arithmetic mean of a set of numbers is defined as

$$\bar{x} = \frac{1}{N} \sum_{i=1}^N x_i \quad (4-1)$$

where  $x_i$  is sample  $i$  out of  $N$  samples. If all of the input values are available in an array, the average of a set of numbers can be calculated by the MATLAB function `mean`. The standard deviation of a set of numbers is defined as

$$s = \sqrt{\frac{N \sum_{i=1}^N x_i^2 - \left( \sum_{i=1}^N x_i \right)^2}{N(N-1)}} \quad (4-2)$$

Standard deviation is a measure of the amount of scatter on the measurements; the greater the standard deviation, the more scattered the points in the data set are. If all of the input values are available in an array, the standard deviation of a set of numbers can be calculated by MATLAB function `std`.

Implement an algorithm that reads in a set of measurements and calculates the mean and the standard deviation of the input data set.

**SOLUTION** This program must be able to read in an arbitrary number of measurements, and then calculate the mean and standard deviation of those measurements. We will use a while loop to accumulate the input measurements before performing the calculations.

When all of the measurements have been read, we must have some way of telling the program that there is no more data to enter. For now, we will assume that all the input measurements are either positive or zero, and we will use a negative input value as a *flag* to indicate that there is no more data to read. If a negative value is entered, then the program will stop reading input values and will calculate the mean and standard deviation of the data set.

### 1. State the problem.

Since we assume that the input numbers must be positive or zero, a proper statement of this problem would be: *calculate the average and the standard deviation of a set of measurements, assuming that all of the measurements are either positive or zero, and assuming that we do not know in advance how many measurements are included in the data set. A negative input value will mark the end of the set of measurements.*

### 2. Define the inputs and outputs.

The inputs required by this program are an unknown number of positive or zero numbers. The outputs from this program are a printout of the mean and the standard deviation of the input data set. In addition, we will print out the number of data points input to the program, since this is a useful check that the input data was read correctly.

### 3. Design the algorithm.

This program can be broken down into three major steps

```
Accumulate the input data
Calculate the mean and standard deviation
Write out the mean, standard deviation, and number of points
```

The first major step of the program is to accumulate the input data. To do this, we will have to prompt the user to enter the desired numbers. When the numbers are entered, we will have to keep track of the number of values entered, plus the sum and the sum of the squares of those values. The pseudocode for these steps is:

```
Initialize n, sum_x, and sum_x2 to 0
Prompt user for first number
Read in first x
while x >= 0
  n <- n + 1
  sum_x <- sum_x + x
  sum_x2 <- sum_x2 + x^2
  Prompt user for next number
  Read in next x
end
```

Note that we have to read in the first value before the while loop starts so that the while loop can have a value to test the first time it executes.

Next, we must calculate the mean and standard deviation. The pseudocode for this step is just the MATLAB versions of Equations (4-1) and (4-2).

```
x_bar <- sum_x / n
std_dev <- sqrt((n*sum_x2 - sum_x^2)/(n*(n-1)))
```

Finally, we must write out the results.

```
Write out the mean value x_bar
Write out the standard deviation std_dev
Write out the number of input data points n
```

#### 4. Turn the algorithm into MATLAB statements.

The final MATLAB program is shown below.

```
% Script file: stats_1.m
%
% Purpose:
% To calculate mean and the standard deviation of
% an input data set containing an arbitrary number
% of input values.
%
% Record of revisions:
%
% Date          Programmer          Description of change
% =====
% 01/07/04     S. J. Chapman          Original code
%
% Define variables:
% n            -- The number of input samples
% std_dev     -- The standard deviation of the input samples
% sum_x       -- The sum of the input values
% sum_x2      -- The sum of the squares of the input values
% x           -- An input data value
% xbar        -- The average of the input samples
%
% Initialize sums.
n = 0; sum_x = 0; sum_x2 = 0;
%
% Read in first value
x = input('Enter first value: ');
%
% While Loop to read input values.
while x >= 0
```

```

% Accumulate sums.
n      = n + 1;
sum_x  = sum_x + x;
sum_x2 = sum_x2 + x^2;

% Read in next value
x = input('Enter next value: ');

end

% Calculate the mean and standard deviation
x_bar = sum_x / n;
std_dev = sqrt((n * sum_x2 - sum_x^2)/(n * (n-1)));

% Tell user.
fprintf('The mean of this data set is:   %f\n', x_bar);
fprintf('The standard deviation is:      %f\n', std_dev);
fprintf('The number of data points is:   %f\n', n);

```

### 5. Test the program.

To test this program, we will calculate the answers by hand for a simple data set, and then compare the answers to the results of the program. If we used three input values: 3, 4, and 5, then the mean and standard deviation would be

$$\bar{x} = \frac{1}{N} \sum_{i=1}^N x_i = \frac{1}{3}(12) = 4$$

$$s = \sqrt{\frac{N \sum_{i=1}^N x_i^2 - \left(\sum_{i=1}^N x_i\right)^2}{N(N-1)}} = 1$$

When the above values are fed into the program, the results are

```

» stats_1
Enter first value: 3
Enter next value: 4
Enter next value: 5
Enter next value: -1
The mean of this data set is: 4.000000
The standard deviation is: 1.000000
The number of data points is: 3.000000

```

The program gives the correct answers for our test data set.

In the example above, we failed to follow the design process completely. This failure has left the program with a fatal flaw! Did you spot it?

We have failed because *we did not completely test the program for all possible types of inputs*. Look at the example once again. If we enter either no numbers or only one number, then we will be dividing by zero in the above equations! The division-by-zero error will cause divide-by-zero warnings to be printed, and the output values will be NaN. We need to modify the program to detect this problem, tell the user what the problem is, and stop gracefully.

A modified version of the program called `stats_2` is shown below. Here, we check to see if there are enough input values before performing the calculations. If not, the program will print out an intelligent error message and quit. Test the modified program for yourself.

```
% Script file: stats_2.m
%
% Purpose:
%   To calculate mean and the standard deviation of
%   an input data set containing an arbitrary number
%   of input values.
%
% Record of revisions:
%   Date      Programmer      Description of change
%   ====      =====
%   01/07/04   S. J. Chapman      Original code
% 1. 01/07/04   S. J. Chapman      Correct divide-by-0 error if
%                                     0 or 1 input values given.
%
% Define variables:
%   n          -- The number of input samples
%   std_dev    -- The standard deviation of the input samples
%   sum_x      -- The sum of the input values
%   sum_x2     -- The sum of the squares of the input values
%   x          -- An input data value
%   xbar       -- The average of the input samples

% Initialize sums.
n = 0; sum_x = 0; sum_x2 = 0;

% Read in first value
x = input('Enter first value: ');

% While Loop to read input values.
while x >= 0

    % Accumulate sums.
    n = n + 1;
```



```

sum_x = sum_x + x;
sum_x2 = sum_x2 + x^2;

% Read in next value
x = input('Enter next value: ');

end

% Check to see if we have enough input data.
if n < 2 % Insufficient information

    disp('At least 2 values must be entered!');

else % There is enough information, so
    % calculate the mean and standard deviation

    x_bar = sum_x / n;
    std_dev = sqrt((n * sum_x2 - sum_x^2)/(n * (n-1)));

    % Tell user.
    fprintf('The mean of this data set is: %f\n', x_bar);
    fprintf('The standard deviation is: %f\n', std_dev);
    fprintf('The number of data points is: %f\n', n);

end

```

Note that the average and standard deviation could have been calculated with the built-in MATLAB functions `mean` and `std` if all of the input values are saved in a vector, and that vector is passed to these functions. You will be asked to create a version of the program that uses the standard MATLAB functions in an exercise at the end of this chapter.

## 4.2 The for Loop

The **for loop** is a loop that executes a block of statements a specified number of times. The for loop has the form

```

for index = expr
    Statement 1
    ...
    Statement n
end

```

} Body

where `index` is the loop variable (also known as the **loop index**) and `expr` is the loop control expression. The columns in `expr` are stored one at a time in the variable `index` and then the loop body is executed, so that the loop is executed once for each column in `expr`. The expression usually takes the form of a vector in shortcut notation `first:incr:last`.

The statements between the `for` statement and the `end` statement are known as the *body* of the loop. They are executed repeatedly during each pass of the `for` loop. The `for` loop construct functions as follows:

1. At the beginning of the loop, MATLAB generates the control expression.
2. The first time through the loop, the program assigns the first column of the expression to the loop variable `index`, and the program executes the statements within the body of the loop.
3. After the statements in the body of the loop have been executed, the program assigns the next column of the expression to the loop variable `index`, and the program executes the statements within the body of the loop again.
4. Step 3 is repeated over and over as long as there are additional columns in the control expression.

Let's look at a number of specific examples to make the operation of the `for` loop clearer. First, consider the following example:

```
for ii = 1:10
    Statement 1
    ...
    Statement n
end
```

In this case, the control expression generates a  $1 \times 10$  array, so statements 1 through `n` will be executed 10 times. The loop index `ii` will be 1 on the first time, 2 on the second time, and so on. The loop index will be 10 on the last pass through the statements. When control is returned to the `for` statement after the tenth pass, there are no more columns in the control expression, so execution transfers to the first statement after the `end` statement. Note that the loop index `ii` is still set to 10 after the loop finishes executing.

Second, consider the following example:

```
for ii = 1:2:10
    Statement 1
    ...
    Statement n
end
```

In this case, the control expression generates a  $1 \times 5$  array, so statements 1 through `n` will be executed 5 times. The loop index `ii` will be 1 the first time, 3 the second time, and so on. The loop index will be 9 on the fifth and last pass through the statements. When control is returned to the `for` statement after the fifth pass, there are no more columns in the control expression, so execution

transfers to the first statement after the end statement. Note that the loop index *ii* is still set to 9 after the loop finishes executing.

Third, consider the following example:

```
for ii = [5 9 7]
    Statement 1
    ...
    Statement n
end
```

Here, the control expression is an explicitly written  $1 \times 3$  array, so statements 1 through *n* will be executed three times with the loop index set to 5 the first time, 9 the second time, and 7 the final time. The loop index *ii* is still set to 7 after the loop finishes executing.

Finally, consider the example:

```
for ii = [1 2 3; 4 5 6]
    Statement 1
    ...
    Statement n
end
```

In this case, the control expression is a  $2 \times 3$  array, so statements 1 through *n* will be executed three times. The loop index *ii* will be the column vector  $\begin{bmatrix} 1 \\ 4 \end{bmatrix}$  the first time,  $\begin{bmatrix} 2 \\ 5 \end{bmatrix}$  the second time, and  $\begin{bmatrix} 3 \\ 6 \end{bmatrix}$  the third time. The loop index *ii* is still set to  $\begin{bmatrix} 3 \\ 6 \end{bmatrix}$  after the loop finishes executing. This example illustrates the fact that a loop index can be a vector.

