CHAPTER 3

Introduction to MATLAB Programming

KEY TERMS

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

MATLAB®. http://dx.doi.org/10.1016/B978-0-12-804525-1.00003-9

the concept. Input will be introduced, both from files and from the user. Output 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.

As of Version R2016a, there are two types of scripts. A new, richer script type called a ***live script*** has been created in MATLAB. Live scripts will be introduced in Chapter 6, in which programming concepts will be covered in more depth. In this chapter, we will create simple scripts stored in MATLAB code files, which have an extension of .m.

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 using 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 includes 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 π multiplied by the square of the radius. So, that means the value of the constant for π 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 three steps that have been outlined below:

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

As can be seen here, even the simplest solution to a problem 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 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 MATLAB code files, which have an extension .m on the file name. These script files are interpreted, rather than compiled. Therefore, the correct terminology is that these are scripts, and not programs. However, the terms are used somewhat loosely by many people, and 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 a file with an extension of .m 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 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. The easiest method is to click on "New Script" under the HOME tab. Alternatively, one can click on the down arrow under "New" and then choose Script (see Fig. 3.1)

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. Make sure that the extension of .m is on the file name (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.)

If you have entered commands in the Command Window, and decide that you would like to put them into a script, an alternate method for creating a script is

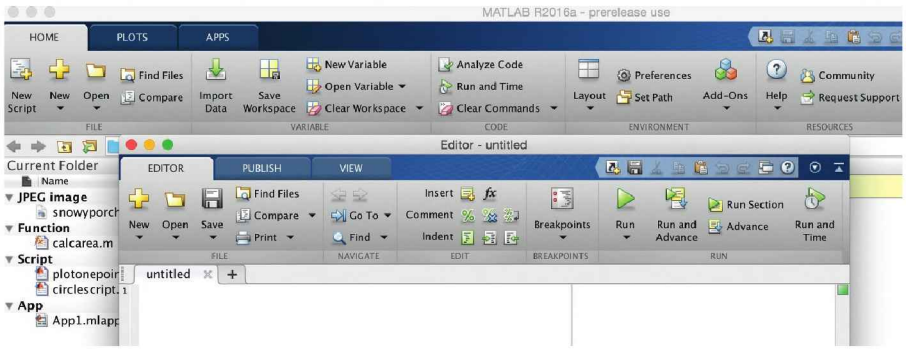

FIGURE 3.1
Toolstrip and Editor.

to select the commands in the Command History window, and then right click. This will give options for creating a script or live script, and will then prepopulate the editor with those commands.

In our first 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.

In this text, scripts will be displayed in a box with the name of the 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 used for 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 hits 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
```

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

```
>> length(mychar)
 ans =
      7
```

The string can actually 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.

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 '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
```

It is also possible to enter a vector. The user can enter any valid vector, using any valid syntax such as square brackets, the colon operator, or functions such as **linspace**.

```
>> v = input('Enter a vector: ')
Enter a vector: [3   8   22]
v =
     3     8     22
```

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 case, 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    character (one character)
%s    string of characters
```

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, the newline 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 in 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 be specified; for example, %.3f indicates 3 decimal places, regardless of the field width.

```
>> 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
```

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

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.

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 significantly.

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!

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 %.2 s\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 2×3 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 straight-forward 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 on two- and three-dimensional 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 am) 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 Fig. 3.2).

In 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])
```

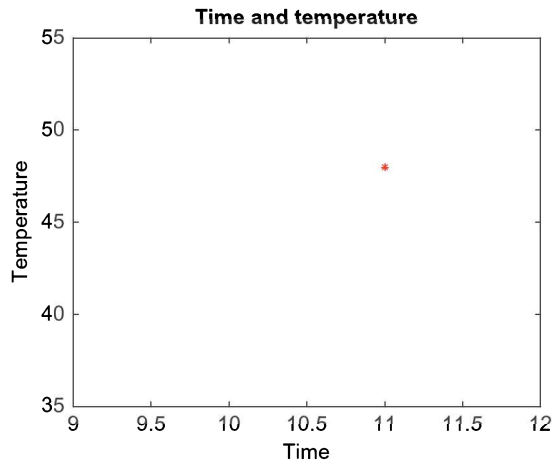

FIGURE 3.2
Plot of one data point.

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.

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 Fig. 3.3).

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

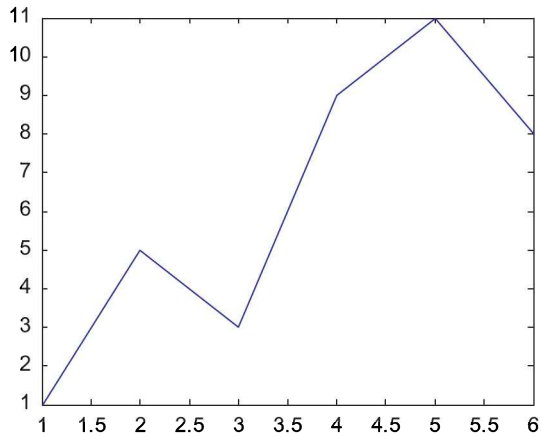

FIGURE 3.3
Plot of data points from vectors.

