CHAPTER 3

# Introduction to MATLAB Programming

## KEY TERMS

| | | |
|---|---|---|
| computer program | comments | toggle |
| scripts | block comment | modes |
| algorithm | comment blocks | writing to a file |
| modular program | input/output (I/O) | appending to a file |
| top-down design | user | reading from a file |
| external file | empty string | user-defined functions |
| default input device | error message | function call |
| prompting | formatting | argument |
| default output device | format string | control |
| execute/run | place holder | return value |
| high level languages | conversion characters | function header |
| machine language | newline character | output arguments |
| executable | field width | input arguments |
| compiler | leading blanks | function body |
| source code | trailing zeros | function definition |
| object code | plot symbols | local variables |
| interpreter | markers | scope of variables |
| documentation | line types | base workspace |

## CONTENTS



We have now used the MATLAB® product interactively in the Command Window. That is sufficient when all one needs is a simple calculation. However, in many cases, quite a few steps are required before the final result can be obtained. In those cases, it is more convenient to group statements together in what is called a ***computer program***.

In this chapter, we will introduce the simplest MATLAB programs, which are called ***scripts***. Examples of scripts that customize simple plots will illustrate the concept. Input will be introduced, both from files and from the user. Output



MATLAB®. http://dx.doi.org/10.1016/B978-0-12-405876-7.00003-1

to files and to the screen will also be introduced. Finally, user-defined functions that calculate and return a single value will be described. These topics serve as an introduction to programming, which will be expanded on in Chapter 6.

## 3.1 ALGORITHMS

Before writing any computer program, it is useful to first outline the steps that will be necessary. An ***algorithm*** is the sequence of steps needed to solve a problem. In a ***modular*** approach to programming, the problem solution is broken down into separate steps, and then each step is further refined until the resulting steps are small enough to be manageable tasks. This is called the ***top-down design*** approach.

As a simple example, consider the problem of calculating the area of a circle. First, it is necessary to determine what information is needed to solve the problem, which, in this case, is the radius of the circle. Next, given the radius of the circle, the area of the circle would be calculated. Finally, once the area has been calculated, it has to be displayed in some way. The basic algorithm then is three steps:

- get the input—the radius
- calculate the result—the area
- display the output.

Even with an algorithm this simple, it is possible to further refine each of the steps. When a program is written to implement this algorithm, the steps would be as follows.

- Where does the input come from? Two possible choices would be from an ***external file*** or from the user (the person who is running the program) who enters the number by typing it from the keyboard. For every system, one of these will be the ***default input device*** (which means, if not specified otherwise, this is where the input comes from!). If the user is supposed to enter the radius, the user has to be told to type in the radius (and in what units). Telling the user what to enter is called ***prompting.*** So, the input step actually becomes two steps: prompt the user to enter a radius and then read it into the program.
- To calculate the area, the formula is needed. In this case, the area of the circle is $\pi$ multiplied by the square of the radius. So, that means the value of the constant for $\pi$ is needed in the program.
- Where does the output go? Two possibilities are (1) to an external file or (2) to the screen. Depending on the system, one of these will be the ***default output device.*** When displaying the output from the program, it should always be as informative as possible. In other words, instead of just

printing the area (just the number), it should be printed in a nice sentence format. Also, to make the output even more clear, the input should be printed. For example, the output might be the sentence "For a circle with a radius of 1 inch, the area is 3.1416 inches squared".

For most programs, the basic algorithm consists of the three steps that have been outlined:

1. Get the input(s)
2. Calculate the result(s)
3. Display the result(s).

As can be seen here, even the simplest problem solutions can then be refined further. This is top-down design.

## 3.2 MATLAB SCRIPTS

Once a problem has been analyzed, and the algorithm for its solution has been written and refined, the solution to the problem is then written in a particular programming language. A computer program is a sequence of instructions, in a given language, that accomplishes a task. To ***execute***, or ***run***, a program is to have the computer actually follow these instructions sequentially.

***High-level languages*** have English-like commands and functions, such as "print this" or "if $x < 5$ do something". The computer, however, can only interpret commands written in its ***machine language***. Programs that are written in high-level languages must therefore be translated into machine language before the computer can actually execute the sequence of instructions in the program. A program that does this translation from a high-level language to an ***executable*** file is called a ***compiler***. The original program is called the ***source code***, and the resulting executable program is called the ***object code***. Compilers translate from the source code to object code; this is then executed as a separate step.

By contrast, an ***interpreter*** goes through the code line-by-line, translating and executing each command as it goes. MATLAB uses what are called either script files, or M-files (the reason for this is that the extension on the filename is .m). These script files are interpreted, rather than compiled. Therefore, the correct terminology is that these are scripts and not programs. However, the terms are somewhat loosely used by many people, and the documentation in MATLAB itself refers to scripts as programs. In this book, we will reserve the use of the word "program" to mean a set of scripts and functions, as described briefly in Section 3.7 and then in more detail in Chapter 6.

A script is a sequence of MATLAB instructions that is stored in an M-file and saved. The contents of a script can be displayed in the Command Window using the **type** command. The script can be executed, or run, by simply entering the name of the file (without the .m extension).

Before creating a script, make sure the Current Folder (called "Current Directory" in earlier versions) is set to the folder in which you want to save your files.

The steps involved in creating a script depend on the version of MATLAB. In the most recent versions the easiest method is to click on "New Script" under the HOME tab. Alternatively, you can click on the down arrow under "New" and then choose Script (see Figure 3.1)

In earlier versions, one would click on File, then New, then Script (or, in even earlier versions, M-file). A new window will appear called the Editor (which can be docked). In the latest versions of MATLAB, this window has three tabs: "EDITOR", "PUBLISH", and "VIEW". Next, simply type the sequence of statements (note that line numbers will appear on the left).

When finished, save the file by choosing the Save down arrow under the EDITOR tab or, in earlier versions of MATLAB, by choosing File and then Save. Make sure that the extension of .m is on the filename (this should be the default). The rules for file names are the same as for variables (they must start with a letter; after that there can be letters, digits, or the underscore). For example, we will now create a script called *script1.m* that calculates the area of a circle. It assigns a value for the radius, and then calculates the area based on that radius.

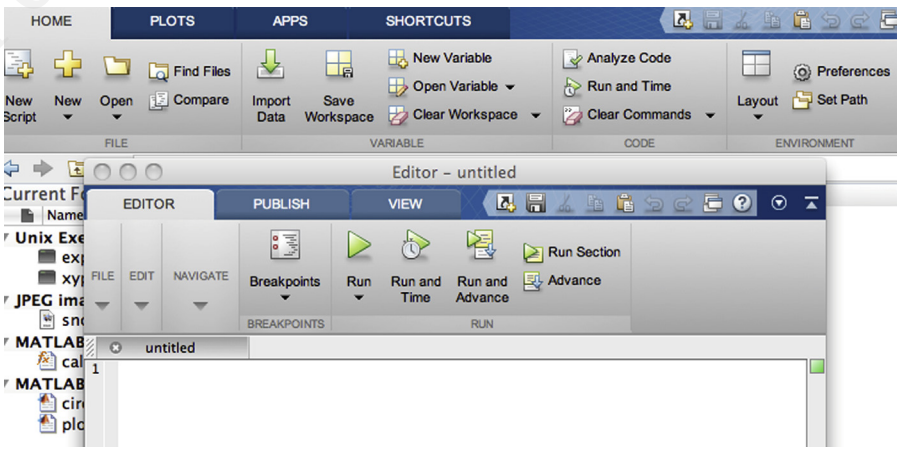


**FIGURE 3.1** Toolstrip and editor

In this text, scripts will be displayed in a box with the name of the M-file on top.

script1.m

```
radius = 5
area = pi * (radius^2)
```

There are two ways to view a script once it has been written: either open the Editor Window to view it or use the **type** command, as shown here, to display it in the Command Window. The **type** command shows the contents of the file named *script1.m*; notice that the .m is not included:

```
>> type script1
radius = 5
area = pi * (radius^2)
```

To actually run or execute the script from the Command Window, the name of the file is entered at the prompt (again, without the .m). When executed, the results of the two assignment statements are displayed, as the output was not suppressed for either statement.

```
>> script1
radius =
     5
area =
    78.5398
```

Once the script has been executed you may find that you want to make changes to it (especially if there are errors!). To edit an existing file, there are several methods to open it. The easiest are:

- within the Current Folder Window, double-click on the name of the file in the list of files
- choosing the Open down arrow will show a list of Recent Files.