The pseudocode corresponding to a *for* loop looks like the loop itself:

```
for index = expression
    Statement 1
    ...
    Statement n
end
```

### Example 4.2—The Factorial Function

To illustrate the operation of a *for* loop, we will use a *for* loop to calculate the factorial function. The factorial function is defined as

$$\begin{array}{ll} N! = 1 & N = 0 \\ N! = N * (N-1) * (N-2) * \dots * 3 * 2 * 1 & N > 0 \end{array}$$

The MATLAB code to calculate  $N$  factorial for positive value of  $N$  would be

```
n_factorial = 1
for ii = 1:n
    n_factorial = n_factorial * ii;
end
```

Suppose that we wish to calculate the value of  $5!$ . If  $n$  is 5, the `for` loop control expression would be the row vector `[1 2 3 4 5]`. This loop will be executed 5 times, with the variable `ii` taking on values of 1, 2, 3, 4, and 5 in the successive loops. The resulting value of `n_factorial` will be  $1 \times 2 \times 3 \times 4 \times 5 = 120$ .

### Example 4.3—Calculating the Day of Year

The *day of year* is the number of days (including the current day) which have elapsed since the beginning of a given year. It is a number in the range 1 to 365 for ordinary years, and 1 to 366 for leap years. Write a MATLAB program that accepts a day, month, and year, and calculates the day of year corresponding to that date.

**SOLUTION** To determine the day of year, this program will need to sum up the number of days in each month preceding the current month, plus the number of elapsed days in the current month. A `for` loop will be used to perform this sum. Since the number of days in each month varies, it is necessary to determine the correct number of days to add for each month. A `switch` construct will be used to determine the proper number of days to add for each month.

During a leap year, an extra day must be added to the day of year for any month after February. This extra day accounts for the presence of February 29 in the leap year. Therefore, to perform the day of year calculation correctly, we must determine which years are leap years. In the Gregorian calendar, leap years are determined by the following rules:

1. Years evenly divisible by 400 are leap years.
2. Years evenly divisible by 100 but *not* by 400 are not leap years.
3. All years divisible by 4 but *not* by 100 are leap years.
4. All other years are not leap years.

We will use the `mod` (for modulo) function to determine whether or not a year is evenly divisible by a given number. If the result of the `mod` function is zero, then the year is evenly divisible.

A program to calculate the day of year is shown below. Note that the program sums up the number of days in each month before the current month and that it uses a `switch` construct to determine the number of days in each month.

```

% Script file: doy.m
%
% Purpose:
%   This program calculates the day of year corresponding
%   to a specified date. It illustrates the use switch and
%   for constructs.
%
% Record of revisions:
%   Date           Programmer           Description of change
%   ====          =====
%   01/07/04      S. J. Chapman           Original code
%
% Define variables:
%   day            -- Day (dd)
%   day_of_year    -- Day of year
%   ii             -- Loop index
%   leap_day       -- Extra day for leap year
%   month          -- Month (mm)
%   year           -- Year (yyyy)

% Get day, month, and year to convert
disp('This program calculates the day of year given the ');
disp('current date. ');
month = input('Enter current month (1-12): ');
day   = input('Enter current day(1-31): ');
year  = input('Enter current year(yyyy): ');

% Check for leap year, and add extra day if necessary
if mod(year,400) == 0
    leap_day = 1;    % Years divisible by 400 are leap years
elseif mod(year,100) == 0
    leap_day = 0;    % Other centuries are not leap years
elseif mod(year,4) == 0
    leap_day = 1;    % Otherwise every 4th year is a leap year
else
    leap_day = 0;    % Other years are not leap years
end

% Calculate day of year by adding current day to the
% days in previous months.
day_of_year = day;
for ii = 1:month-1

```

```

% Add days in months from January to last month
switch (ii)
case {1,3,5,7,8,10,12},
    day_of_year = day_of_year + 31;
case {4,6,9,11},
    day_of_year = day_of_year + 30;
case 2,
    day_of_year = day_of_year + 28 + leap_day;
end

end

% Tell user
fprintf('The date %2d/%2d/%4d is day of year %d.\n', ...
        month, day, year, day_of_year);

```

We will use the following known results to test the program:

1. Year 1999 is not a leap year. January 1 must be day of year 1, and December 31 must be day of year 365.
2. Year 2000 is a leap year. January 1 must be day of year 1, and December 31 must be day of year 366.
3. Year 2001 is not a leap year. March 1 must be day of year 60, since January has 31 days, February has 28 days, and this is the first day of March.

If this program is executed five times with the above dates, the results are

» **doy**

This program calculates the day of year given the current date.

```

Enter current month (1-12): 1
Enter current day(1-31): 1
Enter current year(yyyy): 1999
The date 1/ 1/1999 is day of year 1.

```

» **doy**

This program calculates the day of year given the current date.

```

Enter current month (1-12): 12
Enter current day(1-31): 31
Enter current year(yyyy): 1999
The date 12/31/1999 is day of year 365.

```

» **doy**

This program calculates the day of year given the current date.

```

Enter current month (1-12): 1
Enter current day(1-31): 1
Enter current year(yyyy): 2000
The date 1/ 1/2000 is day of year 1.

```

**» doy**

This program calculates the day of year given the current date.

Enter current month (1-12): 12

Enter current day(1-31): 31

Enter current year(yyyy): 2000

The date 12/31/2000 is day of year 366.

**» doy**

This program calculates the day of year given the current date.

Enter current month (1-12): 3

Enter current day(1-31): 1

Enter current year(yyyy): 2001

The date 3/ 1/2001 is day of year 60.

The program gives the correct answers for our test dates in all five test cases. ◀

#### Example 4.4—Statistical Analysis

Implement an algorithm that reads in a set of measurements and calculates the mean and the standard deviation of the input data set, when any value in the data set can be positive, negative, or zero.

**SOLUTION** This program must be able to read in an arbitrary number of measurements, and then calculate the mean and standard deviation of those measurements. Each measurement can be positive, negative, or zero.

Since we cannot use a data value as a flag this time, we will ask the user for the number of input values, and then use a for loop to read in those values. The modified program that permits the use of any input value is shown below. Verify its operation for yourself by finding the mean and standard deviation of the following five input values: 3., -1., 0., 1., and -2.

```
% Script file: stats_3.m
%
% Purpose:
%   To calculate mean and the standard deviation of
%   an input data set, where each input value can be
%   positive, negative, or zero.
%
% Record of revisions:
%
%   Date           Programmer           Description of change
%   ====          =====
%   01/08/04      S. J. Chapman           Original code
%
```

```

% Define variables:
%   ii      -- Loop index
%   n       -- The number of input samples
%   std_dev -- The standard deviation of the input samples
%   sum_x   -- The sum of the input values
%   sum_x2  -- The sum of the squares of the input values
%   x       -- An input data value
%   xbar    -- The average of the input samples

% Initialize sums.
sum_x = 0; sum_x2 = 0;

% Get the number of points to input.
n = input('Enter number of points: ');

% Check to see if we have enough input data.
if n < 2          % Insufficient data

    disp ('At least 2 values must be entered.');
```

else % we will have enough data, so let's get it.

```

    % Loop to read input values.
    for ii = 1:n

        % Read in next value
        x = input('Enter value: ');

        % Accumulate sums.
        sum_x = sum_x + x;
        sum_x2 = sum_x2 + x^2;

    end

    % Now calculate statistics.
    x_bar = sum_x / n;
    std_dev = sqrt( (n * sum_x2 - sum_x^2) / (n * (n-1)) );

    % Tell user.
    fprintf('The mean of this data set is:    %f\n', x_bar);
    fprintf('The standard deviation is:       %f\n', std_dev);
    fprintf('The number of data points is:      %f\n', n);

end
```



## Details of Operation

Now that we have seen examples of a `for` loop in operation, we must examine some important details required to use `for` loops properly.

1. **Indent the bodies of loops.** It is not necessary to indent the body of a `for` loop as we have shown above. MATLAB will recognize the loop even if every statement in it starts in column 1. However, the code is much more readable if the body of the `for` loop is indented, so you should always indent the bodies of loops.

### \* Good Programming Practice

Always indent the body of a `for` loop by two or more spaces to improve the readability of the code.

2. **Don't modify the loop index within the body of a loop.** The loop index of a `for` loop *should not be modified anywhere within the body of the loop*. The index variable is often used as a counter within the loop, and modifying its value can cause strange and hard-to-find errors. The example shown below is intended to initialize the elements of an array, but the statement "`ii = 5`" has been accidentally inserted into the body of the loop. As a result, only `a(5)` is initialized, and it gets the values that should have gone into `a(1)`, `a(2)`, etc.

```
for ii = 1:10
    ...
    ii = 5;    % Error!
    ...
    a(ii) = <calculation>
end
```

### \* Good Programming Practice

Never modify the value of a loop index within the body of the loop.

3. **Preallocating Arrays.** We learned in Chapter 2 that it is possible to extend an existing array simply by assigning a value to a higher array element. For example, the statement

```
arr = 1:4;
```

defines a 4-element array containing the values [1 2 3 4]. If the statement

```
arr(8) = 6;
```

is executed, the array will be automatically extended to eight elements, and will contain the values [1 2 3 4 0 0 0 6]. Unfortunately, each time that an array is extended, MATLAB has to (1) create a new array, (2) copy the contents of the old array to the new longer array, (3) add the new value to the array, and then (4) delete the old array. This process is very time-consuming for long arrays.

When a `for` loop stores values in a previously undefined array, the loop forces MATLAB to go through this process each time the loop is executed. On the other hand, if the array is **preallocated** to its maximum size before the loop starts executing, no copying is required, and the code executes much faster. The code fragment shown below shows how to pre-allocate an array before the starting the loop.

```
square = zeros(1,100);
for ii = 1:100
    square(ii) = ii^2;
end
```

#### \* Good Programming Practice

Always preallocate all arrays used in a loop before executing the loop. This practice greatly increases the execution speed of the loop.