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. There are many options for the **axis** function; for example, just calling it with no arguments returns the values used for the x and y-axes ranges.

```
>> arang = axis
arang =
     1   6   1  11
```

Axes can also be turned on and off, and they can be made square or equal to each other. A subset of the data can also be shown by limiting the extent of the axes.

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.

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, at 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
g green
r red
c cyan
m magenta
y yellow
k black
w white
```

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:

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

Line types can also be specified by the following:

```
-      solid
:      dotted
-.     dash dot
--     dashed
(none) no line
```

If no line type is specified and no marker 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 12, 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 numbered "Figure 1" in MATLAB. The second will be "Figure 2." See Fig. 3.4 for both plots.

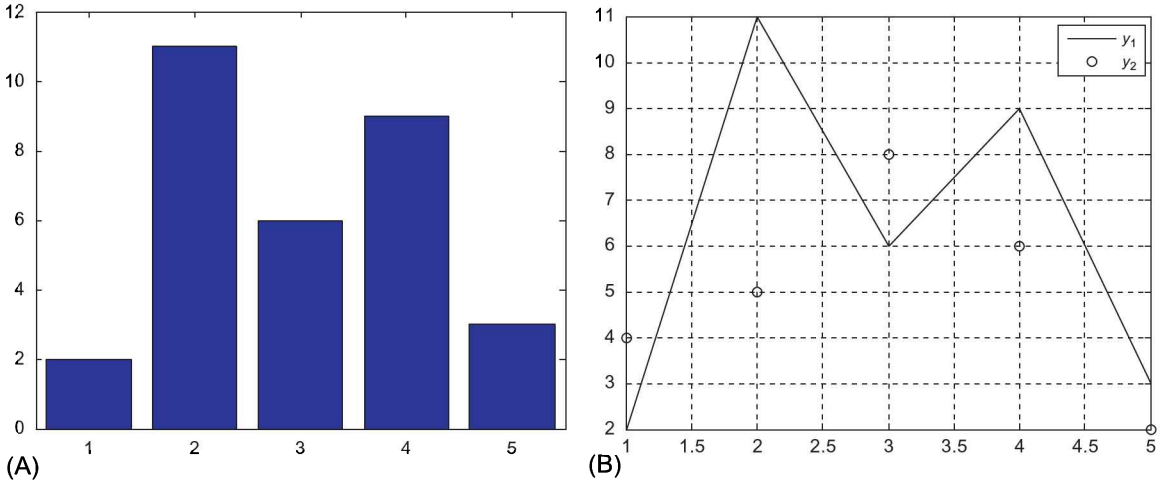

FIGURE 3.4
(A) Bar chart produced by script. (B) Plot produced by script, with a grid and legend.

Note that the first and last points are on the axes, which makes them difficult to be seen. 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 graphically displays 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')
```

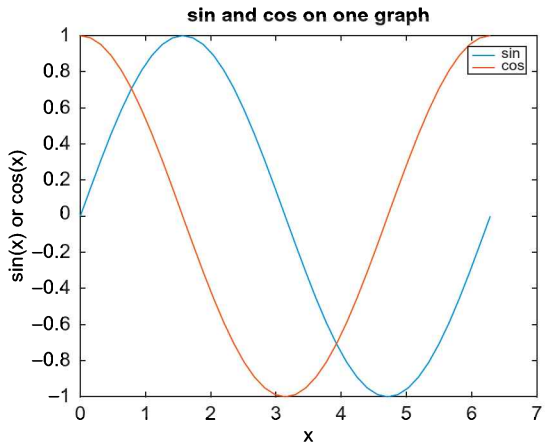

FIGURE 3.5
Plot of **sin** and **cos** in one Figure Window with a legend.

The script creates an x vector; iterating through all of the values from 0 to $2*\pi$ in steps of $2*\pi/40$ gives enough points to get a good graph. It then finds the sine of each x value, and plots these. 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. 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 Fig. 3.5.

Beginning with Version R2014b, when **hold on** is used, MATLAB uses a sequence of colors for the plots, rather than using the default color for each. Of course, colors can also be specified as was done in this script.

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

```
plot(x,sin(x),x,cos(x))
```

PRACTICE 3.6

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

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 is created, 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 times 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 Fig. 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.

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.

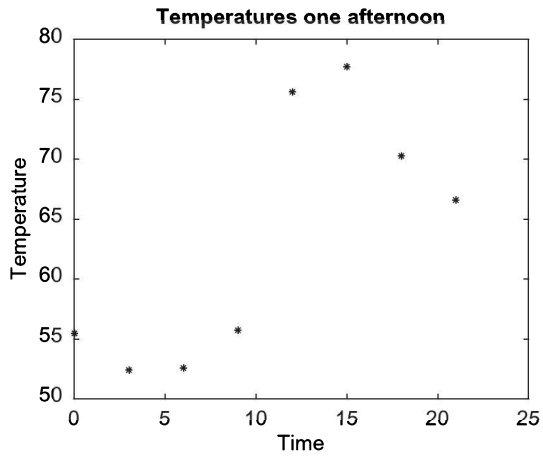

FIGURE 3.6
Plot of temperature data from a file.