### 3.2.1 Documentation

It is very important that all scripts be ***documented*** well, so that people can understand what the script does and how it accomplishes its task. One way of documenting a script is to put ***comments*** in it. In MATLAB, a comment is anything from a `%` to the end of that particular line. Comments are completely ignored when the script is executed. To put in a comment, simply type the `%` symbol at the beginning of a line, or select the comment lines and then click on the Edit down arrow and click on the % symbol, and the Editor will put in the `%` symbols at the beginning of those lines for the comments.

For example, the previous script to calculate the area of a circle could be modified to have comments:

circlescript.m

```
% This script calculates the area of a circle

% First the radius is assigned
radius = 5
% The area is calculated based on the radius
area = pi * (radius^2)
```

The first comment at the beginning of the script describes what the script does; this is sometimes called a ***block comment***. Then, throughout the script, comments describe different parts of the script (not usually a comment for every line, however!). Comments don't affect what a script does, so the output from this script would be the same as for the previous version.

The **help** command in MATLAB works with scripts as well as with built-in functions. The first block of comments (defined as contiguous lines at the beginning) will be displayed. For example, for *circlescript*:

```
>> help circlescript
  This script calculates the area of a circle
```

The reason that a blank line was inserted in the script between the first two comments is that otherwise both would have been interpreted as one contiguous comment, and both lines would have been displayed with **help**. The very first comment line is called the "H1 line"; it is what the function **lookfor** searches through.

### PRACTICE 3.1

Write a script to calculate the circumference of a circle ($C = 2 \pi r$). Comment the script.

Longer comments, called ***comment blocks***, consist of everything in between %{and %}, which must be alone on separate lines. For example:

```
%{
   this is a really
   Really
   REALLY
   long comment
%}
```

## 3.3 INPUT AND OUTPUT

The previous script would be much more useful if it were more general; for example, if the value of the radius could be read from an external source rather

than being assigned in the script. Also, it would be better to have the script print the output in a nice, informative way. Statements that accomplish these tasks are called ***input/output*** statements, or ***I/O*** for short. Although, for simplicity, examples of input and output statements will be shown here in the Command Window, these statements will make the most sense in scripts.

### 3.3.1 Input Function

Input statements read in values from the default or standard input device. In most systems, the default input device is the keyboard, so the input statement reads in values that have been entered by the ***user***, or the person who is running the script. To let the user know what he or she is supposed to enter, the script must first prompt the user for the specified values.

The simplest input function in MATLAB is called **input**. The **input** function is used in an assignment statement. To call it, a string is passed that is the prompt that will appear on the screen, and whatever the user types will be stored in the variable named on the left of the assignment statement. For ease of reading the prompt, it is useful to put a colon and then a space after the prompt. For example,

```
>> rad = input('Enter the radius: ')
Enter the radius: 5
rad =
    5
```

If character or string input is desired, 's' must be added as a second argument to the **input** function:

```
>> letter = input('Enter a char: ','s')
Enter a char: g
letter =
g
```

If the user enters only spaces or tabs before hitting the Enter key, they are ignored and an ***empty string*** is stored in the variable:

```
>> mychar = input('Enter a character: ', 's')
Enter a character:
mychar =
    ''
```

**Note**

Although normally the quotes are not shown around a character or string, in this case they are shown to demonstrate that there is nothing inside of the string.

However, if blank spaces are entered before other characters, they are included in the string. In the next example, the user hit the space bar four times before entering "go". The **length** function returns the number of characters in the string.

```
>> mystr = input('Enter a string: ', 's')
Enter a string:     go
mystr =
    go
>> length(mystr)
ans =
     6
```

## QUICK QUESTION!

What would be the result if the user enters blank spaces after other characters? For example, the user here entered "xyz " (four blank spaces):

```
>> mychar = input('Enter chars: ', 's')
Enter chars: xyz
mychar =
xyz
>> length(mychar)
ans =
     7
```

The length can be seen in the Command Window by using the mouse to highlight the value of the variable; the xyz and four spaces will be highlighted.

### Answer

The space characters would be stored in the string variable. It is difficult to see above, but is clear from the length of the string.

It is also possible for the user to type quotation marks around the string rather than including the second argument 's' in the call to the **input** function.

```
>> name = input('Enter your name: ')
Enter your name: 'Stormy'
name =
Stormy
```

However, this assumes that the user would know to do this so it is better to signify that character input is desired in the **input** function itself. Also, if the 's' is specified and the user enters quotation marks, these would become part of the string.

```
>> name = input('Enter your name: ','s')
Enter your name: 'Stormy'
name =
'Stormy'
>> length(name)
ans =
     8
```

Note what happens if string input has not been specified, but the user enters a letter rather than a number.

```
>> num = input('Enter a number: ')
Enter a number: t
Error using input
Undefined function or variable 't'.

Enter a number: 3
num =
     3
```

MATLAB gave an ***error message*** and repeated the prompt. However, if *t* is the name of a variable, MATLAB will take its value as the input.

```
>> t = 11;
>> num = input('Enter a number: ')
Enter a number: t
num =
    11
```

Separate **input** statements are necessary if more than one input is desired. For example,

```
>> x = input('Enter the x coordinate: ');
>> y = input('Enter the y coordinate: ');
```

Normally in a script the results from **input** statements are suppressed with a semicolon at the end of the assignment statements.

## PRACTICE 3.2

Create a script that would prompt the user for a length, and then use 'f' for feet or 'm' for meters, and store both inputs in variables. For example, when executed it would look like this (assuming the user enters 12.3 and then m):

```
Enter the length: 12.3
Is that f(eet)or m(eters)?: m
```

### 3.3.2 Output Statements: disp and fprintf

Output statements display strings and/or the results of expressions, and can allow for ***formatting***, or customizing how they are displayed. The simplest output function in MATLAB is **disp**, which is used to display the result of an expression or a string without assigning any value to the default variable *ans*. However, **disp** does not allow formatting. For example,

```
>> disp('Hello')
Hello

>> disp(4^3)
    64
```

Formatted output can be printed to the screen using the **fprintf** function. For example,

```
>> fprintf('The value is %d, for sure!\n',4^3)
The value is 64, for sure!
>>
```

To the **fprintf** function, first a string (called the ***format string***) is passed that contains any text to be printed, as well as formatting information for the expressions to be printed. In this example, the %d is an example of format information.