4. **Vectorizing Arrays.** It is often possible to perform calculations with either `for` loops or vectors. For example, the following code fragment calculates the squares, square roots, and cube roots of all integers between 1 and 100 using a `for` loop.

```
for ii = 1:100
    square(ii) = ii^2;
    square_root(ii) = ii^(1/2);
    cube_root(ii) = ii^(1/3);
end
```

The following code fragment performs the same calculation with vectors.

```
ii = 1:100;
square = ii.^2;
square_root = ii.^(1/2);
cube_root(ii) = ii.^(1/3);
```

Even though these two calculations produce the same answers, they are *not* equivalent. The version with the `for` loop can be *more than 15 times slower* than the vectorized version! This happens because the statements in the `for` loop must be interpreted<sup>1</sup> and executed a line at a time by MATLAB during each pass of the loop. In effect, MATLAB must interpret and execute 300 separate lines of code. In contrast, MATLAB only has to interpret and execute 4 lines in the vectorized case. Since MATLAB is designed to implement vectorized statements in a very efficient fashion, it is much faster in that mode.

In MATLAB, the process of replacing loops by vectorized statements is known as **vectorization**. Vectorization can yield dramatic improvements in performance for many MATLAB programs.

### \* Good Programming Practice

If it is possible to implement a calculation either with a `for` loop or using vectors, implement the calculation with vectors. Your program will be much faster.

## The MATLAB Just-in-Time (JIT) Compiler

A just-in-time (JIT) compiler was added to MATLAB 6.5 and later versions. The JIT compiler examines MATLAB code before it is executed and, where possible, compiles the code before executing it. Since the MATLAB code is compiled instead of being interpreted, it runs almost as fast as a vectorized code. The JIT compiler can sometimes dramatically speed up the execution of `for` loops.

The JIT compiler is a very nice tool when it works, because it speeds up the loops without any action by the programmer. However, the JIT compiler has many limitations that prevent it from speeding up all loops. A full list of JIT compiler limitations appears in the MATLAB documentation, but some of the more important limitations are:

1. The JIT only accelerates loops containing `double`, `logical`, and `char` data types (plus integer data types that we haven't met yet). If other data types such as cell arrays or structures<sup>2</sup> appear in the loop, it will not be accelerated.
2. If an array in the loop has more than two dimensions, the loop will not be accelerated.
3. If the code in the loop calls external functions (other than built-in functions), it will not be accelerated.

<sup>1</sup>But see the next item about the MATLAB Just-In-Time compiler.

<sup>2</sup>We will learn about these data types in Chapter 7.

- If the code in the loop changes the data type of a variable within a loop, the loop will not be accelerated.

Because of these limitations, a good programmer using vectorization can almost always create a faster program than one relying on the JIT compiler.

### \* Good Programming Practice

Do not rely on the JIT compiler to speed up your code. It has many limitations, and a programmer can typically do a better job with manual vectorization.

#### Example 4.5—Comparing Loops and Vectors

To compare the execution speeds of loops and vectors, we will perform and time the following four sets of calculations.

- Calculate the squares of every integer from 1 to 10,000 in a `for` loop without initializing the array of squares first.
- Calculate the squares of every integer from 1 to 10,000 in a `for` loop, using the `zeros` function to preallocate the array of squares first, but calling an external function to perform the squaring. (This will disable the JIT compiler.)
- Calculate the squares of every integer from 1 to 10,000 in a `for` loop, using the `zeros` function to preallocate the array of squares first, and calculating the square of the number in-line. (This will allow the JIT compiler to function.)
- Calculate the squares of every integer from 1 to 10,000 with vectors.

**SOLUTION** This program must calculate the squares of the integers from 1 to 10,000 in each of the four ways described above, timing the executions in each case. The timing can be accomplished using the MATLAB functions `tic` and `toc`. Function `tic` resets the built-in elapsed time counter, and function `toc` returns the elapsed time in seconds since the last call to function `tic`.

Since the real-time clocks in many computers have a fairly coarse granularity, it may be necessary to execute each set of instructions multiple times to get a valid average time.

A MATLAB program to compare the speeds of the four approaches is shown below:

```
% Script file: timings.m
%
% Purpose:
% This program calculates the time required to
```

```

% calculate the squares of all integers from 1 to
% 10,000 in four different ways:
% 1. Using a for loop with an uninitialized output
% array.
% 2. Using a for loop with a pre-allocated output
% array and NO JIT compiler.
% 3. Using a for loop with a pre-allocated output
% array and the JIT compiler.
% 4. Using vectors.
%
% Record of revisions:
% Date Programmer Description of change
% ==== =====
% 01/09/04 S. J. Chapman Original code
%
% Define variables:
% ii, jj -- Loop index
% averagel -- Average time for calculation 1
% average2 -- Average time for calculation 2
% average3 -- Average time for calculation 3
% average4 -- Average time for calculation 4
% maxcount -- Number of times to loop calculation
% square -- Array of squares

% Perform calculation with an uninitialized array
% "square". This calculation is done only once
% because it is so slow.
maxcount = 1; % Number of repetitions
tic; % Start timer
for jj = 1:maxcount
    clear square % Clear output array
    for ii = 1:10000
        square(ii) = ii^2; % Calculate square
    end
end
averagel = (toc)/maxcount; % Calculate average time

% Perform calculation with a pre-allocated array
% "square", calling an external function to square
% the number. This calculation is averaged over 10
% loops.
maxcount = 10; % Number of repetitions
tic; % Start timer

```

```

for jj = 1:maxcount
    clear square % Clear output array
    square = zeros(1,10000); % Pre-initialize array
    for ii = 1:10000
        square(ii) = sqr(ii); % Calculate square
    end
end
average2 = (toc)/maxcount; % Calculate average time

% Perform calculation with a pre-allocated array
% "square". This calculation is averaged over 100
% loops.
maxcount = 100; % Number of repetitions
tic; % Start timer
for jj = 1:maxcount
    clear square % Clear output array
    square = zeros(1,10000); % Pre-initialize array
    for ii = 1:10000
        square(ii) = ii^2; % Calculate square
    end
end
average3 = (toc)/maxcount; % Calculate average time

% Perform calculation with vectors. This calculation
% averaged over 1000 executions.
maxcount = 1000; % Number of repetitions
tic; % Start timer
for jj = 1:maxcount
    clear square % Clear output array
    ii = 1:10000; % Set up vector
    square = ii.^2; % Calculate square
end
average4 = (toc)/maxcount; % Calculate average time

% Display results
fprintf('Loop / uninitialized array = %8.4f\n', average1);
fprintf('Loop / initialized array / no JIT = %8.4f\n', average2);
fprintf('Loop / initialized array / JIT = %8.4f\n', average3);
fprintf('Vectorized = %8.4f\n', average4);

```

When this program is executed using MATLAB 7.0 on a 2.4 GHz Pentium IV computer, the results are:

» **timings**

```

Loop / uninitialized array = 0.1100
Loop / initialized array / no JIT = 0.1797

```

```

Loop / initialized array / JIT      = 0.0005
Vectorized                          = 0.0001

```

The loop with the initialized array and the loop with the initialized array but no JIT were very slow compared with the loop executed with the JIT compiler or the vectorized loop. The vectorized loop was the fastest way to perform the calculation, but if the JIT compiler works for your loop, you get most of the acceleration without having to do anything! As you can see, designing loops to allow the JIT compiler to function<sup>3</sup> or replacing the loops with vectorized calculations can make an incredible difference in the speed of your MATLAB code!

## The break and continue Statements

There are two additional statements that can be used to control the operation of while loops and for loops: the `break` and `continue` statements. The `break` statement terminates the execution of a loop and passes control to the next statement after the end of the loop, while the `continue` statement terminates the current pass through the loop and returns control to the top of the loop.

If a `break` statement is executed in the body of a loop, the execution of the body will stop and control will be transferred to the first executable statement after the loop. An example of the `break` statement in a `for` loop is shown below.

```

for ii = 1:5
    if ii == 3;
        break;
    end
    fprintf('ii = %d\n',ii);
end
disp(['End of loop!']);

```

When this program is executed, the output is:

```

» test_break
ii = 1
ii = 2
End of loop!

```

Note that the `break` statement was executed on the iteration when `ii` was 3, and control transferred to the first executable statement after the loop without executing the `fprintf` statement.

<sup>3</sup>The MATLAB Profiler can help you speed up your code. This tool can identify loops that do not get speeded up by the JIT compiler, and tell you why they can't be accelerated. See the MATLAB documentation for details about the Profiler.

If a `continue` statement is executed in the body of a loop, the execution of the current pass through the loop will stop and control will return to the top of the loop. The controlling variable in the `for` loop will take on its next value, and the loop will be executed again. An example of the `continue` statement in a `for` loop is shown below.

```
for ii = 1:5
    if ii == 3;
        continue;
    end
    fprintf('ii = %d\n',ii);
end
disp(['End of loop!']);
```

When this program is executed, the output is:

```
>> test_continue
ii = 1
ii = 2
ii = 4
ii = 5
End of loop!
```

Note that the `continue` statement was executed on the iteration when `ii` was 3, and control transferred to the top of the loop without executing the `fprintf` statement.

The `break` and `continue` statements work with both `while` loops and `for` loops.

## Nesting Loops

It is possible for one loop to be completely inside another loop. If one loop is completely inside another one, the two loops are called **nested loops**. The following example shows two nested `for` loops used to calculate and write out the product of two integers.

```
for ii = 1:3
    for jj = 1:3
        product = ii * jj;
        fprintf('%d * %d = %d\n',ii,jj,product);
    end
end
```

In this example, the outer `for` loop will assign a value of 1 to index variable `ii`, and then the inner `for` loop will be executed. The inner `for` loop will be executed 3 times with index variable `jj` having values 1, 2, and 3. When the entire inner `for` loop has been completed, the outer `for` loop will assign a value of 2



to index variable `ii`, and the inner for loop will be executed again. This process repeats until the outer for loop has executed 3 times, and the resulting output is

```
1 * 1 = 1
1 * 2 = 2
1 * 3 = 3
2 * 1 = 2
2 * 2 = 4
2 * 3 = 6
3 * 1 = 3
3 * 2 = 6
3 * 3 = 9
```

Note that the inner for loop executes completely before the index variable of the outer for loop is incremented.

When MATLAB encounters an `end` statement, it associates that statement with the innermost currently open construct. Therefore, the first `end` statement above closes the “for `jj` = 1:3” loop, and the second `end` statement above closes the “for `ii` = 1:3” loop. This fact can produce hard-to-find errors if an `end` statement is accidentally deleted somewhere within a nested loop construct.

If for loops are nested, they should have independent loop index variables. If they have the same index variable, then the inner loop will change the value of the loop index that the outer loop just set.