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 Fig. 3.7 (which uses black circles and stars as the plot symbols).

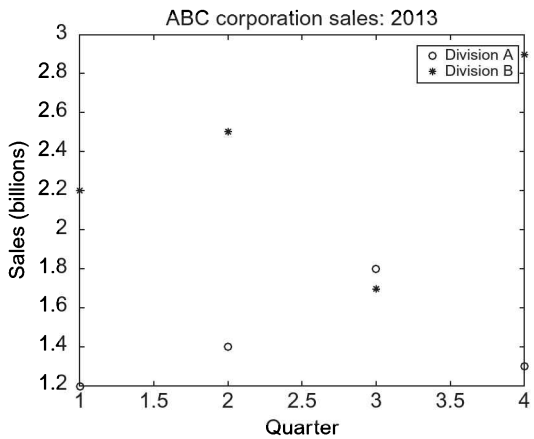

FIGURE 3.7
Plot of sales data from file.

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 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 is necessary (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 file. Like scripts, function files have an extension of .m. Although to enter function definitions 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 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 a 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
```

The 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 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*.

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
```

Note
Many of the functions in MATLAB are implemented as functions that are stored in files with an extension of .m; these can also be displayed using **type**.

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 file name, 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
```

The suggested corrections for invalid filenames in the Command Window works for user-defined files as of Version R2014b.

```
>> clacarea(3)
Undefined function or variable 'clacarea'.
Did you mean:
>> calcarea(3)
```

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 an excellent programming style, for the most part of 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*. A simple program, consisting of a script that calls a function to calculate and return a value, looks like the format shown in Fig. 3.8.

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

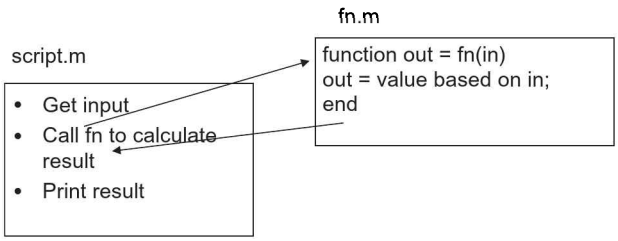

FIGURE 3.8
General form of a simple program.

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?

Answer; Why pass the third argument if it is not used?

fun.m

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

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 in., height of 73 in., 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 in^2.
- 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 easiesr 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
```

This stores the result in a variable *points* rather than *pointcoords*.

■ 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, for example, 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 code 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.

function	end

MATLAB Functions and Commands

type	xlabel	figure	load
input	ylabel	hold	save
disp	title	legend	
fprintf	axis	grid	
plot	clf	bar	

comment %	comment block %{, %}

Exercises

1. Using the top-down design approach, write an algorithm for making a sandwich.
2. Write a simple script that will calculate the volume of a hollow sphere,

   $$\frac{4\pi}{3}(r_o^3 - r_i^3)$$

   where r_i is the inner radius and r_o is the outer radius. Assign a value to a variable for the inner radius, and also assign a value to another variable for the outer radius. Then, using these variables, assign the volume to a third variable. Include comments in the script. Use **help** to view the comments in your script.
3. Write a statement that prompts the user for his/her favorite number.

4. Write a statement that prompts the user for his/her name.
5. Write an **input** statement that will prompt the user for a real number, and store it in a variable. Then, use the **fprintf** function to print the value of this variable using 2 decimal places.
6. Experiment, in the Command Window, with the **fprintf** function for real numbers. Make a note of what happens for each. Use **fprintf** to print the real number 12345.6789.
   without specifying any field width
   in a field width of 10 with 4 decimal places
   in a field width of 10 with 2 decimal places
   in a field width of 6 with 4 decimal places
   in a field width of 2 with 4 decimal places
7. Experiment, in the Command Window, with the **fprintf** function for integers. Make a note of what happens for each. Use **fprintf** to print the integer 12345.
   without specifying any field width
   in a field width of 5
   in a field width of 8
   in a field width of 3
8. When would you use **disp** instead of **fprintf**? When would you use **fprintf** instead of **disp**?
9. Write a script called *echostring* that will prompt the user for a string, and will echo print the string in quotes:

```
>> echostring
Enter your string: hi there
Your string was: 'hi there'
```

10. If the lengths of two sides of a triangle and the angle between them are known, the length of the third side can be calculated. Given the lengths of two sides (b and c) of a triangle, and the angle between them α in degrees, the third side a is calculated as follows:

$$a^2 = b^2 + c^2 - 2\,b\,c\,\cos(\alpha)$$

Write a script *thirdside* that will prompt the user and read in values for b, c, and α (in degrees), and then calculate and print the value of a, with 3 decimal places. The format of the output from the script should look exactly like this:

```
>> thirdside
Enter the first side: 2.2
Enter the second side: 4.4
Enter the angle between them: 50

The third side is 3.429
```

For more practice, write a function to calculate the third side, so the script will call this function.