The %d is sometimes called a ***place holder*** because it specifies where the value of the expression that is after the string is to be printed. The character in the place holder is called the ***conversion character***, and it specifies the type of value that is being printed. There are others, but what follows is a list of the simple place holders:

```
%d  integer (it stands for decimal integer)
%f  float (real number)
%c  single character
%s  string
```

**Note**
Don't confuse the % in the place holder with the symbol used to designate a comment.

The character '\n' at the end of the string is a special character called the ***newline character***; what happens when it is printed is that the output that follows moves down to the next line.

## QUICK QUESTION!

What do you think would happen if the newline character is omitted from the end of an **fprintf** statement?

**Answer**
Without it, the next prompt would end up on the same line as the output. It is still a prompt, and so an expression can be entered, but it looks messy as shown here.

```
>> fprintf('The value is %d, surely!',...
4^3)
The value is 64, surely!>> 5 + 3
ans =
     8
```

Note that with the **disp** function, however, the prompt will always appear on the next line:

```
>> disp('Hi')
Hi
>>
```

Also, note that an ellipsis can be used after a string but not in the middle.

## QUICK QUESTION!

How can you get a blank line in the output?

**Answer**
Have two newline characters in a row.

```
>> fprintf('The value is %d,\n\nOK!\n',4^3)
The value is 64,

OK!
```

This also points out that the newline character can be anywhere in the string; when it is printed, the output moves down to the next line.

Note that the newline character can also be used in the prompt in the **input** statement; for example:

```
>> x = input('Enter the \nx coordinate: ');
Enter the
x coordinate: 4
```

However, that is the only formatting character allowed in the prompt in **input**.

To print two values, there would be two place holders in the format string, and two expressions after the format string. The expressions fill in for the place holders in sequence.

```
>> fprintf('The int is %d and the char is %c\n', ...
      33 - 2, 'x')
The int is 31 and the char is x
```

A ***field width*** can also be included in the place holder in **fprintf**, which specifies how many characters total are to be used in printing. For example, %5d would indicate a field width of 5 for printing an integer and %10s would indicate a field width of 10 for a string. For floats, the number of decimal places can also be specified; for example, %6.2f means a field width of 6 (including the decimal point and the two decimal places) with 2 decimal places. For floats, just the number of decimal places can also be specified; for example, %.3f indicates 3 decimal places, regardless of the field width.

**Note**

If the field width is wider than necessary, ***leading blanks*** are printed, and if more decimal places are specified than necessary, ***trailing zeros*** are printed.

```
>> fprintf('The int is %3d and the float is %6.2f\n', 5, 4.9)
The int is 5 and the float is 4.90
```

## QUICK QUESTION!

What do you think would happen if you tried to print 1234.5678 in a field width of 3 with 2 decimal places?

```
>> fprintf('%3.2f\n', 1234.5678)
```

**Answer**

It would print the entire 1234, but round the decimals to two places, that is

```
1234.57
```

If the field width is not large enough to print the number, the field width will be increased. Basically, to cut the number off would give a misleading result, but rounding the decimal places does not change the number by much.

## QUICK QUESTION!

What would happen if you use the %d conversion character, but you're trying to print a real number?

**Answer**

MATLAB will show the result using exponential notation

```
>> fprintf('%d\n',1234567.89)
1.234568e+006
```

Note that if you want exponential notation, this is not the correct way to get it; instead, there are conversion characters that can be used. Use the **help** browser to see this option, as well as many others!

There are many other options for the format string. For example, the value being printed can be left-justified within the field width using a minus sign. The following example shows the difference between printing the integer 3 using %5d and using %-5d. The x's below are used to show the spacing.

```
>> fprintf('The integer is xx%5dxx and xx%-5dxx\n',3,3)
The integer is xx    3xx and xx3    xx
```

Also, strings can be truncated by specifying "decimal places":

```
>> fprintf('The string is %s or %.2s\n', 'street', 'street')
The string is street or st
```

There are several special characters that can be printed in the format string in addition to the newline character. To print a slash, two slashes in a row are used, and also to print a single quote, two single quotes in a row are used. Additionally, '\t' is the tab character.

```
>> fprintf('Try this out: tab\t quote '' slash \\ \n')
Try this out: tab quote ' slash \
```

### 3.3.2.1 Printing Vectors and Matrices

For a vector, if a conversion character and the newline character are in the format string, it will print in a column regardless of whether the vector itself is a row vector or a column vector.

```
>> vec = 2:5;
>> fprintf('%d\n', vec)
2
3
4
5
```

Without the newline character, it would print in a row, but the next prompt would appear on the same line:

```
>> fprintf('%d', vec)
2345>>
```

However, in a script, a separate newline character could be printed to avoid this problem. It is also much better to separate the numbers with spaces.

printvec.m

```
% This demonstrates printing a vector

vec = 2:5;
fprintf('%d ',vec)
fprintf('\n')
```

```
>> printvec
2 3 4 5
>>
```

If the number of elements in the vector is known, that many conversion characters can be specified and then the newline:

```
>> fprintf('%d %d %d %d\n', vec)
2 3 4 5
```

This is not very general, however, and is therefore not preferable.

For matrices, MATLAB unwinds the matrix column by column. For example, consider the following *2 x 3* matrix:

```
>> mat = [5  9  8; 4  1  10]
mat =
     5     9     8
     4     1    10
```

Specifying one conversion character and then the newline character will print the elements from the matrix in one column. The first values printed are from the first column, then the second column, and so on.

```
>> fprintf('%d\n', mat)
5
4
9
1
8
10
```

If three of the %d conversion characters are specified, the **fprintf** will print three numbers across on each line of output, but again the matrix is unwound column–by-column. It again prints first the two numbers from the first column (across on the first row of output), then the first value from the second column, and so on.

```
>> fprintf('%d %d %d\n', mat)
5 4 9
1 8 10
```

If the transpose of the matrix is printed, however, using the three %d conversion characters, the matrix is printed as it appears when created.

```
>> fprintf('%d %d %d\n', mat') % Note the transpose
5 9 8
4 1 10
```

For vectors and matrices, even though formatting cannot be specified, the **disp** function may be easier to use in general than **fprintf** because it displays the result in a straightforward manner. For example,

```
>> mat = [15 11 14; 7 10 13]
mat =
    15    11    14
     7    10    13

>> disp(mat)
    15    11    14
     7    10    13

>> vec = 2:5
vec =
     2     3     4     5

>> disp(vec)
     2     3     4     5
```

Note that when loops are covered in Chapter 5, formatting the output of matrices will be easier. For now, however, **disp** works well.

## 3.4 SCRIPTS WITH INPUT AND OUTPUT

Putting all of this together now, we can implement the algorithm from the beginning of this chapter. The following script calculates and prints the area of a circle. It first prompts the user for a radius, reads in the radius, and then calculates and prints the area of the circle based on this radius.

circleIO.m

```
% This script calculates the area of a circle
% It prompts the user for the radius

% Prompt the user for the radius and calculate
% the area based on that radius
fprintf('Note: the units will be inches.\n')
radius = input('Please enter the radius: ');
area = pi * (radius^2);

% Print all variables in a sentence format
fprintf('For a circle with a radius of %.2f inches,\n', radius)
fprintf('the area is %.2f inches squared\n',area)
```

Executing the script produces the following output:

```
>> circleIO
Note: the units will be inches.
Please enter the radius: 3.9
For a circle with a radius of 3.90 inches,
the area is 47.78 inches squared
```

Note that the output from the first two assignment statements (including the **input**) is suppressed by putting semicolons at the end. That is usually done in scripts, so that the exact format of what is displayed by the program is controlled by the **fprintf** functions.