If a `break` or `continue` statement appears inside a set of nested loops, then that statement refers to the innermost of the loops containing it. For example, consider the following program

```
for ii = 1:3
    for jj = 1:3
        if jj == 3;
            break;
        end
        product = ii * jj;
        fprintf('%d * %d = %d\n', ii, jj, product);
    end
    fprintf('End of inner loop\n');
end
fprintf('End of outer loop\n');
```

If the inner loop counter `jj` is equal to 3, then the `break` statement will be executed. This will cause the program to exit the innermost loop. The program will print out “End of inner loop”, the index of the outer loop will be increased by 1, and execution of the innermost loop will start over. The resulting output values are

```

1 * 1 = 1
1 * 2 = 2
End of inner loop
2 * 1 = 2
2 * 2 = 4
End of inner loop
3 * 1 = 3
3 * 2 = 6
End of inner loop
End of outer loop

```

### 4.3 Logical Arrays and Vectorization

We learned about the logical data type in Chapter 3. Logical data can have one of two possible values: true (1) or false (0). Scalars and arrays of logical data are created as the output of relational and logic operators.

For example, consider the following statements:

```

a = [1 2 3; 4 5 6; 7 8 9];
b = a > 5;

```

These statements produced two arrays *a* and *b*. Array *a* is a double array con-

taining the values  $\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$ , while array *b* is a logical array containing the

values  $\begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 1 \\ 1 & 1 & 1 \end{bmatrix}$ . When the `whos` command is executed, the results are as shown below.

```

» whos
Name      Size      Bytes      Class
a         3x3         72      double array
b         3x3          9      logical array
Grand total is 18 elements using 81 bytes

```

Logical arrays have a very important special property—they can serve as a *mask for arithmetic operations*. A mask is an array that selects the elements of another array for use in an operation. The specified operation will be applied to the selected elements, and *not* to the remaining elements.

For example, suppose that arrays *a* and *b* are as defined above. Then the statement `a(b) = sqrt(a(b))` will take the square root of all elements for which the logical array *b* is true, and leave all the other elements in the array unchanged.

```

» a(b) = sqrt(a(b))
a =
    1.0000    2.0000    3.0000
    4.0000    5.0000    2.4495
    2.6458    2.8284    3.0000

```

This is a very fast and very clever way of performing an operation on a subset of an array without needing loops and branches.

The following two code fragments both take the square root of all elements in array *a* whose value is greater than 5, but the vectorized approach is much faster than the loop approach.

```

for ii = 1:size(a,1)
    for jj = 1:size(a,2)
        if a(ii,jj) > 5
            a(ii,jj) = sqrt(a(ii,jj));
        end
    end
end

b = a > 5;
a(b) = sqrt(a(b));

```

#### Example 4.6—Using Logical Arrays to Mask Operations

To compare the execution speeds of loops and branches versus vectorized code using a logical array, we will perform and time the following two sets of calculations.

1. Create a 10,000-element array containing the values, 1, 2, . . . , 10,000. Then take the square root of all elements whose value is greater than 5000 using a *for* loop and an *if* construct.
2. Create a 10,000-element array containing the values, 1, 2, . . . , 10,000. Then take the square root of all elements whose value is greater than 5000 using a logical array.

**SOLUTION** This program must create an array containing the integers from 1 to 10000, and take the square roots of those value that are greater than 5000 in each of the two ways described above.

A MATLAB program to compare the speeds of the two approaches is shown below:

```

% Script file: logical1.m
%
% Purpose:
% This program calculates the time required to
% calculate the square roots of all elements in

```

```

%   array a whose value exceeds 5000. This is done
%   in two different ways:
%   1. Using a for loop and if construct.
%   2. Using a logical array.
%
% Record of revisions:
%   Date           Programmer           Description of change
%   ====          =====          =====
%   01/10/04      S. J. Chapman           Original code
%
% Define variables:
%   a              -- Array of input values
%   b              -- Logical array to serve as a mask
%   ii, jj         -- Loop index
%   average1       -- Average time for calculation 1
%   average2       -- Average time for calculation 2
%   maxcount       -- Number of times to loop calculation
%   month          -- Month (mm)
%   year           -- Year (yyyy)

% Perform calculation using loops and branches.
maxcount = 1;           % One repetition
tic;                   % Start timer
for jj = 1:maxcount
    a = 1:10000;        % Declare array a
    for ii = 1:10000
        if a(ii) > 5000
            a(ii) = sqrt(a(ii));
        end
    end
end
average1 = (toc)/maxcount; % Calculate average time

% Perform calculation using logical arrays.
maxcount = 10;         % One repetition
tic;                   % Start timer
for jj = 1:maxcount
    a = 1:10000;        % Declare array a
    b = a > 5000;       % Create mask
    a(b) = sqrt(a(b)); % Take square root
end
average2 = (toc)/maxcount; % Calculate average time

```

```
% Display results
fprintf('Loop / if approach = %8.4f\n', ...
        average1);
fprintf('Logical array approach = %8.4f\n', ...
        average2);
```

When this program is executed using MATLAB 7.0 on a 2.4 GHz Pentium IV computer, the results are:

```
> logical1
Loop / if approach =      0.1200
Logical array approach =  0.0060
```

As you can see, the use of logical arrays can speed up code execution by a factor of 20!

### ★ Good Programming Practice

Where possible, use logical arrays as masks to select the elements of an array for processing. If logical arrays are used instead of loops and `if` constructs, your program will be much faster.

## Creating the Equivalent of `if/else` Constructs with Logical Arrays

Logical arrays can also be used to implement the equivalent of an `if/else` construct inside a set of `for` loops. As we saw in the preceding section, it is possible to apply an operation to selected elements of an array using a logical array as a mask. It is also possible to apply a different set of operations to the *unselected* elements of the array by simply adding the not operator (`~`) to the logical mask. For example, suppose that we wanted to take the square root of any elements in a two-dimensional array whose value is greater than 5 and to square the remaining elements in the array. The code for this operation using loops and branches is

```
for ii = 1:size(a,1)
    for jj = 1:size(a,2)
        if a(ii,jj) > 5
            a(ii,jj) = sqrt(a(ii,jj));
        else
            a(ii,jj) = a(ii,jj)^2;
        end
    end
end
```

The vectorized code for this operation is

```
b = a > 5;
a(b) = sqrt(a(b));
a(~b) = a(~b).^2;
```

The vectorized code is enormously faster than the loops-and-branches version.

### Quiz 4.1

This quiz provides a quick check to see if you have understood the concepts introduced in Sections 4.1 through 4.3. If you have trouble with the quiz, reread the section, ask your instructor, or discuss the material with a fellow student. The answers to this quiz are found in the back of the book.

Examine the following for loops and determine how many times each loop will be executed.

1. for index = 7:10
2. for jj = 7:-1:10
3. for index = 1:10:10
4. for ii = -10:3:-7
5. for kk = [0 5 ; 3 3]

Examine the following loops and determine the value in ires at the end of each of the loops.

6.
 

```
ires = 0;
for index = 1:10
    ires = ires + 1;
end
```
7.
 

```
ires = 0;
for index = 1:10
    ires = ires + index;
end
```
8.
 

```
ires = 0;
for index1 = 1:10
    for index2 = index1:10
        if index2 == 6
            break;
        end
        ires = ires + 1;
    end
end
```

```

9.      ires = 0;
        for index1 = 1:10
            for index2 = index1:10
                if index2 == 6
                    continue;
                end
                ires = ires + 1;
            end
        end
    end
end

```

10. Write the MATLAB statements to calculate the values of the function

$$f(t) = \begin{cases} \sin t & \text{for all } t \text{ where } \sin t > 0 \\ 0 & \text{elsewhere} \end{cases}$$

for  $-6\pi \leq t \leq 6\pi$  at intervals of  $\pi/10$ . Do this twice, once using loops and branches, and once using vectorized code.

## 4.4 Additional Examples

### Example 4.7—Fitting a Line to a Set of Noisy Measurements

The velocity of a falling object in the presence of a constant gravitational field is given by the equation

$$v(t) = at + v_0 \quad (4-3)$$

where  $v(t)$  is the velocity at any time  $t$ ,  $a$  is the acceleration due to gravity, and  $v_0$  is the velocity at time 0. This equation is derived from elementary physics—it is known to every freshman physics student. If we plot velocity versus time for the falling object, our  $(v, t)$  measurement points should fall along a straight line. However, the same freshman physics student also knows that if we go out into the laboratory and attempt to *measure* the velocity versus time of an object, our measurements will *not* fall along a straight line. They may come close, but they will never line up perfectly. Why not? Because we can never make perfect measurements. There is always some *noise* included in the measurements, which distorts them.

There are many cases in science and engineering where there are noisy sets of data such as this, and we wish to estimate the straight line which “best fits” the data. This problem is called the *linear regression* problem. Given a noisy set of measurements  $(x, y)$  that appear to fall along a straight line, how can we find the equation of the line

$$y = mx + b \quad (4-4)$$

that “best fits” the measurements? If we can determine the regression coefficients  $m$  and  $b$ , then we can use this equation to predict the value of  $y$  at any given  $x$  by evaluating Equation (4-4) for that value of  $x$ .

A standard method for finding the regression coefficients  $m$  and  $b$  is the *method of least squares*. This method is named “least squares” because it produces the line  $y = mx + b$  for which the sum of the squares of the differences between the observed  $y$  values and the predicted  $y$  values is as small as possible. The slope of the least squares line is given by

$$m = \frac{(\sum xy) - (\sum x)\bar{y}}{(\sum x^2) - (\sum x)\bar{x}} \quad (4-5)$$

and the intercept of the least squares line is given by

$$b = \bar{y} - m\bar{x} \quad (4-6)$$

where

$\sum x$  is the sum of the  $x$  values

$\sum x^2$  is the sum of the squares of the  $x$  values

$\sum xy$  is the sum of the products of the corresponding  $x$  and  $y$  values

$\bar{x}$  is the mean (average) of the  $x$  values

$\bar{y}$  is the mean (average) of the  $y$  values

Write a program that will calculate the least-squares slope  $m$  and  $y$ -axis intercept  $b$  for a given set of noisy measured data points  $(x, y)$ . The data points should be read from the keyboard, and both the individual data points and the resulting least-squares fitted line should be plotted.

#### SOLUTION

##### 1. State the problem.

Calculate the slope  $m$  and intercept  $b$  of a least-squares line that best fits an input data set consisting of an arbitrary number of  $(x, y)$  pairs. The input  $(x, y)$  data is read from the keyboard. Plot both the input data points and the fitted line on a single plot.

##### 2. Define the inputs and outputs.

The inputs required by this program are the number of points to read, plus the pairs of points  $(x, y)$ .

The outputs from this program are the slope and intercept of the least-squares fitted line, the number of points going into the fit, and a plot of the input data and the fitted line.