11. Write a script that will prompt the user for a character, and will print it twice; once left-justified in a field width of 5, and again right-justified in a field width of 3.
12. Write a script *lumin* that will calculate and print the luminosity *L* of a star in Watts. The luminosity *L* is given by $L = 4\pi d^2\ b$ where *d* is the distance from the sun in meters and *b* is the brightness in Watts/meters^2. Here is an example of executing the script:

```
>> lumin
This script will calculate the luminosity of a star.
When prompted, enter the star's distance from the sun
  in meters, and its brightness in W/meters squared.

Enter the distance: 1.26e12
Enter the brightness: 2e-17
The luminosity of this star is 399007399.75 watts
```

13. A script "iotrace" has been written. Here's what the desired output looks like:

```
>> iotrace
Please enter a number: 33
Please enter a character: x
Your number is 33.00
Your char is       x!
```

Fix this script so that it works as shown previously:

```
mynum = input('Please enter a number:\n ');
mychar = input('Please enter a character: ');
fprintf('Your number is %.2f, mynum)
fprintf('Your char is %c!\n', mychar)
```

14. Write a script that assigns values for the *x* coordinate and then *y* coordinate of a point, and then plot this using a green +.
15. Plot **sin(x)** for *x* values ranging from 0 to π (in separate Figure Windows):
    using 10 points in this range
    using 100 points in this range
16. When would it be important to use **legend** in a plot?
17. Why do we always suppress all assignment statements in scripts?
18. Atmospheric properties such as temperature, air density, and air pressure are important in aviation. Create a file that stores temperatures in degrees Kelvin at various altitudes. The altitudes are in the first column and the temperatures in the second. For example, it may look like this:

```
1000    288
2000    281
3000    269
```

19. Generate a random integer *n*, create a vector of the integers 1 through *n* in steps of 2, square them, and plot the squares.

20. Create a 3×6 matrix of random integers, each in the range of 50–100. Write this to a file called *randfile.dat*. Then, create a new matrix of random integers, but this time make it a 2×6 matrix of random integers, each in the range of 50–100. Append this matrix to the original file. Then, read the file (which will be to a variable called *randfile*) just to make sure that it worked!
21. A particular part is being turned on a lathe. The diameter of the part is supposed to be 20,000 mm. The diameter is measured every 10 min and the results are stored in a file called *partdiam.dat*. Create a data file to simulate this. The file will store the time in minutes and the diameter at each time. Plot the data.
22. Create a file called "testtan.dat" comprised of two lines with three real numbers on each line (some negative, some positive, in the—1 to 3 range). The file can be created from the Editor, or saved from a matrix. Then, **load** the file into a matrix and calculate the tangent of every element in the resulting matrix.
23. Write a function *calcrectarea* that will calculate and return the area of a rectangle. Pass the length and width to the function as input arguments.

Renewable energy sources such as biomass are gaining increasing attention. Biomass energy units include megawatt hours (MWh) and gigajoules (GJ). One MWh is equivalent to 3.6 GJ. For example, one cubic meter of wood chips produces 1 MWh.

24. Write a function *mwh_to_gj* that will convert MWh to GJ.
25. List some differences between a script and a function.
26. In quantum mechanics, the angular wavelength for a wavelength λ is defined as $\lambda/2\pi$. Write a function named *makeitangular* that will receive the wavelength as an input argument, and will return the angular wavelength.
27. Write a *fives* function that will receive two arguments for the number of rows and columns, and will return a matrix with that size of all fives.
28. Write a function *isdivby4* that will receive an integer input argument, and will return **logical** 1 for **true** if the input argument is divisible by 4, or **logical false** if it is not.
29. Write a function *isint* that will receive a number input argument *innum*, and will return 1 for **true** if this number is an integer, or 0 for **false** if not. Use the fact that *innum* should be equal to **int32(innum)** if it is an integer. Unfortunately, due to round-off errors, it should be noted that it is possible to get **logical** 1 for **true** if the input argument is close to an integer. Therefore the output may not be what you might expect, as shown here.

```
>> isint(4)
ans =
       1
>> isint(4.9999)
ans =
       0
>> isint(4.99999999999999999999999999)
ans =
       1
```

30. A Pythagorean triple is a set of positive integers (a,b,c) such that $a^2+b^2=c^2$. Write a function *ispythag* that will receive three positive integers (a, b, c in that order) and will return **logical** 1 for **true** if they form a Pythagorean triple, or 0 for **false** if not.
31. A function can return a vector as a result. Write a function *vecout* that will receive one integer argument and will return a vector that increments from the value of the input argument to its value plus 5, using the colon operator. For example,

```
>> vecout(4)
ans =
     4   5   6   7   8   9
```

32. Write a function that is called *pickone*, which will receive one input argument *x*, which is a vector, and will return one random element from the vector. For example,

```
>> pickone(4:7)
ans =
     5
>> disp(pickone(-2:0))
-1
>> help pickone
pickone(x) returns a random element from vector x
```