### PRACTICE 3.3

Write a script to prompt the user separately for a character and a number, and print the character in a field width of 3 and the number left-justified in a field width of 8 with 3 decimal places. Test this by entering numbers with varying widths.

## 3.5 SCRIPTS TO PRODUCE AND CUSTOMIZE SIMPLE PLOTS

MATLAB has many graphing capabilities. Customizing plots is often desired, and this is easiest to accomplish by creating a script rather than typing one command at a time in the Command Window. For that reason, simple plots and how to customize them will be introduced in this chapter on MATLAB programming.

The help topics that contain graph functions include **graph2d** and **graph3d**. Typing **help graph2d** would display some of the two-dimensional graph functions, as well as functions to manipulate the axes and to put labels and titles on the graphs. The Search Documentation under MATLAB Graphics also has a section "2-D and 3-D Plots".

### 3.5.1 The Plot Function

For now, we'll start with a very simple graph of one point using the **plot** function.

The following script, *plotonepoint*, plots one point. To do this, first values are given for the x and y coordinates of the point in separate variables. The point is plotted using a red star ('*'). The plot is then customized by specifying the minimum and maximum values on first the x and then y axes. Labels are then put on the x-axis, the y-axis, and the graph itself using the functions **xlabel**, **ylabel**, and **title**. (Note: there are no default labels for the axes.)

All of this can be done from the Command Window, but it is much easier to use a script. The following shows the contents of the script *plotonepoint* that

accomplishes this. The x coordinate represents the time of day (e.g., 11 a.m.) and the y coordinate represents the temperature (e.g., in degrees Fahrenheit) at that time.

plotonepoint.m

```
% This is a really simple plot of just one point!
% Create coordinate variables and plot a red '*'
x = 11;
y = 48;
plot(x,y,'r*')
% Change the axes and label them axis([9 12 35 55])
xlabel('Time')
ylabel('Temperature')
% Put a title on the plot
title('Time and Temp')
```

In the call to the **axis** function, one vector is passed. The first two values are the minimum and maximum for the x-axis, and the last two are the minimum and maximum for the y-axis. Executing this script brings up a Figure Window with the plot (see Figure 3.2).

To be more general, the script could prompt the user for the time and temperature, rather than just assigning values. Then, the axis function could be used based on whatever the values of x and y are, as in the following example:

```
axis([x-2 x+2 y-10 y+10])
```

In addition, although they are the x and y coordinates of a point, variables named *time* and *temp* might be more mnemonic than *x* and *y*.

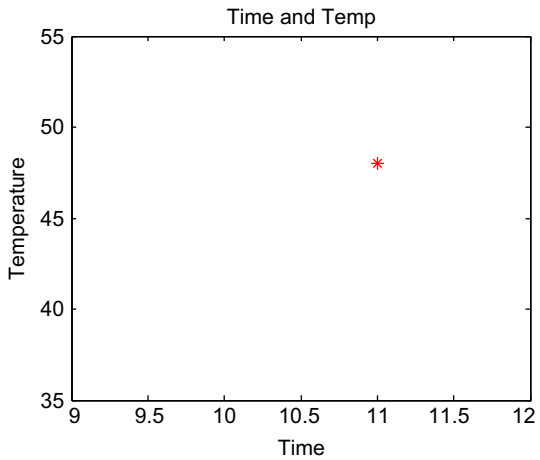


**FIGURE 3.2** Plot of one data point

## PRACTICE 3.4

Modify the script *plotonepoint* to prompt the user for the time and temperature, and set the axes based on these values.

To plot more than one point, $x$ and $y$ vectors are created to store the values of the (x,y) points. For example, to plot the points

```
(1,1)
(2,5)
(3,3)
(4,9)
(5,11)
(6,8)
```

first an $x$ vector is created that has the x values (as they range from 1 to 6 in steps of 1, the colon operator can be used) and then a $y$ vector is created with the y values. The following will create (in the Command Window) $x$ and $y$ vectors and then plot them (see Figure 3.3).

```
>> x = 1:6;
>> y = [1 5 3 9 11 8];
>> plot(x,y)
```

Note that the points are plotted with straight lines drawn in between. Also, the axes are set up according to the data; for example, the x values range from 1 to 6 and the y values from 1 to 11, so that is how the axes are set up.

Also, note that in this case the x values are the indices of the $y$ vector (the $y$ vector has six values in it, so the indices iterate from 1 to 6). When this is the case it is not necessary to create the $x$ vector. For example,

```
>> plot(y)
```

will plot exactly the same figure without using an $x$ vector.

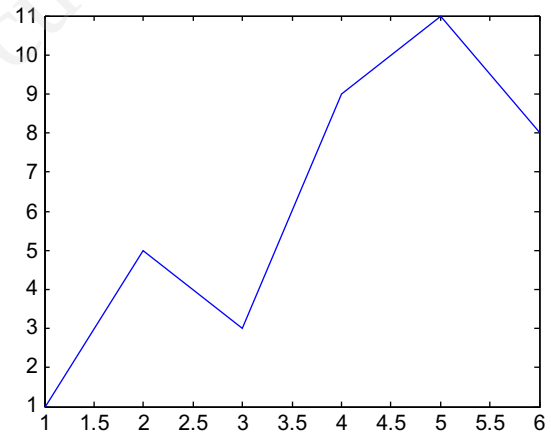


**FIGURE 3.3** Plot of data points from vectors

### 3.5.1.1 Customizing a Plot: Color, Line Types, Marker Types

Plots can be done in the Command Window, as shown here, if they are really simple. However, many times it is desired to customize the plot with labels, titles, and so on, so it makes more sense to do this in a script. Using the **help** function for **plot** will show the many options such as the line types and colors. In the previous script *plotonepoint*, the string 'r*' specified a red star for the point type. The LineSpec, or line specification, can specify up to three different properties in a string, including the color, line type, and the symbol or marker used for the data points.

The possible colors are:

```
b  blue
c  cyan
g  green
k  black
m  magenta
r  red
w  white
y  yellow
```

Either the single character listed above or the full name of the color can be used in the string to specify the color. The ***plot symbols***, or ***markers***, that can be used are:

```
o  circle
d  diamond
h  hexagram
p  pentagram
+  plus
.  point
s  square
*  star
v  down triangle
<  left triangle
>  right triangle
^  up triangle
x  x-mark
```

***Line types*** can also be specified by the following:

```
--  dashed
-.  dash dot
:   dotted
-   solid
```

If no line type is specified, a solid line is drawn between the points, as seen in the last example.

## 3.5.2 Simple Related Plot Functions

Other functions that are useful in customizing plots include **clf**, **figure**, **hold**, **legend**, and **grid**. Brief descriptions of these functions are given here; use **help** to find out more about them.

`clf` clears the Figure Window by removing everything from it.

`figure` creates a new, empty Figure Window when called without any arguments. Calling it as `figure(n)` where *n* is an integer is a way of creating and maintaining multiple Figure Windows, and of referring to each individually.

`hold` is a toggle that freezes the current graph in the Figure Window, so that new plots will be superimposed on the current one. Just `hold` by itself is a ***toggle***, so calling this function once turns the hold on, and then the next time turns it off. Alternatively, the commands `hold on` and `hold off` can be used.

`legend` displays strings passed to it in a legend box in the Figure Window in order of the plots in the Figure Window

`grid` displays grid lines on a graph. Called by itself, it is a toggle that turns the grid lines on and off. Alternatively, the commands `grid on` and `grid off` can be used.