##### 3. Describe the algorithm.

This program can be broken down into six major steps

Get the number of input data points

Read the input statistics

Calculate the required statistics



Calculate the slope and intercept  
 Write out the slope and intercept  
 Plot the input points and the fitted line

The first major step of the program is to get the number of points to read in. To do this, we will prompt the user and read his or her answer with an input function. Next we will read the input (x, y) pairs one pair at a time using an input function in a for loop. Each pair of input value will be placed in an array ([x y]), and the function will return that array to the calling program. Note that a for loop is appropriate because we know in advance how many times the loop will be executed.

The pseudocode for these steps is shown below below.

```
Print message describing purpose of the program
n_points <- input('Enter number of [x y] pairs: ');
for ii = 1:n_points
    temp <- input('Enter [x y] pair: ');
    x(ii) <- temp(1)
    y(ii) <- temp(2)
end
```

Next, we must accumulate the statistics required for the calculation. These statistics are the sums  $\sum x$ ,  $\sum y$ ,  $\sum x^2$ , and  $\sum xy$ . The pseudocode for these steps is:

```
Clear the variables sum_x, sum_y, sum_x2, and sum_xy
for ii = 1:n_points
    sum_x <- sum_x + x(ii)
    sum_y <- sum_y + y(ii)
    sum_x2 <- sum_x2 + x(ii)^2
    sum_xy <- sum_xy + x(ii)*y(ii)
end
```

Next, we must calculate the slope and intercept of the least-squares line. The pseudocode for this step is just the MATLAB versions of Equations (4-4) and (4-5).

```
x_bar <- sum_x / n_points
y_bar <- sum_y / n_points
slope <- (sum_xy - sum_x * y_bar) / (sum_x2 - sum_x * x_bar)
y_int <- y_bar - slope * x_bar
```

Finally, we must write out and plot the results. The input data points should be plotted with circular markers and without a connecting line, while the fitted line should be plotted as a solid 2-pixel-wide line. To do this, we will need to plot the points first, set hold on, plot the fitted line, and set hold off. We will add titles and a legend to the plot for completeness.

## 4. Turn the algorithm into MATLAB statements.

The final MATLAB program is shown below:

```
%
% Purpose:
% To perform a least-squares fit of an input data set
% to a straight line, and print out the resulting slope
% and intercept values. The input data for this fit
% comes from a user-specified input data file.
%
% Record of revisions:
%      Date      Programmer      Description of change
%      ----      -
%      01/10/04  S. J. Chapman      Original code
%
% Define variables:
%   ii          -- Loop index
%   n_points    -- Number in input [x y] points
%   slope       -- Slope of the line
%   sum_x       -- Sum of all input x values
%   sum_x2      -- Sum of all input x values squared
%   sum_xy      -- Sum of all input x*y values
%   sum_y       -- Sum of all input y values
%   temp        -- Variable to read user input
%   x           -- Array of x values
%   x_bar       -- Average x value
%   y           -- Array of y values
%   y_bar       -- Average y value
%   y_int       -- y-axis intercept of the line

disp('This program performs a least-squares fit of an ');
disp('input data set to a straight line. ');
n_points = input('Enter the number of input [x y] points: ');

% Read the input data
for ii = 1:n_points
    temp = input('Enter [x y] pair: ');
    x(ii) = temp(1);
    y(ii) = temp(2);
end

% Accumulate statistics
sum_x = 0;
sum_y = 0;
sum_x2 = 0;
sum_xy = 0;
```

```

for ii = 1:n_points
    sum_x = sum_x + x(ii);
    sum_y = sum_y + y(ii);
    sum_x2 = sum_x2 + x(ii)^2;
    sum_xy = sum_xy + x(ii) * y(ii);
end

% Now calculate the slope and intercept.
x_bar = sum_x / n_points;
y_bar = sum_y / n_points;
slope = (sum_xy - sum_x * y_bar) / (sum_x2 - sum_x * x_bar);
y_int = y_bar - slope * x_bar;

% Tell user.
disp('Regression coefficients for the least-squares line:');
fprintf(' Slope (m) = %8.3f\n', slope);
fprintf(' Intercept (b) = %8.3f\n', y_int);
fprintf(' No of points = %8d\n', n_points);

% Plot the data points as blue circles with no
% connecting lines.
plot(x,y,'bo');
hold on;

% Create the fitted line
xmin = min(x);
xmax = max(x);
ymin = slope * xmin + y_int;
ymax = slope * xmax + y_int;

% Plot a solid red line with no markers
plot([xmin xmax],[ymin ymax],'r-','LineWidth',2);
hold off;

% Add a title and legend
title ('\bfLeast-Squares Fit');
xlabel ('\bf\itx');
ylabel ('\bf\ity');
legend('Input data','Fitted line');
grid on

```

### 5. Test the program.

To test this program, we will try a simple data set. For example, if every point in the input data set actually falls along a line, then the resulting slope and intercept should be exactly the slope and intercept of that line.

Thus the data set

```
[1.1 1.1]
[2.2 2.2]
[3.3 3.3]
[4.4 4.4]
[5.5 5.5]
[6.6 6.6]
[7.7 7.7]
```

should produce a slope of 1.0 and an intercept of 0.0. If we run the program with these values, the results are:

» **lsqfit**

This program performs a least-squares fit of an input data set to a straight line.

Enter the number of input [x y] points: 7

Enter [x y] pair: [1.1 1.1]

Enter [x y] pair: [2.2 2.2]

Enter [x y] pair: [3.3 3.3]

Enter [x y] pair: [4.4 4.4]

Enter [x y] pair: [5.5 5.5]

Enter [x y] pair: [6.6 6.6]

Enter [x y] pair: [7.7 7.7]

Regression coefficients for the least-squares line:

Slope (m) = 1.000

Intercept (b) = 0.000

No of points = 7

Now let's add some noise to the measurements. The data set becomes

```
[1.1 1.01]
[2.2 2.30]
[3.3 3.05]
[4.4 4.28]
[5.5 5.75]
[6.6 6.48]
[7.7 7.84]
```

If we run the program with these values, the results are:

» **lsqfit**

This program performs a least-squares fit of an input data set to a straight line.

Enter the number of input [x y] points: 7

Enter [x y] pair: [1.1 1.01]

Enter [x y] pair: [2.2 2.30]

Enter [x y] pair: [3.3 3.05]

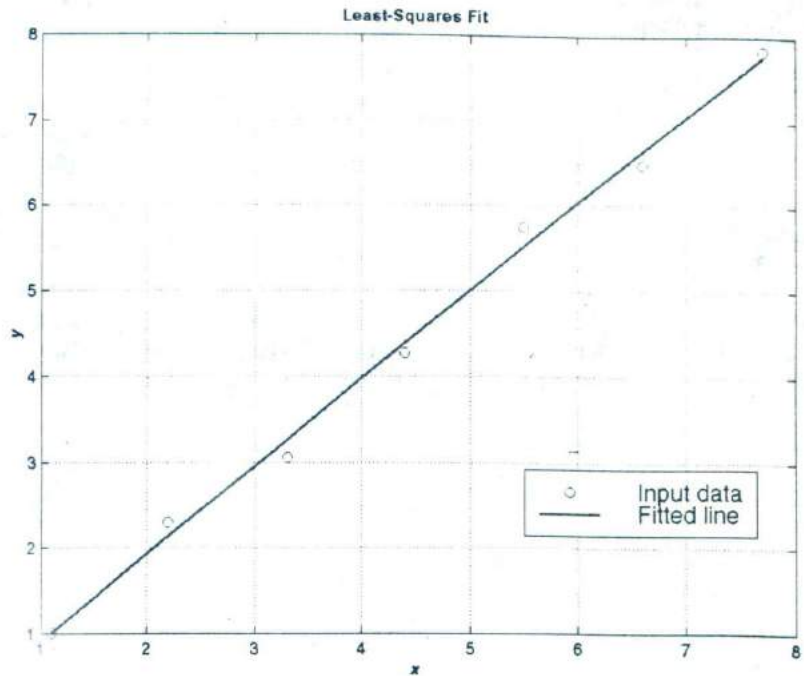


Figure 4.1 A noisy data set with a least-squares fitted line.

```

Enter [x y] pair: [4.4 4.28]
Enter [x y] pair: [5.5 5.75]
Enter [x y] pair: [6.6 6.48]
Enter [x y] pair: [7.7 7.84]
Regression coefficients for the least-squares line:
Slope (m) = 1.024
Intercept (b) = -0.120
No. of points = 7

```

If we calculate the answer by hand, it is easy to show that the program gives the correct answers for our two test data sets. The noisy input data set and the resulting least-squares fitted line are shown in Figure 4.1. ◀

This example uses several of the plotting capabilities that we introduced in Chapter 3. It uses the `hold` command to allow multiple plots to be placed on the same figure. It uses the `LineWidth` property to set the width of the least-squares fitted line, and the `FontWeight` and `FontStyle` properties to make the title bold face and the axis labels bold italic.

**Example 4.8—Physics—The Flight of a Ball**

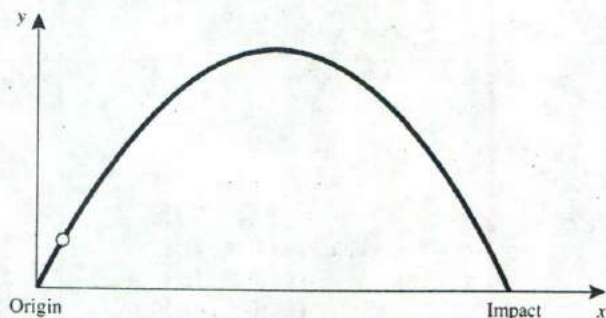
If we assume negligible air friction and ignore the curvature of the Earth, a ball that is thrown into the air from any point on the Earth's surface will follow a parabolic flight path (see Figure 4.2a). The height of the ball at any time  $t$  after it is thrown is given by Equation (4-7)

$$y(t) = y_0 + v_{y0}t + \frac{1}{2}gt^2 \quad (4-7)$$

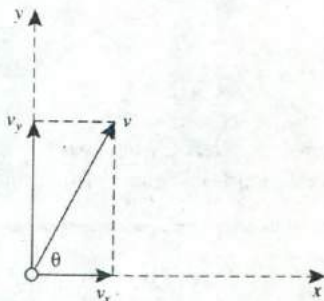
where  $y_0$  is the initial height of the object above the ground,  $v_{y0}$  is the initial vertical velocity of the object, and  $g$  is the acceleration due to the Earth's gravity. The horizontal distance (range) traveled by the ball as a function of time after it is thrown is given by Equation (4-8)

$$x(t) = x_0 + v_{x0}t \quad (4-8)$$

where  $x_0$  is the initial horizontal position of the ball on the ground, and  $v_{x0}$  is the initial horizontal velocity of the ball.