33. The conversion depends on the temperature and other factors, but an approximation is that 1 in. of rain is equivalent to 6.5 in. of snow. Write a script that prompts the user for the number of inches of rain, calls a function to return the equivalent amount of snow, and prints this result. Write the function, as well!
34. In thermodynamics, the Carnot efficiency is the maximum possible efficiency of a heat engine operating between two reservoirs at different temperatures. The Carnot efficiency is given as

$$\eta = 1 - \frac{T_C}{T_H}$$

where T_C and T_H are the absolute temperatures at the cold and hot reservoirs, respectively. Write a script *carnot* that will prompt the user for the two reservoir temperatures in Kelvin, call a function to calculate the Carnot efficiency, and then print the corresponding Carnot efficiency to 3 decimal places. Also write the function.
35. Many mathematical models in engineering use the exponential function. The general form of the exponential decay function is:

$$y(t) = Ae^{-\tau t}$$

where A is the initial value at $t = 0$, and τ is the time constant for the function. Write a script to study the effect of the time constant. To simplify the equation, set A equal to 1. Prompt the user for two different values for the time constant, and for beginning and ending values for the range of a t vector. Then, calculate two different y vectors using the above equation and the two time constants, and graph both exponential functions on the same graph within the range the user specified. Use a function to calculate y. Make one plot red. Be sure to label the graph and both axes. What happens to the decay rate as the time constant gets larger?

This page intentionally left blank

CHAPTER 4

Selection Statements

KEY TERMS

selection statements
branching statements
condition
action
temporary variable
error-checking
throwing an error
nesting statements
cascading if-else
"is" functions

CONTENTS

In the scripts and functions we've seen thus far, every statement was executed in sequence. This is not always desirable, and in this chapter we'll see how to make choices as to whether statements are executed or not, and how to choose between or among statements. The statements that accomplish this are called ***selection*** or ***branching*** statements.

The MATLAB® software has two basic statements that allow us to make choices: the **if** statement and the **switch** statement. The **if** statement has optional **else** and **elseif** clauses for branching. The **if** statement uses expressions that are logically **true** or **false**. These expressions use relational and logical operators. MATLAB also has "is" functions that test whether an attribute is **true** or not; these can be used with the selection statements.

4.1 THE IF STATEMENT

The **if** statement chooses whether another statement, or group of statements, is executed or not. The general form of the **if** statement is:

```
if condition
    action
end
```

A ***condition*** is a relational expression that is conceptually, or logically, **true** or **false**. The ***action*** is a statement, or a group of statements, that will be executed if the condition is **true**. When the **if** statement is executed, first the condition is

MATLAB®. http://dx.doi.org/10.1016/B978-0-12-804525-1.00004-0

evaluated. If the value of the condition is **true**, the action will be executed; if not, the action will not be executed. The action can be any number of statements until the reserved word **end**; the action is naturally bracketed by the reserved words **if** and **end**. (Note that this is different from the **end** that is used as an index into a vector or matrix.) The action is usually indented to make it easier to see.

For example, the following **if** statement checks to see whether the value of a variable is negative. If it is, the value is changed to a zero; otherwise, nothing is changed.

```
if num < 0
     num = 0
end
```

If statements can be entered in the Command Window, although they generally make more sense in scripts or functions. In the Command Window, the **if** line would be entered, followed by the Enter key, the action, the Enter key, and finally **end** and Enter. The results will follow immediately. For example, the preceding **if** statement is shown twice here.

```
>> num = -4;
>> if num < 0
       num = 0
   end
num =
     0

>> num = 5;
>> if num < 0
       num = 0
  end
>>
```

Note that the output from the assignment is not suppressed, so the result of the action will be shown if the action is executed. The first time, the value of the variable is negative so the action is executed and the variable is modified, but, in the second case, the variable is positive so the action is skipped.

This may be used, for example, to make sure that the square root function is not used on a negative number. The following script prompts the user for a number and prints the square root. If the user enters a negative number the **if** statement changes it to zero before taking the square root.

sqrtifexamp.m

```
% Prompt the user for a number and print its sqrt

num = input('Please enter a number: ');

% If the user entered a negative number, change it
if num < 0
    num = 0;
end
fprintf('The sqrt of %.1f is %.1f\n',num,sqrt(num))
```

Here are two examples of running this script:

```
>> sqrtifexamp
Please enter a number: -4.2
The sqrt of 0.0 is 0.0

>> sqrtifexamp
Please enter a number: 1.44
The sqrt of 1.4 is 1.2
```

Note that in the script the output from the assignment statement is suppressed. In this case, the action of the **if** statement was a single assignment statement. The action can be any number of valid statements. For example, we may wish to print a note to the user to say that the number entered was being changed. Also, instead of changing it to zero we will use the absolute value of the negative number entered by the user.

sqrtifexampii.m

```
% Prompt the user for a number and print its sqrt

num = input('Please enter a number: ');

% If the user entered a negative number, tell
% the user and change it
if num < 0
    disp('OK, we''ll use the absolute value')
    num = abs(num);
end
fprintf('The sqrt of %.1f is %.1f\n',num,sqrt(num))
```

```
>> sqrtifexampii
Please enter a number: -25
OK, we'll use the absolute value
The sqrt of 25.0 is 5.0
```

Note that as seen in this example, two single quotes in the **disp** statement are used to print one single quote.