Also, there are many plot types. We will see more in Chapter 11, but another simple plot type is a **bar** chart.

For example, the following script creates two separate Figure Windows. First, it clears the Figure Window. Then, it creates an *x* vector and two different y vectors (*y1* and *y2*). In the first Figure Window, it plots the *y1* values using a bar chart. In the second Figure Window, it plots the *y1* values as black lines, puts **hold on** so that the next graph will be superimposed, and plots the *y2* values as black circles. It also puts a legend on this graph and uses a grid. Labels and titles are omitted in this case as it is generic data.

plot2figs.m

```
% This creates 2 different plots, in 2 different
% Figure Windows, to demonstrate some plot features

clf
x = 1:5; % Not necessary
y1 = [2 11 6 9 3];
y2 = [4 5 8 6 2];
% Put a bar chart in Figure 1
figure(1)
bar(x,y1)
% Put plots using different y values on one plot
% with a legend
figure(2)
plot(x,y1,'k')
hold on
plot(x,y2,'ko')
grid on
legend('y1','y2')
```

Running this script will produce two separate Figure Windows. If there are no other active Figure Windows, the first, which is the bar chart, will be in the one titled "Figure 1" in MATLAB. The second will be in "Figure 2". See Figure 3.4 for both plots.

Note that the first and last points are on the axes, which makes them difficult to see. That is why the **axis** function is used frequently, as it creates space around the points so that they are all visible.

## PRACTICE 3.5

Modify the *plot2figs* script using the **axis** function so that all points are easily seen.

The ability to pass a vector to a function and have the function evaluate every element of the vector can be very useful in creating plots. For example, the following script displays graphically the difference between the **sin** and **cos** functions:

sinncos.m

```
% This script plots sin(x) and cos(x) in the same Figure Window
% for values of x ranging from 0 to 2*pi

clf
x = 0: 2*pi/40: 2*pi;
y = sin(x);
plot(x,y,'ro')
hold on
y = cos(x);
plot(x,y,'b+')
legend('sin', 'cos')
xlabel('x')
ylabel('sin(x) or cos(x)')
title('sin and cos on one graph')
```

The script creates an *x* vector; iterating through all of the values from 0 to 2*π in steps of 2*π/40 gives enough points to get a good graph. It then finds the sine of each *x* value, and plots these points using red circles. The command **hold on** freezes this in the Figure Window so the next plot will be superimposed. Next, it finds the cosine of each *x* value and plots these points using blue plus symbols (+). The **legend** function creates a legend; the first string is paired with the first plot and the second string with the second plot. Running this script produces the plot seen in Figure 3.5.

Note that instead of using **hold on**, both functions could have been plotted using one call to the **plot** function:

```
plot(x,sin(x),'ro',x,cos(x),'b+')
```

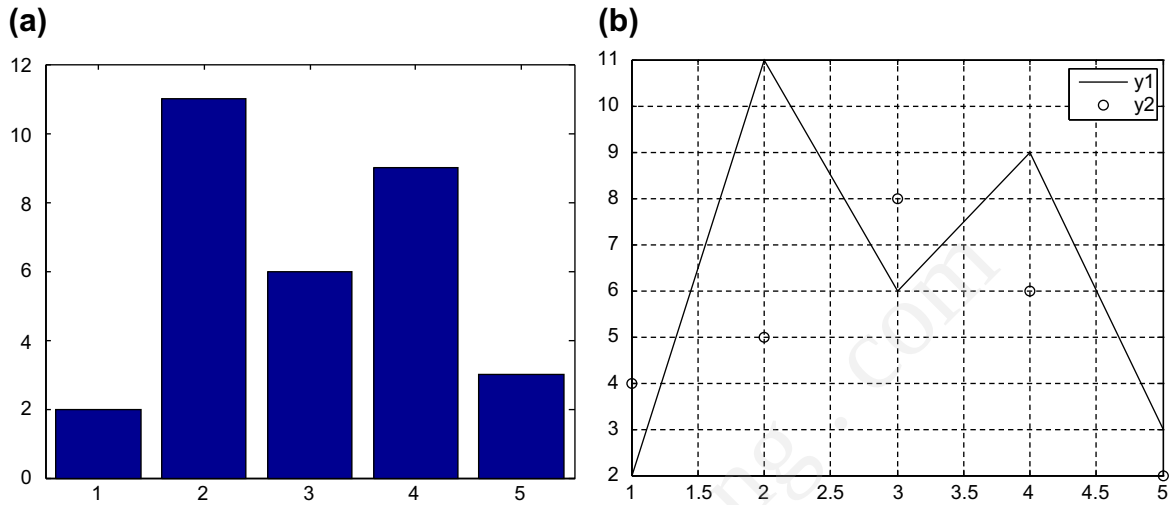


**FIGURE 3.4** (a) Bar chart produced by script. (b) Plot produced by script, with a grid and legend

## PRACTICE 3.6

Write a script that plots **exp(x)** and **log(x)** for values of x ranging from 0 to 3.5.

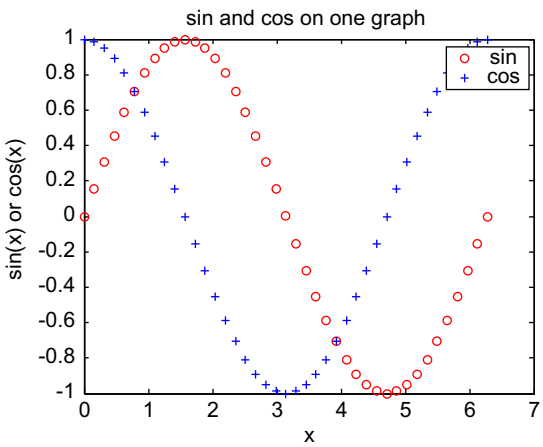


**FIGURE 3.5** Plot of **sin** and **cos** in one figure window with a legend

## 3.6 INTRODUCTION TO FILE INPUT/OUTPUT (LOAD AND SAVE)

In many cases, input to a script will come from a data file that has been created by another source. Also, it is useful to be able to store output in an external file that can be manipulated and/or printed later. In this section, the simplest methods used to read from an external data file and also to write to an external data file will be demonstrated.

There are basically three different operations, or ***modes*** on files. Files can be:

- read from
- written to
- appended to.

***Writing to a file*** means writing to a file from the beginning. ***Appending to a file*** is also writing, but starting at the end of the file rather than the

beginning. In other words, appending to a file means adding to what was already there.

There are many different file types, which use different filename extensions. For now, we will keep it simple and just work with .dat or .txt files when working with data, or text, files. There are several methods for reading from files and writing to files; we will, for now, use the **load** function to read and the **save** function to write to files. More file types and functions for manipulating them will be discussed in Chapter 9.

### 3.6.1 Writing Data to a File

The **save** command can be used to write data from a matrix to a data file, or to append to a data file. The format is:

```
save filename matrixvariablename –ascii
```

The "-ascii" qualifier is used when creating a text or data file. For example, the following creates a matrix and then saves the values from the matrix variable to a data file called "testfile.dat":

```
>> mymat = rand(2,3)
mymat =
    0.4565    0.8214    0.6154
    0.0185    0.4447    0.7919

>> save testfile.dat mymat -ascii
```

This creates a file called "testfile.dat" that stores the numbers:

```
 0.4565    0.8214    0.6154
 0.0185    0.4447    0.7919
```

The **type** command can be used to display the contents of the file; note that scientific notation is used:

```
>> type testfile.dat

  4.5646767e-001  8.2140716e-001  6.1543235e-001
  1.8503643e-002  4.4470336e-001  7.9193704e-001
```

Note that if the file already exists, the **save** command will overwrite the file; **save** always writes from the beginning of a file.