(a)



(b)

**Figure 4.2** (a) When a ball is thrown upwards, it follows a parabolic trajectory. (b) The horizontal and vertical components of a velocity vector  $v$  at an angle  $\theta$  with respect to the horizontal.

If the ball is thrown with some initial velocity  $v_0$  at an angle of  $\theta$  degrees with respect to the Earth's surface, then the initial horizontal and vertical components of velocity will be

$$v_{x0} = v_0 \cos \theta \quad (4-9)$$

$$v_{y0} = v_0 \sin \theta \quad (4-10)$$

Assume that the ball is initially thrown from position  $(x_0, y_0) = (0, 0)$  with an initial velocity  $v_0$  of 20 meters per second at an initial angle of  $\theta$  degrees. Write a program that will plot the trajectory of the ball and also determine the horizontal distance traveled before it touches the ground again. The program should plot the trajectories of the ball for all angles  $\theta$  from  $5^\circ$  to  $85^\circ$  in  $10^\circ$  steps and should determine the horizontal distance traveled for all angles  $\theta$  from  $0^\circ$  to  $90^\circ$  in  $1^\circ$  steps. Finally, it should determine the angle  $\theta$  that maximizes the range of the ball and plot that particular trajectory in a different color with a thicker line.

**SOLUTION** To solve this problem, we must determine an equation for the time that the ball returns to the ground. Then, we can calculate the  $(x, y)$  position of the ball using Equations (4-7) through (4-10). If we do this for many times between  $0$  and the time that the ball returns to the ground, we can use those points to plot the ball's trajectory.

The time  $t$  the ball will remain in the air after it is thrown may be calculated from Equation (4-7). The ball will touch the ground at the time  $t$  for which  $y(t) = 0$ . Remembering that the ball will start from ground level ( $y(0) = 0$ ), and solving for  $t$ , we get:

$$y(t) = y_0 + v_{y0}t + \frac{1}{2}gt^2 \quad (4-7)$$

$$0 = 0 + v_{y0}t + \frac{1}{2}gt^2$$

$$0 = \left( v_{y0} + \frac{1}{2}gt \right) t$$

so the ball will be at ground level at time  $t_1 = 0$  (when we threw it), and at time

$$t_2 = -\frac{2v_{y0}}{g} \quad (4-11)$$

From the problem statement, we know that the initial velocity  $v_0$  is 20 meters per second, and that the ball will be thrown at all angles from  $0^\circ$  to  $90^\circ$  in  $1^\circ$  steps. Finally, any elementary physics textbook will tell us that the acceleration due to the earth's gravity is  $-9.81$  meters per second squared.

Now let's apply our design technique to this problem.

#### 1. State the problem.

A proper statement of this problem would be: *Calculate the range that a ball would travel when it is thrown with an initial velocity of  $v_0$  of 20 m/s at an initial angle  $\theta$ . Calculate this range for all angles between  $0^\circ$  and  $90^\circ$  in  $1^\circ$  steps. Determine the angle  $\theta$  that will result in the maximum*

range for the ball. Plot the trajectory of the ball for angles between  $5^\circ$  and  $85^\circ$  in  $10^\circ$  increments. Plot the maximum-range trajectory in a different color and with a thicker line. Assume that there is no air friction.

## 2. Define the inputs and outputs.

As the problem is defined above, no inputs are required. We know from the problem statement what  $v_0$  and  $\theta$  will be, so there is no need to input them. The outputs from this program will be a table showing the range of the ball for each angle  $\theta$ , the angle  $\theta$  for which the range is maximum, and a plot of the specified trajectories.

## 3. Design the algorithm.

This program can be broken down into the following major steps

```
Calculate the range of the ball for  $\theta$  between 0 and  $90^\circ$ 
Write a table of ranges
Determine the maximum range and write it out
Plot the trajectories for  $\theta$  between 5 and  $85^\circ$ 
Plot the maximum-range trajectory
```

Since we know the exact number of times the loops will be repeated, for loops are appropriate for this algorithm. We will now refine the pseudocode for each of the major steps above.

To calculate the maximum range of the ball for each angle, we will first calculate the initial horizontal and vertical velocity from Equations (4-9) and (4-10). Then we will determine the time when the ball returns to Earth from Equation (4-11). Finally, we will calculate the range at that time from Equation (4-7). The detailed pseudocode for these steps is shown below. Note that we must convert all angles to radians before using the trig functions!

```
Create and initialize an array to hold ranges
for ii = 1:91
    theta <- ii * pi / 180
    vx0 <- v0 * cos(theta)
    vyo <- v0 * sin(theta)
    max_time <- -2 * vyo / g
    range(ii) <- vx0 * max_time
end
```

Next, we must write a table of ranges. The pseudocode for this step is:

```
Write heading
for ii = 1:91
    theta <- ii * pi / 180
    print theta and range(ii)
end
```

The maximum range can be found with the `max` function. Recall that this function returns both the maximum value and its location. The pseudocode for this step is:



```
[maxrange index] <- max(range)
Print out maximum range and angle (=index-1)
```

We will use nested for loops to calculate and plot the trajectories. To get all of the plots to appear on the screen, we must plot the first trajectory and then set `hold on` before plotting any other trajectories. After plotting the last trajectory, we must set `hold off`. To perform this calculation, we will divide each trajectory into 21 time steps, and find the  $x$  and  $y$  positions of the ball for each time step. Then, we will plot those  $(x, y)$  positions. The pseudocode for this step is:

```
for ii = 5:10:85

    % Get velocities and max time for this angle
    theta <- ii - 1
    vx0 <- vo * cos(theta*conv)
    vyo <- vo * sin(theta*conv)
    max_time <- -2 * vyo / g

    Initialize x and y arrays
    for jj = 1:21
        time <- (jj-1) * max_time/20
        x(time) <- vx0 * time
        y(time) <- vyo * time + 0.5 * g * time^2
    end
    plot(x,y) with thin green lines
    Set "hold on" after first plot
end
Add titles and axis labels
```

Finally, we must plot the maximum range trajectory in a different color and with a thicker line.

```
vx0 <- vo * cos(max_angle*conv)
vyo <- vo * sin(max_angle*conv)
max_time <- -2 * vyo / g

Initialize x and y arrays
for jj = 1:21
    time <- (jj-1) * max_time/20
    x(jj) <- vx0 * time
    y(jj) <- vyo * time + 0.5 * g * time^2
end
plot(x,y) with a thick red line
hold off
```

4. Turn the algorithm into MATLAB statements.  
The final MATLAB program is shown below.

```

% Script file: ball.m
%
% Purpose:
% This program calculates the distance traveled by a
% ball thrown at a specified angle "theta" and a
% specified velocity "vo" from a point on the surface of
% the Earth, ignoring air friction and the Earth's
% curvature. It calculates the angle yielding maximum
% range, and also plots selected trajectories.
%
% Record of revisions:
%      Date           Programmer           Description of change
%      ====          =====
%      01/10/04      S. J. Chapman           Original code
%
% Define variables:
% conv              -- Degrees to radians conv factor
% gravity           -- Accel. due to gravity (m/s^2)
% ii, jj           -- Loop index
% index            -- Location of maximum range in array
% maxangle         -- Angle that gives maximum range (deg)
% maxrange         -- Maximum range (m)
% range           -- Range for a particular angle (m)
% time            -- Time (s)
% theta           -- Initial angle (deg)
% traj_time       -- Total trajectory time (s)
% vo              -- Initial velocity (m/s)
% vxo            -- X-component of initial velocity (m/s)
% vyo            -- Y-component of initial velocity (m/s)
% x              -- X-position of ball (m)
% y              -- Y-position of ball (m)

% Constants
conv = pi / 180; % Degrees-to-radians conversion factor
g = -9.81;      % Accel. due to gravity
vo = 20;       % Initial velocity

% Create an array to hold ranges
range = zeros(1,91);

% Calculate maximum ranges
for ii = 1:91
    theta = ii - 1;
    vxo = vo * cos(theta*conv);
    vyo = vo * sin(theta*conv);

```

```

    max_time = -2 * vyo / g;
    range(ii) = vx0 * max_time;
end

% Write out table of ranges
fprintf ('Range versus angle theta:\n');
for ii = 1:91
    theta = ii - 1;
    fprintf(' %2d %8.4f\n',theta, range(ii));
end

% Calculate the maximum range and angle
[maxrange index] = max(range);
maxangle = index - 1;
fprintf ('\nMax range is %8.4f at %2d degrees.\n',...
        maxrange, maxangle);

% Now plot the trajectories
for ii = 5:10:85

    % Get velocities and max time for this angle
    theta = ii;
    vx0 = vo * cos(theta*conv);
    vyo = vo * sin(theta*conv);
    max_time = -2 * vyo / g;
    % Calculate the (x,y) positions
    x = zeros(1,21);
    y = zeros(1,21);
    for jj = 1:21
        time = (jj-1) * max_time/20;
        x(jj) = vx0 * time;
        y(jj) = vyo * time + 0.5 * g * time^2;
    end
    plot(x,y,'b');
    if ii == 5
        hold on;
    end
end

% Add titles and axis labels
title ('\bfTrajectory of Ball vs Initial Angle \theta');
xlabel ('\bf\itx \rm\bf(meters)');
ylabel ('\bf\ity \rm\bf(meters)');
axis ([0 45 0 25]);
grid on;

```

```

% Now plot the max range trajectory
vx0 = vo * cos(maxangle*conv);
vyo = vo * sin(maxangle*conv);
max_time = -2 * \vyo / g;

% Calculate the (x,y) positions
x = zeros(1,21);
y = zeros(1,21);
for jj = 1:21
    time = (jj-1) * max_time/20;
    x(jj) = vx0 * time;
    y(jj) = vyo * time + 0.5 * g * time^2;
end
plot(x,y,'r','Linewidth',3.0);
hold off

```

The acceleration due to gravity at sea level can be found in any physics text. It is about  $9.81 \text{ m/sec}^2$ , directed downward.

#### 5. Test the program.

To test this program, we will calculate the answers by hand for a few of the angles, and compare the results with the output of the program.