PRACTICE 4.1

Write an **if** statement that would print "Hey, you get overtime!" if the value of a variable *hours* is greater than 40. Test the **if** statement for values of *hours* less than, equal to, and greater than 40. Will it be easier to do this in the Command Window or in a script?

QUICK QUESTION!

Assume that we want to create a vector of increasing integer values from *mymin* to *mymax*. We will write a function *createvec* that receives two input arguments, *mymin* and *mymax*, and returns a vector with values from *mymin* to *mymax* in steps of one. First, we would make sure that the value of *mymin* is less than the value of *mymax*. If not, we would need to exchange their values before creating the vector. How would we accomplish this?

Answer: To exchange values, a third variable, a temporary variable, is required. For example, let's say that we have two variables, *a* and *b*, storing the values:

```
a = 3;
b = 5;
```

To exchange values, we could *not* just assign the value of *b* to *a*, as follows:

```
a = b;
```

If that were done, then the value of *a* (the 3), is lost! Instead, we need to assign the value of *a* first to a ***temporary variable*** so that the value is not lost. The algorithm would be:

- assign the value of *a* to *temp*
- assign the value of *b* to *a*
- assign the value of *temp* to *b*

```
>> temp = a;
>> a = b
a =
     5
>> b = temp
b =
     3
```

Now, for the function. An **if** statement is used to determine whether or not the exchange is necessary.

createvec.m

```
function outvec = createvec(mymin, mymax)
% createvec creates a vector that iterates from a
%  specified minimum to a maximum
% Format of call: createvec(minimum, maximum)
% Returns a vector

%If the "minimum" isn't smaller than the "maximum",
% exchange the values using a temporary variable
if mymin > mymax
    temp = mymin;
    mymin = mymax;
    mymax = temp;
end

% Use the colon operator to create the vector
outvec = mymin:mymax;
end
```

Examples of calling the function are:

```
>> createvec(4,6)
ans =
     4   5   6

>> createvec(7,3)
ans =
     3   4   5   6   7
```

4.1.1 Representing Logical True and False

It has been stated that conceptually true expressions have the **logical** value of 1, and expressions that are conceptually false have the **logical** value of 0. Representing the concepts of **logical true** and **false** in MATLAB is slightly different: the concept of false is represented by the value of 0, but the concept of true can be represented by *any nonzero value* (not just 1). This can lead to some strange **logical** expressions. For example:

```
>> all(1:3)
ans =
     1
```

Also, consider the following **if** statement:

```
>> if 5
       disp('Yes, this is true!')
   end
Yes, this is true!
```

As 5 is a nonzero value, the condition is **true**. Therefore, when this **logical** expression is evaluated, it will be **true**, so the **disp** function will be executed and "Yes, this is true" is displayed. Of course, this is a pretty bizarre **if** statement, one that hopefully would never be encountered!

However, a simple mistake in an expression can lead to a similar result. For example, let's say that the user is prompted for a choice of 'Y' or 'N' for a yes/no question.

```
letter = input('Choice (Y/N): ','s');
```

In a script we might want to execute a particular action if the user responded with 'Y.' Most scripts would allow the user to enter either lowercase or uppercase; for example, either 'y' or 'Y' to indicate "yes." The proper expression that would return **true** if the value of *letter* was 'y' or 'Y' would be

```
letter == 'y' || letter == 'Y'
```

However, if by mistake this was written as:

```
letter == 'y' || 'Y'          %Note: incorrect!!
```

this expression would ALWAYS be **true**, regardless of the value of the variable *letter*. This is because `'Y'` is a nonzero value, so it is a **true** expression. The first part of the expression may be **false**, but as the second expression is **true** the entire expression would be **true**, regardless of the value of the variable *letter*.

4.2 THE IF-ELSE STATEMENT

The **if** statement chooses whether or not an action is executed. Choosing between two actions, or choosing from among several actions, is accomplished using **if-else**, nested **if-else**, and **switch** statements.

The **if-else** statement is used to choose between two statements, or sets of statements. The general form is:

```
if condition
    action1
else
    action2
end
```

First, the condition is evaluated. If it is **true**, then the set of statements designated as "action1" is executed, and that is the end of the **if-else** statement. If, instead, the condition is **false**, the second set of statements designated as "action2" is executed, and that is the end of the **if-else** statement. The first set of statements ("action1") is called the action of the **if** clause; it is what will be executed if the expression is **true**. The second set of statements ("action2") is called the action of the **else** clause; it is what will be executed if the expression is **false**. One of these actions, and only one, will be executed—which one depends on the value of the condition.

For example, to determine and print whether or not a random number in the range from 0 to 1 is less than 0.5, an **if-else** statement could be used:

```
if rand < 0.5
   disp('It was less than .5!')
else
   disp('It was not less than .5!')
end
```

PRACTICE 4.2

Write a script *printsindegorrad* that will:

- prompt the user for an angle
- prompt the user for (r)adians or (d)egrees, with radians as the default
- if the user enters 'd,' the **sind** function will be used to get the sine of the angle in degrees; otherwise, the **sin** function will be used. Which sine function to use will be based solely on whether the user entered a 'd' or not ('d' means degrees, so **sind** is used; otherwise, for any other character the default of radians is assumed so **sin** is used)
- print the result.