### 3.6.2 Appending Data to a Data File

Once a text file exists, data can be appended to it. The format is the same as the preceding, with the addition of the qualifier "-append". For example, the

following creates a new random matrix and appends it to the file that was just created:

```
>> mat2 = rand(3,3)
mymat =
    0.9218    0.4057    0.4103
    0.7382    0.9355    0.8936
    0.1763    0.9169    0.0579
>> save testfile.dat mat2 -ascii -append
```

This results in the file "testfile.dat" containing the following:

```
    0.4565    0.8214    0.6154
    0.0185    0.4447    0.7919
    0.9218    0.4057    0.4103
    0.7382    0.9355    0.8936
    0.1763    0.9169    0.0579
```

**Note**

Although technically any size matrix could be appended to this data file, to be able to read it back into a matrix later there would have to be the same number of values on every row (or, in other words, the same number of columns).

## PRACTICE 3.7

Prompt the user for the number of rows and columns of a matrix, create a matrix with that many rows and columns of random integers, and write it to a file.

### 3.6.3 Reading From a File

***Reading from a file*** is accomplished using **load**. Once a file has been created (as in the preceding), it can be read into a matrix variable. If the file is a data file, the **load** command will read from the file "filename.ext" (e.g., the extension might be .dat) and create a matrix with the same name as the file. For example, if the data file "testfile.dat" had been created as shown in the previous section, this would read from it, and store the result in a matrix variable called *testfile*:

```
>> clear
>> load testfile.dat
>> who
Your variables are:
testfile
>> testfile
testfile =
    0.4565    0.8214    0.6154
    0.0185    0.4447    0.7919
    0.9218    0.4057    0.4103
    0.7382    0.9355    0.8936
    0.1763    0.9169    0.0579
```

The **load** command works only if there are the same number of values in each line so that the data can be stored in a matrix, and the **save** command only writes from a matrix to a file. If this is not the case, lower-level file I/O functions must be used; these will be discussed in Chapter 9.

### 3.6.3.1 Example: Load from a File and Plot the Data

As an example, a file called "timetemp.dat" stores two lines of data. The first line is the time of day and the second line is the recorded temperature at each of those times. The first value of 0 for the time represents midnight. For example, the contents of the file might be:

```
0     3     6     9     12     15     18     21
55.5  52.4  52.6  55.7  75.6   77.7   70.3   66.6
```

The following script loads the data from the file into a matrix called *timetemp*. It then separates the matrix into vectors for the time and temperature, and then plots the data using black star (*) symbols.

timetempprob.m

```
% This reads time and temperature data for an afternoon
% from a file and plots the data

load timetemp.dat

% The times are in the first row, temps in the second row
time = timetemp(1,:);
temp = timetemp(2,:);

% Plot the data and label the plot
plot(time,temp,'k*')
xlabel('Time')
ylabel('Temperature')
title('Temperatures one afternoon')
```

Running the script produces the plot seen in Figure 3.6.

Note that it is difficult to see the point at time 0 as it falls on the y-axis. The **axis** function could be used to change the axes from the defaults shown here.

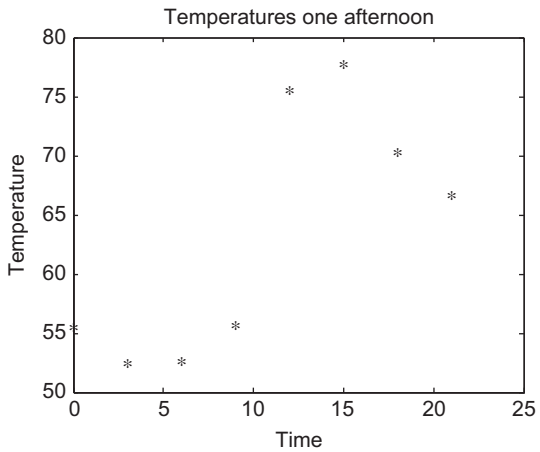


**FIGURE 3.6** Plot of temperature data from a file

To create the data file, the Editor in MATLAB can be used; it is not necessary to create a matrix and **save** it to a file. Instead, just enter the numbers in a new script file, and Save As "timetemp.dat", making sure that the Current Folder is set.

## PRACTICE 3.8

The sales (in billions) for two separate divisions of the ABC Corporation for each of the four quarters of 2013 are stored in a file called "salesfigs.dat":

```
1.2 1.4 1.8 1.3
2.2 2.5 1.7 2.9
```

- First, create this file (just type the numbers in the Editor, and Save As "salesfigs.dat").
- Then, write a script that will
  - load the data from the file into a matrix
  - separate this matrix into 2 vectors
  - create the plot seen in Figure 3.7 (which uses black circles and stars as the plot symbols).

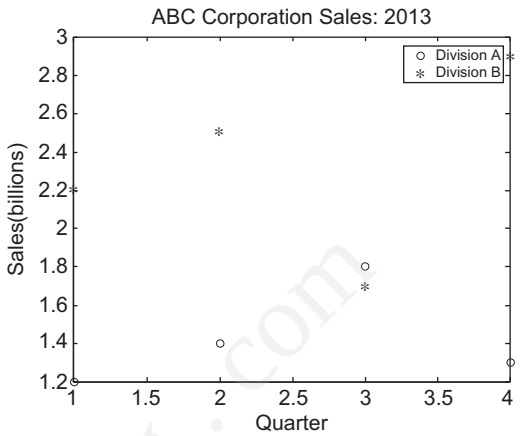


**FIGURE 3.7** Plot of sales data from file

## QUICK QUESTION!

Sometimes files are not in the format that is desired. For example, a file "expresults.dat" has been created that has some experimental results, but the order of the values is reversed in the file:

```
4   53.4
3   44.3
2   50.0
1   55.5
```

How could we create a new file that reverses the order?

### Answer

We can **load** from this file into a matrix, use the **flipud** function to "flip" the matrix up to down, and then **save** this matrix to a new file:

```
>> load expresults.dat
>> expresults
expresults =
    4.0000   53.4000
    3.0000   44.3000
    2.0000   50.0000
    1.0000   55.5000

>> correctorder = flipud(expresults)
correctorder =
    1.0000   55.5000
    2.0000   50.0000
    3.0000   44.3000
    4.0000   53.4000
>> save neworder.dat correctorder -ascii
```

# 3.7 USER-DEFINED FUNCTIONS THAT RETURN A SINGLE VALUE

We have already seen the use of many functions in MATLAB. We have used many built-in functions, such as **sin**, **fix**, **abs**, and **double**. In this section,

***user-defined functions*** will be introduced. These are functions that the programmer defines, and then uses, in either the Command Window or in a script.

There are several different types of functions. For now, we will concentrate on the kind of function that calculates and returns a single result. Other types of functions will be introduced in Chapter 6.

First, let us review some of what we already know about functions, including the use of built-in functions. Although, by now, the use of these functions is straightforward, explanations will be given in some detail here in order to compare and contrast to the use of user-defined functions.