$\theta$	$v_{x0} = v_0 \cos \theta$	$v_{y0} = v_0 \sin \theta$	$t_2 = -\frac{2v_{y0}}{g}$	$x = v_{x0}t_2$
$0^\circ$	20 m/s	0 m/s	0 s	0 m
$5^\circ$	19.92 m/s	1.74 m/s	0.355 s	7.08 m
$40^\circ$	15.32 m/s	12.86 m/s	2.621 s	40.15 m
$45^\circ$	14.14 m/s	14.14 m/s	2.883 s	40.77 m

When program ball is executed, a 91-line table of angles and ranges is produced. To save space, only a portion of the table is reproduced below.

```

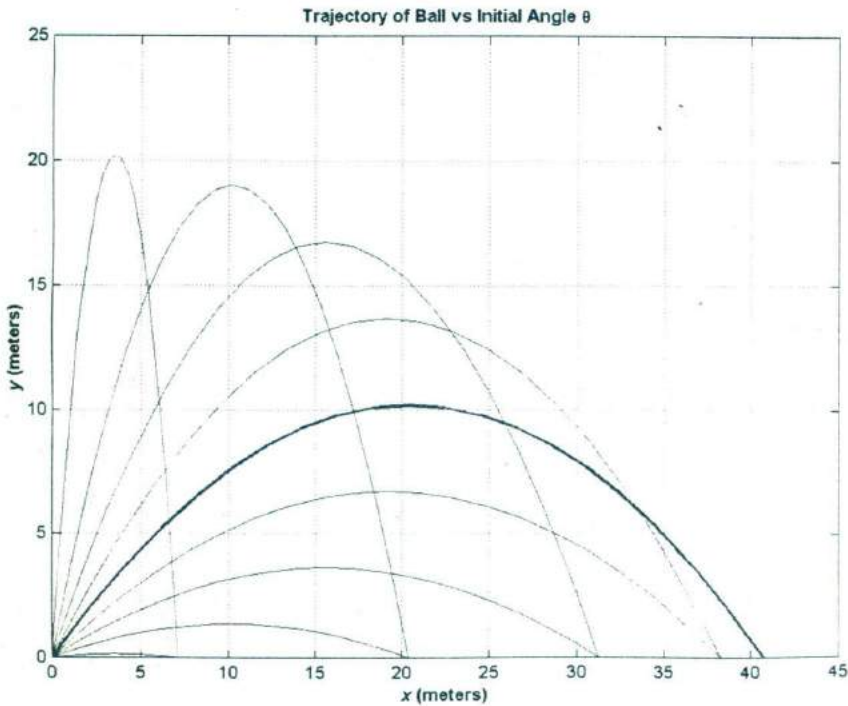
>> ball
Range versus angle theta:
0      0.0000
1      1.4230
2      2.8443
3      4.2621
4      5.6747
5      7.0805
...
40     40.1553
41     40.3779
42     40.5514

```

43	40.6754
44	40.7499
45	40.7747
46	40.7499
47	40.6754
48	40.5514
49	40.3779
50	40.1553
...	...
85	7.0805
86	5.6747
87	4.2621
88	2.8443
89	1.4230
90	0.0000

Max range is 40.7747 at 45 degrees.

The resulting plot is shown in Figure 4.3. The program output matches our hand calculation for the angles calculated above to the 4-digit accuracy of



**Figure 4.3** Possible trajectories for the ball.

the hand calculation. Note that the maximum range occurred at an angle of  $45^\circ$ .

This example uses several of the plotting capabilities that we introduced in Chapter 3. It uses the `axis` command to set the range of data to display, the `hold` command to allow multiple plots to be placed on the same axes, the `LineWidth` property to set the width of the line corresponding to the maximum-range trajectory, and escape sequences to create the desired title and  $x$ - and  $y$ -axis labels.

However, this program is not written in the most efficient manner, since there are a number of loops that could have been better replaced by vectorized statements. You will be asked to rewrite and improve `ball.m` in Exercise 4.11 at the end of this chapter.

## 4.5 Summary

There are two basic types of loops in MATLAB, the `while` loop and the `for` loop. The `while` loop is used to repeat a section of code in cases where we do not know in advance how many times the loop must be repeated. The `for` loop is used to repeat a section of code in cases where we know in advance how many times the loop should be repeated. It is possible to exit from either type of loop at any time using the `break` statement.

### Summary of Good Programming Practice

The following guidelines should be adhered to when programming with loop constructs. By following them consistently, your code will contain fewer bugs, will be easier to debug, and will be more understandable to others who may need to work with it in the future.

1. Always indent code blocks in `while` and `for` constructs to make them more readable.
2. Use a `while` loop to repeat sections of code when you don't know in advance how often the loop will be executed.
3. Use a `for` loop to repeat sections of code when you know in advance how often the loop will be executed.
4. Never modify the values of a `for` loop index while inside the loop.
5. Always preallocate all arrays used in a loop before executing the loop. This practice greatly increases the execution speed of the loop.
6. If it is possible to implement a calculation either with a `for` loop or using vectors, implement the calculation with vectors. Your program will be much faster.

7. Do not rely on the JIT compiler to speed up your code. It has many limitations, and a programmer can typically do a better job with manual vectorization.
8. Where possible, use logical arrays as masks to select the elements of an array for processing. If logical arrays are used instead of loops and `if` constructs, your program will be much faster.

## MATLAB Summary

The following summary lists all of the MATLAB commands and functions described in this chapter, along with a brief description of each one.

### Commands and Functions

---

<code>break</code>	Stop the execution of a loop, and transfer control to the first statement after the end of the loop.
<code>Continue</code>	Stop the execution of a loop, and transfer control to the top of the loop for the next iteration.
<code>for</code> loop	Loops over a block of statements a specified number of times.
<code>tic</code>	Resets elapsed time counter.
<code>toc</code>	Returns elapsed time since last call to <code>tic</code> .
<code>while</code> loop	Loops over a block of statements until a test condition becomes 0 (false).

---

## 4.6 Exercises

- 4.1 Write the MATLAB statements required to calculate  $y(t)$  from the equation

$$y(t) = \begin{cases} -3t^2 + 5 & t \geq 0 \\ 3t^2 + 5 & t < 0 \end{cases}$$

for values of  $t$  between  $-9$  and  $9$  in steps of  $0.5$ . Use loops and branches to perform this calculation.

- 4.2 Rewrite the statements required to solve Exercise 4.1 using vectorization.
- 4.3 Write the MATLAB statements required to calculate and print out the squares of all the even integers between 0 and 50. Create a table consisting of each integer and its square, with appropriate labels over each column.
- 4.4 Write an M-file to evaluate the equation  $y(x) = x^2 - 3x + 2$  for all values of  $x$  between  $-1$  and  $3$ , in steps of  $0.1$ . Do this twice, once with a `for` loop and once with vectors. Plot the resulting function using a 3-point-thick dashed red line.
- 4.5 Write an M-file to calculate the factorial function  $N!$ , as defined in Example 4.2. Be sure to handle the special case of  $0!$ . Also, be sure to report an error if  $N$  is negative or not an integer.

- 4.6** Examine the following for statements and determine how many times each loop will be executed.

```
(a) for ii = -32768:32767
(b) for ii = 32768:32767
(c) for kk = 2:4:3
(d) for jj = ones(5,5)
```

- 4.7** Examine the following for loops and determine the value of `ires` at the end of each of the loops, and also the number of times each loop executes.

```
(a) ires = 0;
    for index = -10:10
        ires = ires + 1;
    end

(b) ires = 0;
    for index = 10:-2:4
        if index == 0
            continue;
        end
        ires = ires + index;
    end

(c) ires = 0;
    for index = 10:-2:4
        if index == 0
            break;
        end
        ires = ires + index;
    end

(d) ires = 0;
    for index1 = 10:-2:4
        for index2 = 2:2:index1
            if index2 == 6
                break
            end
            ires = ires + index2;
        end
    end
```

- 4.8** Examine the following while loops and determine the value of `ires` at the end of each of the loops and the number of times each loop executes.

```
(a) ires = 1;
    while mod(ires,10) ~= 0
        ires = ires + 1
    end
```



```
(b) ires = 2;
    while ires <= 200
        ires = ires^2;
    end
```

```
(c) ires = 2;
    while ires > 200
        ires = ires^2;
    end
```

**4.9** What is contained in array `arr1` after each of the following sets of statements are executed?

```
(a) arr1 = [1 2 3 4; 5 6 7 8; 9 10 11 12];
    mask = mod(arr1,2) == 0;
    arr1(mask) = -arr1(mask);
```

```
(b) arr1 = [1 2 3 4; 5 6 7 8; 9 10 11 12];
    arr2 = arr1 <= 5;
    arr1(arr2) = 0;
    arr1(~arr2) = arr1(~arr2).^2;
```

- 4.10** How can a logical array be made to behave as a logical mask for vector operations?
- 4.11** Modify program `ball` from Example 4.8 by replacing the inner `for` loops with vectorized calculations.
- 4.12** Modify program `ball` from Example 4.8 to read in the acceleration due to gravity at a particular location and to calculate the maximum range of the ball for that acceleration. After modifying the program, run it with accelerations of  $-9.8 \text{ m/sec}^2$ ,  $-9.7 \text{ m/sec}^2$ , and  $-9.6 \text{ m/sec}^2$ . What effect does the reduction in gravitational attraction have on the range of the ball? What effect does the reduction in gravitational attraction have on the best angle  $\theta$  at which to throw the ball?
- 4.13** Modify program `ball` from Example 4.8 to read in the initial velocity with which the ball is thrown. After modifying the program, run it with initial velocities of 10 m/sec, 20 m/sec, and 30 m/sec. What effect does changing the initial velocity  $v_0$  have on the range of the ball? What effect does it have on the best angle  $\theta$  at which to throw the ball?
- 4.14** Program `lsqfit` from Example 4.7 required the user to specify the number of input data points before entering the values. Modify the program so that it reads an arbitrary number of data values using a `while` loop, and stops reading input values when the user presses the Enter key without typing any values. Test your program using the same two data sets that were used in Example 4.7. (*Hint:* The `input` function returns an empty array (`[]`) if a user presses Enter without supplying any data. You can use function `isempty` to test for an empty array, and stop reading data when one is detected.)

- 4.15 Modify program `lsqfit` from Example 4.7 to read its input values from an ASCII file named `input1.dat`. The data in the file will be organized in rows, with one pair of  $(x, y)$  values on each row, as shown below:

```
1.1  2.2
2.2  3.3
```

Test your program using the same two data sets that were used in Example 4.6. (*Hint:* Use the `load` command to read the data into an array named `input1`, and then store the first column of `input1` into array `x` and the second column of `input1` into array `y`.)

- 4.16 **MATLAB Least-Squares Fit Function** MATLAB includes a standard function that performs a least-squares fit to a polynomial. Function `polyfit` calculates the least-squares fit of a data set to a polynomial of order  $N$ :

$$p(x) = a_n x^n + a_{n-1} x^{n-1} + \cdots + a_1 x + a_0 \quad (4-12)$$

where  $N$  can be any value greater than or equal to 1. Note that for  $N = 1$ , this polynomial is a linear equation, with the slope being the coefficient  $a_1$  and the  $y$ -intercept being the coefficient  $a_0$ . The form of this function is

$$p = \text{polyfit}(x, y, n)$$

where `x` and `y` are vectors of  $x$  and  $y$  components and `n` is the order of the fit.