Here are examples of running the script:

```
>> printsindegorrad
Enter the angle: 45
(r)adians (the default) or (d)egrees: d
The sin is 0.71

>> printsindegorrad
Enter the angle: pi
(r)adians (the default) or (d)egrees: r
The sin is 0.00
```

One application of an **if-else** statement is to check for errors in the inputs to a script (this is called ***error-checking***). For example, an earlier script prompted the user for a radius, and then used that to calculate the area of a circle. However, it did not check to make sure that the radius was valid (e.g., a positive number). Here is a modified script that checks the radius:

checkradius.m

```
% This script calculates the area of a circle
% It error-checks the user's radius
radius = input('Please enter the radius: ');
if radius <= 0
    fprintf('Sorry; %.2f is not a valid radius\n',radius)
else
    area = calcarea(radius);
    fprintf('For a circle with a radius of %.2f,',radius)
    fprintf(' the area is %.2f\n',area)
end
```

Examples of running this script when the user enters invalid and then valid radii are shown as follows:

```
>> checkradius
Please enter the radius: -4
Sorry; -4.00 is not a valid radius

>> checkradius
Please enter the radius: 5.5
For a circle with a radius of 5.50, the area is 95.03
```

The **if-else** statement in this example chooses between two actions: printing an error message, or using the radius to calculate the area and then printing out the result. Note that the action of the **if** clause is a single statement, whereas the action of the **else** clause is a group of three statements.

MATLAB also has an **error** function that can be used to display an error message; the terminology is that this is ***throwing an error***. In the previous script, the if clause could be modified to use the **error** function rather than **fprintf**; the result will be displayed in red as with the error messages generated by MATLAB.

```
>> if radius <= 0
   error('Sorry; %.2f is not a valid radius\n', radius)
end

Sorry; -4.00 is not a valid radius
```

4.3 NESTED IF-ELSE STATEMENTS

The **if-else** statement is used to choose between two actions. To choose from more than two actions the **if-else** statements can be ***nested***, meaning one

statement inside of another. For example, consider implementing the following continuous mathematical function $y=f(x)$:

$y = 1$ if $x < -1$
$y = x^2$ if $-1 \leq x \leq 2$
$y = 4$ if $x > 2$

The value of y is based on the value of x, which could be in one of three possible ranges. Choosing which range could be accomplished with three separate **if** statements, is as follows:

```
if x < -1
   y = 1;
end
if x >= -1 && x <=2
   y = x^2;
end
if x > 2
   y = 4;
end
```

Note that the && in the expression of the second if statement is necessary. Writing the expression as $-1<=x<=2$ would be incorrect; recall from Chapter 1 that that expression would always be **true**, regardless of the value of the variable x.

As the three possibilities are mutually exclusive, the value of y can be determined by using three separate **if** statements. However, this is not a very efficient code: all three **logical** expressions must be evaluated, regardless of the range in which x falls. For example, if x is less than -1, the first expression is **true** and 1 would be assigned to y. However, the two expressions in the next two **if** statements are still evaluated. Instead of writing it this way, the statements can be nested so that the entire **if-else** statement ends when an expression is found to be **true**:

```
if x < -1
   y = 1;
else
   % If we are here, x must be >= -1
   % Use an if-else statement to choose
   %  between the two remaining ranges
   if  x <= 2
      y = x^2;
   else
      % No need to check
      % If we are here, x must be > 2
      y = 4;
   end
end
```

By using a nested **if-else** to choose from among the three possibilities, not all conditions must be tested as they were in the previous example. In this case, if x is less than -1, the statement to assign 1 to y is executed, and the **if-else** statement is completed so no other conditions are tested. If, however, x is not less than -1, then the **else** clause is executed. If the **else** clause is executed, then we already know that x is greater than or equal to -1 so that part does not need to be tested.

Instead, there are only two remaining possibilities: either x is less than or equal to 2, or it is greater than 2. An **if-else** statement is used to choose between those two possibilities. So, the action of the **else** clause was another **if-else** statement. Although it is long, all of the above code is one **if-else** statement, a nested **if-else** statement. The actions are indented to show the structure of the statement. Nesting **if-else** statements in this way can be used to choose from among 3, 4, 5, 6, ... the possibilities are practically endless!

This is actually an example of a particular kind of nested **if-else** called a ***cascading* if-else** statement. This is a type of nested **if-else** statement in which the conditions and actions cascade in a stair-like pattern.

Not all nested **if-else** statements are cascading. For example, consider the following (which assumes that a variable x has been initialized):

```
if x >= 0
   if x < 4
      disp('a')
   else
      disp('b')
   end
else
   disp('c')
end
```

4.3.1 The elseif Clause

THE PROGRAMMING CONCEPT

In some programming languages, choosing from multiple options means using nested **if-else** statements. However, MATLAB has another method of accomplishing this using the **elseif** clause.