The **length** function is an example of a built-in function that calculates a single value; it returns the length of a vector. As an example,

```
length(vec)
```

is an expression that represents the number of elements in the vector *vec*. This expression could be used in the Command Window or in a script. Typically, the value returned from this expression might be assigned to a variable:

```
>> vec = 1:3:10;
>> lv = length(vec)
lv =
     4
```

Alternatively, the length of the vector could be printed:

```
>> fprintf('The length of the vector is %d\n', length(vec))
The length of the vector is 4
```

The ***function call*** to the **length** function consists of the name of the function, followed by the ***argument*** in parentheses. The function receives as input the argument, and returns a result. What happens when the call to the function is encountered is that ***control*** is passed to the function itself (in other words, the function begins executing). The argument(s) are also passed to the function.

The function executes its statements and does whatever it does (the actual contents of the built-in functions are not generally known or seen by the user) to determine the number of elements in the vector. As the function is calculating a single value, this result is then ***returned*** and it becomes the value of the expression. Control is also passed back to the expression that called it in the first place, which then continues (e.g., in the first example the value would then be assigned to the variable *lv* and in the second example the value was printed).

### 3.7.1 Function Definitions

There are different ways to organize scripts and functions, but, for now, every function that we write will be stored in a separate M-file, which is why they are

commonly called "M-file functions". Although to type in functions in the Editor it is possible to choose the New down arrow and then Function, it will be easier for now to type in the function by choosing New Script (this ignores the defaults that are provided when you choose Function).

A function in MATLAB that returns a single result consists of the following.

- The ***function header*** (the first line), comprised of:
  - the reserved word **function**
  - the name of the output argument followed by the assignment operator (=), as the function ***returns*** a result
  - the name of the function (*important*—this should be the same as the name of the M-file in which this function is stored to avoid confusion)
  - the ***input arguments*** in parentheses, which correspond to the arguments that are passed to the function in the function call.
- A comment that describes what the function does (this is printed when **help** is used).
- The ***body*** of the function, which includes all statements and eventually must put a value in the output argument.
- **end** at the end of the function (note that this is not necessary in many cases in current versions of MATLAB, but it is considered good style anyway).

The general form of a ***function definition*** for a function that calculates and returns one value looks like this:

functionname.m

```
function outputargument = functionname(input arguments)
% Comment describing the function

Statements here; these must include putting a value in the output
argument

end % of the function
```

For example, the following is a function called *calcarea* that calculates and returns the area of a circle; it is stored in a file called *calcarea.m*.

calcarea.m

```
function area = calcarea(rad)
% calcarea calculates the area of a circle
% Format of call: calcarea(radius)
% Returns the area

area = pi * rad * rad;
end
```

A radius of a circle is passed to the function to the input argument *rad*; the function calculates the area of this circle and stores it in the output argument *area*.

In the function header, we have the reserved word **function**, then the output argument *area* followed by the assignment operator =, then the name of the function (the same as the name of the M-file), and then the input argument *rad*, which is the radius. As there is an output argument in the function header, somewhere in the body of the function we must put a value in this output argument. This is how a value is returned from the function. In this case, the function is simple and all we have to do is assign to the output argument *area* the value of the built-in constant **pi** multiplied by the square of the input argument *rad*.

**Note**
Many of the functions in MATLAB are implemented as M-file functions; these can also be displayed using **type**.

The function can be displayed in the Command Window using the **type** command.

```
>> type calcarea

        function area = calcarea(rad)
        % calcarea calculates the area of a circle
        % Format of call: calcarea(radius)
        % Returns the area

        area = pi * rad * rad;
        end
```

## 3.7.2 Calling a Function

The following is an example of a call to this function in which the value returned is stored in the default variable *ans*:

```
>> calcarea(4)
ans =
   50.2655
```

Technically, calling the function is done with the name of the file in which the function resides. To avoid confusion, it is easiest to give the function the same name as the filename, so that is how it will be presented in this book. In this example, the function name is *calcarea* and the name of the file is *calcarea.m*. The result returned from this function can also be stored in a variable in an assignment statement; the name could be the same as the name of the output argument in the function itself, but that is not necessary. So, for example, either of these assignments would be fine:

```
>> area = calcarea(5)
area =
   78.5398

>> myarea = calcarea(6)
myarea =
   113.0973
```

The output could also be suppressed when calling the function:

```
>> mya = calcarea(5.2);
```

The value returned from the *calcarea* function could also be printed using either **disp** or **fprintf**:

```
>> disp(calcarea(4))
   50.2655
>> fprintf('The area is %.1f\n', calcarea(4))
The area is 50.3
```

**Note**
The printing is not done in the function itself; rather, the function returns the area and then an output statement can print or display it.

## QUICK QUESTION!

Could we pass a vector of radii to the *calcarea* function?

### Answer

This function was written assuming that the argument was a scalar, so calling it with a vector instead would produce an error message:

```
>> calcarea(1:3)
Error using *
Inner matrix dimensions must agree.

Error in calcarea (line 6)
      area = pi * rad * rad;
```

This is because the * was used for multiplication in the function, but .* must be used when multiplying vectors term by term. Changing this in the function would allow either scalars or vectors to be passed to this function:

calcareaii.m

```
function area = calcareaii(rad)
% calcareaii returns the area of a circle
% The input argument can be a vector
% of radii
% Format: calcareaii(radiiVector)

area = pi * rad .* rad;
end
```

```
>> calcareaii(1:3)
ans =
    3.1416   12.5664   28.2743

>> calcareaii(4)
ans =
   50.2655
```

Note that the .* operator is only necessary when multiplying the radius vector by itself. Multiplying by **pi** is scalar multiplication, so the .* operator is not needed there. We could have also used:

```
area = pi * rad .^ 2;
```

Using **help** with either of these functions displays the contiguous block of comments under the function header (the block comment). It is useful to put the format of the call to the function in this block comment:

```
>> help calcarea
  calcarea calculates the area of a circle
  Format of call: calcarea(radius)
  Returns the area
```

Many organizations have standards regarding what information should be included in the block comment in a function. These can include:

- name of the function
- description of what the function does

- format of the function call
- description of input arguments
- description of output argument
- description of variables used in function
- programmer name and date written
- information on revisions.

Although this is excellent programming style, for the most part in this book these will be omitted simply to save space. Also, documentation in MATLAB suggests that the name of the function should be in all uppercase letters in the beginning of the block comment. However, this can be somewhat misleading in that MATLAB is case-sensitive and typically lowercase letters are used for the actual function name.

### 3.7.3 Calling a User-defined Function From a Script

Now, we will modify our script that prompts the user for the radius and calculates the area of a circle to call our function *calcarea* to calculate the area of the circle rather than doing this in the script.

circleCallFn.m

```
% This script calculates the area of a circle
% It prompts the user for the radius
radius = input('Please enter the radius: ');
% It then calls our function to calculate the
% area and then prints the result
area = calcarea(radius);
fprintf('For a circle with a radius of %.2f,',radius)
fprintf(' the area is %.2f\n',area)
```

Running this will produce the following:

```
>> circleCallFn
Please enter the radius: 5
For a circle with a radius of 5.00, the area is 78.54
```

#### *3.7.3.1 Simple Programs*

In this book, a script that calls function(s) is what we will call a MATLAB program. In the previous example, the program consisted of the script *circleCallFn* and the function it calls, *calcarea*. The general form of a simple program, consisting of a script that calls a function to calculate and return a value, looks like the diagram shown in Figure 3.8.

It is also possible for a function to call another (whether built-in or user-defined).