Write a program that calculates the least-squares fit of a data set to a straight line using `polyfit`. Plot the input data points and the resulting fitted line. Compare the produced by the program using `polyfit` with the result produced by `lsqfit` for the input data set in Example 4.6.

- 4.17 Program `doy` in Example 4.3 calculates the day of year associated with any given month, day, and year. As written, this program does not check to see if the data entered by the user is valid. It will accept nonsense values for months and days, and do calculations with them to produce meaningless results. Modify the program so that it checks the input values for validity before using them. If the inputs are invalid, the program should tell the user what is wrong and quit. The year should be a number greater than zero, the month should be a number between 1 and 12, and the day should be a number between 1 and a maximum that depends on the month. Use a `switch` construct to implement the bounds checking performed on the day.
- 4.18 Write a MATLAB program to evaluate the function

$$y(x) = \ln \frac{1}{1-x}$$

for any user-specified value of  $x$ , where  $\ln$  is the natural logarithm (logarithm to the base  $e$ ). Write the program with a `while` loop, so that the program repeats the calculation for each legal value of  $x$  entered into the program. When an illegal value of  $x$  is entered, terminate the program. (Any  $x \geq 1$  is considered an illegal value.)

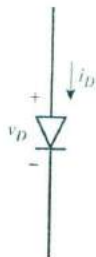


Figure 4.4 A semiconductor diode.

- 4.19 Fibonacci Numbers** The  $n$ th Fibonacci number is defined by the following recursive equations:

$$f(1) = 1$$

$$f(2) = 2$$

$$f(n) = f(n - 1) + f(n - 2)$$

Therefore,  $f(3) = f(2) + f(1) = 2 + 1 = 3$ , and so forth for higher numbers. Write an M-file to calculate and write out the  $n$ th Fibonacci number for  $n > 2$ , where  $n$  is input by the user. Use a `while` loop to perform the calculation.

- 4.20 Current Through a Diode** The current flowing through the semiconductor diode shown in Figure 4.4 is given by the equation

$$i_D = I_0 \left( e^{\frac{q v_D}{k T}} - 1 \right) \quad (4-13)$$

where  $i_D$  = the current flow through the diode, in amps

$v_D$  = the voltage across the diode, in volts

$I_0$  = the leakage current of the diode, in amps

$q$  = the charge on an electron,  $1.602 \times 10^{-19}$  coulombs

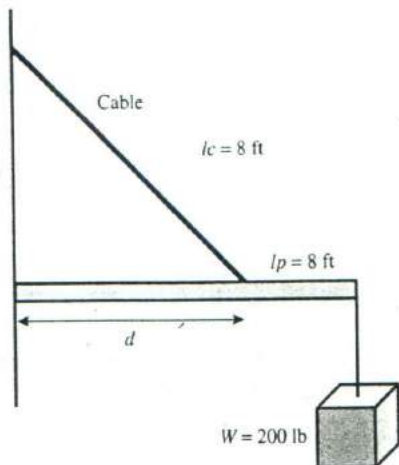
$k$  = Boltzmann's constant,  $1.38 \times 10^{-23}$  joule/K

$T$  = temperature, in kelvins (K)

The leakage current  $I_0$  of the diode is  $2.0 \mu\text{A}$ . Write a program to calculate the current flowing through this diode for all voltages from  $-1.0$  V to  $+0.6$  V, in  $0.1$  V steps. Repeat this process for the following temperatures:  $75^\circ\text{F}$  and  $100^\circ\text{F}$ , and  $125^\circ\text{F}$ . Create a plot of the current as a function of applied voltage, with the curves for the three different temperatures appearing as different colors.

- 4.21 Tension on a Cable** A 200-pound object is to be hung from the end of a rigid 8-foot horizontal pole of negligible weight, as shown in Figure 4.5. The pole is attached to a wall by a pivot and is supported by an 8-foot cable that is attached to the wall at a higher point. The tension on this cable is given by the equation

$$T = \frac{W \cdot lc \cdot lp}{d \sqrt{lp^2 - d^2}} \quad (4-14)$$



**Figure 4.5** A 200-pound weight suspended from a rigid bar supported by a cable.

where  $T$  is the tension on the cable,  $W$  is the weight of the object,  $l_c$  is the length of the cable,  $l_p$  is the length of the pole, and  $d$  is the distance along the pole at which the cable is attached. Write a program to determine the distance  $d$  at which to attach the cable to the pole in order to minimize the tension on the cable. To do this, the program should calculate the tension on the cable at regular one-foot intervals from  $d = 1$  foot to  $d = 7$  feet, and should locate the position  $d$  that produces the minimum tension. Also, the program should plot the tension on the cable as a function of  $d$ , with appropriate titles and axis labels.

- 4.22 Bacterial Growth** Suppose that a biologist performs an experiment in which he or she measures the rate at which a specific type of bacterium reproduces asexually in different culture media. The experiment shows that in Medium A the bacteria reproduce once every 60 minutes, and in Medium B the bacteria reproduce once every 90 minutes. Assume that a single bacterium is placed on each culture medium at the beginning of the experiment. Write a program that calculates and plots the number of bacteria present in each culture at intervals of three hours from the beginning of the experiment until 24 hours have elapsed. Make two plots, one a linear  $xy$  plot and the other a linear-log (semi log) plot. How do the numbers of bacteria compare on the two media after 24 hours?
- 4.23 Decibels** Engineers often measure the ratio of two power measurements in *decibels*, or dB. The equation for the ratio of two power measurements in decibels is

$$\text{dB} = 10 \log_{10} \frac{P_2}{P_1} \quad (4-15)$$

where  $P_2$  is the power level being measured, and  $P_1$  is some reference power level. Assume that the reference power level  $P_1$  is 1 watt, and write a program that calculates the decibel level corresponding to power levels between 1 and 20 watts, in 0.5 W steps. Plot the dB-versus-power curve on a log-linear scale.

- 4.24 Geometric Mean** The *geometric mean* of a set of numbers  $x_1$  through  $x_n$  is defined as the  $n$ th root of the product of the numbers:

$$\text{geometric mean} = \sqrt[n]{x_1 x_2 x_3 \dots x_n} \quad (4-16)$$

Write a MATLAB program that will accept an arbitrary number of positive input values and calculate both the arithmetic mean (i.e., the average) and the geometric mean of the numbers. Use a `while` loop to get the input values, and terminate the inputs when a user enters a negative number. Test your program by calculating the average and geometric mean of the four numbers 10, 5, 2, and 5.

- 4.25 RMS Average** The *root-mean-square (rms) average* is another way of calculating a mean for a set of numbers. The rms average of a series of numbers is the square root of the arithmetic mean of the squares of the numbers:

$$\text{rms average} = \sqrt{\frac{1}{N} \sum_{i=1}^N x_i^2} \quad (4-17)$$

Write a MATLAB program that will accept an arbitrary number of positive input values and calculate the rms average of the numbers. Prompt the user for the number of values to be entered, and use a `for` loop to read in the numbers. Test your program by calculating the rms average of the four numbers 10, 5, 2, and 5.

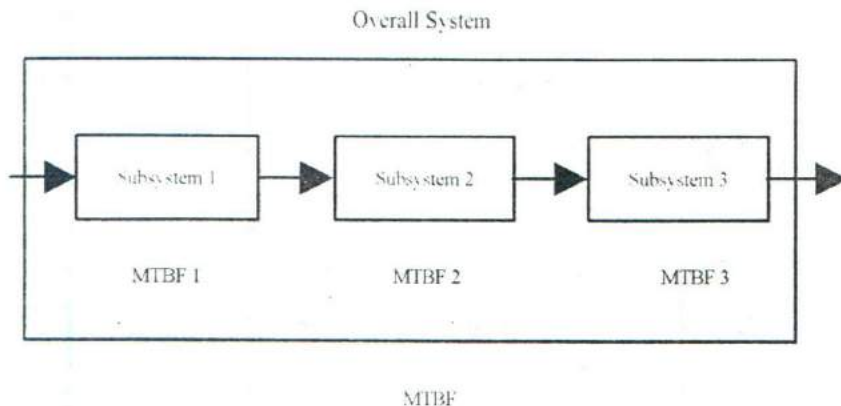
- 4.26 Harmonic Mean** The *harmonic mean* is yet another way of calculating a mean for a set of numbers. The harmonic mean of a set of numbers is given by the equation:

$$\text{harmonic mean} = \frac{N}{\frac{1}{x_1} + \frac{1}{x_2} + \dots + \frac{1}{x_N}} \quad (4-18)$$

Write a MATLAB program that will read in an arbitrary number of positive input values and calculate the harmonic mean of the numbers. Use any method that you desire to read in the input values. Test your program by calculating the harmonic mean of the four numbers 10, 5, 2, and 5.

- 4.27** Write a single program that calculates the arithmetic mean (average), rms average, geometric mean, and harmonic mean for a set of positive numbers. Use any method that you desire to read in the input values. Compare these values for each of the following sets of numbers:

- (a) 4, 4, 4, 4, 4, 4, 4
- (b) 4, 3, 4, 5, 4, 3, 5
- (c) 4, 1, 4, 7, 4, 1, 7
- (d) 1, 2, 3, 4, 5, 6, 7



**Figure 4.6** An electronic system containing three subsystems with known MTBFs.

- 4.28 Mean Time Between Failure Calculations** The reliability of a piece of electronic equipment is usually measured in terms of mean time between failures (MTBF), where MTBF is the average time that the piece of equipment can operate before a failure occurs in it. For large systems containing many pieces of electronic equipment, it is customary to determine the MTBFs of each component, and to calculate the overall MTBF of the system from the failure rates of the individual components. If the system is structured like the one shown in Figure 4.6, every component must work in order for the whole system to work, and the overall system MTBF can be calculated as

$$\text{MTBF}_{\text{sys}} = \frac{1}{\frac{1}{\text{MTBF}_1} + \frac{1}{\text{MTBF}_2} + \cdots + \frac{1}{\text{MTBF}_n}} \quad (4-19)$$

Write a program that reads in the number of series components in a system and the MTBFs for each component, and then calculates the overall MTBF for the system. To test your program, determine the MTBF for a radar system consisting of an antenna subsystem with an MTBF of 2000 hours, a transmitter with an MTBF of 800 hours, a receiver with an MTBF of 3000 hours, and a computer with an MTBF of 5000 hours.