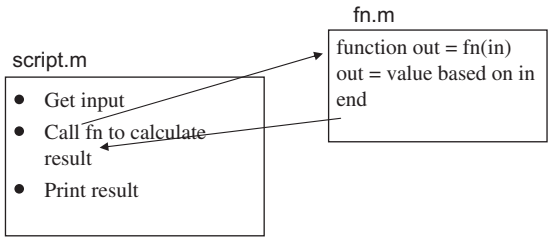


**FIGURE 3.8** General form of a simple program

### 3.7.4 Passing Multiple Arguments

In many cases it is necessary to pass more than one argument to a function. For example, the volume of a cone is given by

$$V = \frac{1}{3}\pi r^2 h$$

where r is the radius of the circular base and h is the height of the cone. Therefore, a function that calculates the volume of a cone needs both the radius and the height:

conevol.m

```
function outarg = conevol(radius, height)
% conevol calculates the volume of a cone
% Format of call: conevol(radius, height)
% Returns the volume

outarg = (pi/3) * radius .^ 2 .* height;
end
```

As the function has two input arguments in the function header, two values must be passed to the function when it is called. The order makes a difference. The first value that is passed to the function is stored in the first input argument (in this case, *radius*) and the second argument in the function call is passed to the second input argument in the function header.

This is very important: the arguments in the function call must correspond one-to-one with the input arguments in the function header.

Here is an example of calling this function. The result returned from the function is simply stored in the default variable *ans*.

```
>> conevol(4,6.1)
ans =
  102.2065
```

In the next example, the result is instead printed with a format of two decimal places.

```
>> fprintf('The cone volume is %.2f\n',conevol(3, 5.5))
The cone volume is 51.84
```

Note that by using the array exponentiation and multiplication operators, it would be possible to pass arrays for the input arguments, as long as the dimensions are the same.

## QUICK QUESTION!

Nothing is technically wrong with the following function, but what about it does not make sense?

fun.m

```
function out = fun(a,b,c)
out = a*b;
end
```

**Answer**

Why pass the third argument if it is not used?

## PRACTICE 3.9

Write a script that will prompt the user for the radius and height, call the function *conevol* to calculate the cone volume, and print the result in a nice sentence format. So, the program will consist of a script and the *conevol* function that it calls.

## PRACTICE 3.10

For a project, we need some material to form a rectangle. Write a function *calcrectarea* that will receive the length and width of a rectangle in inches as input arguments, and will return the area of the rectangle. For example, the function could be called as shown, in which the result is stored in a variable and then the amount of material required is printed, rounded up to the nearest square inch.

```
>> ra = calcrectarea(3.1, 4.4)
ra =
   13.6400

>> fprintf('We need %d sq in.\n', ...
       ceil(ra))
We need 14 sq in.
```

### 3.7.5 Functions with Local Variables

The functions discussed thus far have been very simple. However, in many cases the calculations in a function are more complicated, and may require the use of extra variables within the function; these are called ***local variables***.

For example, a closed cylinder is being constructed of a material that costs a certain dollar amount per square foot. We will write a function that will calculate and return the cost of the material, rounded up to the nearest square

foot, for a cylinder with a given radius and a given height. The total surface area for the closed cylinder is

$$SA = 2\pi rh + 2\pi r^2$$

For a cylinder with a radius of 32 inches, height of 73 inches, and cost per square foot of the material of $4.50, the calculation would be given by the following algorithm.

- Calculate the surface area SA = 2 * π * 32 * 73 + 2 * π * 32 * 32 inches squared.
- Convert the SA from square inches to square feet = SA/144.
- Calculate the total cost = SA in square feet * cost per square foot.

The function includes local variables to store the intermediate results.

cylcost.m

```
function outcost = cylcost(radius, height, cost)
% cylcost calculates the cost of constructing a closed
%   cylinder
% Format of call: cylcost(radius, height, cost)
% Returns the total cost

% The radius and height are in inches
% The cost is per square foot

% Calculate surface area in square inches
surf_area = 2 * pi * radius .* height + 2 * pi * radius .^ 2;

% Convert surface area in square feet and round up
surf_areasf = ceil(surf_area/144);

% Calculate cost
outcost = surf_areasf .* cost;
end
```

The following shows examples of calling the function:

```
>> cylcost(32,73,4.50)
ans =
  661.5000

>> fprintf('The cost would be $%.2f\n', cylcost(32,73,4.50))
The cost would be $661.50
```

### 3.7.6 Introduction to Scope

It is important to understand the ***scope of variables***, which is where they are valid. More will be described in Chapter 6, but, basically, variables used in a script are also known in the Command Window and vice versa. All variables used in a function, however, are local to that function. Both the Command Window and scripts use a common workspace, the ***base workspace***. Functions, however, have their own workspaces. This means that when a script is executed, the variables

can subsequently be seen in the Workspace Window and can be used from the Command Window. This is not the case with functions, however.

## 3.8 COMMANDS AND FUNCTIONS

Some of the commands that we have used (e.g., **format**, **type**, **save**, and **load**) are just shortcuts for function calls. If all of the arguments to be passed to a function are strings, and the function does not return any values, it can be used as a command. For example, the following produce the same results:

```
>> type script1

radius = 5
area = pi * (radius^2)

>> type('script1')

radius = 5
area = pi * (radius^2)
```

Using **load** as a command creates a variable with the same name as the file. If a different variable name is desired, it is easiest to use the functional form of **load**. For example,

```
>> type pointcoords.dat

3.3    1.2
4      5.3

>> points = load('pointcoords.dat')
points =
    3.3000    1.2000
    4.0000    5.3000
```

### Explore Other Interesting Features

Note that this chapter serves as an introduction to several topics, most of which will be covered in more detail in future chapters. Before getting to those chapters, the following are some things you may wish to explore.

- The **help** command can be used to see short explanations of built-in functions. At the end of this, a doc page link is also listed. These documentation pages frequently have much more information and useful examples. They can also be reached by typing "doc fnname", where fnname is the name of the function.
- Look at formatSpec on the doc page on the **fprintf** function for more ways in which expressions can be formatted (e.g., padding numbers with zeros and printing the sign of a number).
- Use the Search Documentation to find the conversion characters used to print other types, such as unsigned integers and exponential notation. ■

## Summary

**Common Pitfalls**

- Spelling a variable name different ways in different places in a script or function.
- Forgetting to add the second 's' argument to the **input** function when character input is desired.
- Not using the correct conversion character when printing.
- Confusing **fprintf** and **disp.** Remember that only **fprintf** can format.

**Programming Style Guidelines**

- Especially for longer scripts and functions, start by writing an algorithm.
- Use comments to document scripts and functions, as follows:
  - a block of contiguous comments at the top to describe a script
  - a block of contiguous comments under the function header for functions
  - comments throughout any M-file (script or function) to describe each section.
- Make sure that the "H1" comment line has useful information.
- Use your organization's standard style guidelines for block comments.
- Use mnemonic identifier names (names that make sense, e.g., *radius* instead of *xyz*) for variable names and for file names.
- Make all output easy to read and informative.
- Put a newline character at the end of every string printed by **fprintf** so that the next output or the prompt appears on the line below.
- Put informative labels on the x and y axes, and a title on all plots.
- Keep functions short — typically no longer than one page in length.
- Suppress the output from all assignment statements in functions and scripts.
- Functions that return a value do not normally print the value; it should simply be returned by the function.
- Use the array operators .*, ./, .\, and .^ in functions so that the input arguments can be arrays and not just scalars. ■

| MATLAB Reserved Words | |
|---|---|
| function | end |

| MATLAB Functions and Commands | | | |
|---|---|---|---|
| type | xlabel | clf | grid |
| input | ylabel | figure | bar |
| disp | title | hold | load |
| fprintf | axis | legend | save |
| plot | | | |