THE EFFICIENT METHOD

To choose from among more than two actions, the **elseif** clause is used. For example, if there are n choices (where $n > 3$ in this example), the following general form would be used:

Continued

THE EFFICIENT METHOD—CONT'D

```
if condition1
    action1
elseif condition2
    action2
elseif condition3
     action3
% etc: there can be many of these
else
    actionn     % the nth action
end
```

The actions of the **if**, **elseif**, and **else** clauses are naturally bracketed by the reserved words **if**, **elseif**, **else**, and **end**.

For example, the previous example could be written using the **elseif** clause, rather than nesting **if-else** statements:

```
if x < -1
   y = 1;
elseif x <= 2
   y = x^2;
else
   y = 4;
end
```

Note that in this example we only need one **end**. So, there are three ways of accomplishing the original task: using three separate **if** statements, using nested **if-else** statements, and using an **if** statement with **elseif** clauses, which is the simplest.

This could be implemented in a function that receives a value of x and returns the corresponding value of y:

calcy.m

```
function y = calcy(x)
% calcy calculates y as a function of x
% Format of call: calcy(x)
% y = 1          if    x < -1
% y = x^2        if    -1 <= x <= 2
% y = 4          if    x > 2

if x < -1
    y = 1;
elseif x <= 2
    y = x^2;
else
    y = 4;
end
end
```

```
>> x = 1.1;
>> y = calcy(x)
 y =
   1.2100
```

QUICK QUESTION!

How could you write a function to determine whether an input argument is a scalar, a vector, or a matrix?

Answer: To do this, the **size** function can be used to find the dimensions of the input argument. If both the number of rows and columns is equal to 1, then the input argument is scalar. If, however, only one dimension is 1, the input argument is a vector (either a row or column vector). If neither dimension is 1, the input argument is a matrix. These three options can be tested using a nested **if-else** statement. In this example, the word 'scalar,' 'vector,' or 'matrix' is returned from the function.

findargtype.m

```
function outtype = findargtype(inputarg)
% findargtype determines whether the input
%    argument is a scalar, vector, or matrix
% Format of call: findargtype(inputArgument)
% Returns a string

[r c] = size(inputarg);
if r ==1 && c ==1
   outtype = 'scalar';
elseif r ==1 || c ==1
    outtype = 'vector';
else
    outtype = 'matrix';
end
end
```

Note that there is no need to check for the last case: if the input argument isn't a scalar or a vector, it must be a matrix! Examples of calling this function are:

```
>> findargtype(33)
ans =
scalar

>> disp(findargtype(2:5))
vector

>> findargtype(zeros(2,3))
ans =
matrix
```

PRACTICE 4.3

Modify the function *findargtype* to return either 'scalar,' 'row vector,' 'column vector,' or 'matrix,' depending on the input argument.

PRACTICE 4.4

Modify the original function *findargtype* to use three separate **if** statements instead of a nested **if-else** statement.

Another example demonstrates choosing from more than just a few options. The following function receives an integer quiz grade, which should be in the range from 0 to 10. The function then returns a corresponding letter grade, according to the following scheme: a 9 or 10 is an 'A,' an 8 is a 'B,' a 7 is a 'C,' a 6 is a 'D,' and anything below that is an 'F.' As the possibilities are mutually exclusive, we could implement the grading scheme using separate **if** statements. However, it is more efficient to have one **if-else** statement with multiple **elseif** clauses. Also, the function returns the letter 'X' if the quiz grade is not valid. The function assumes that the input is an integer.

letgrade.m

```
function grade = letgrade(quiz)
% letgrade returns the letter grade corresponding
%   to the integer quiz grade argument
% Format of call: letgrade(integerQuiz)
% Returns a character

% First, error-check
if quiz < 0 || quiz > 10
    grade = 'X';

% If here, it is valid so figure out the
%  corresponding letter grade
elseif quiz == 9 || quiz == 10
    grade = 'A';
elseif quiz == 8
    grade = 'B';
elseif quiz == 7
    grade = 'C';
elseif quiz == 6
    grade = 'D';
else
    grade = 'F';
end
end
```

Three examples of calling this function are:

```
>> quiz =8;
>> lettergrade = letgrade(quiz)
lettergrade =
B

>> quiz =4;
>> letgrade(quiz)
ans =
F

>> lg = letgrade(22)
lg =
X
```

In the part of this **if** statement that chooses the appropriate letter grade to return, all of the **logical** expressions are testing the value of the variable *quiz* to see if it is equal to several possible values, in sequence (first 9 or 10, then 8, then 7, etc.) This part can be replaced by a **switch** statement.

4.4 THE SWITCH STATEMENT

A **switch** statement can often be used in place of a nested **if-else** or an **if** statement with many **elseif** clauses. **Switch** statements are used when an expression is tested to see whether it is *equal to* one of several possible values.

The general form of the **switch** statement is:

```
switch switch_expression
  case caseexp1
    action1
  case caseexp2
    action2
  case caseexp3
    action3
  % etc: there can be many of these
  otherwise
    actionn
end
```

The **switch** statement starts with the reserved word **switch**, and ends with the reserved word **end**. The *switch_expression* is compared, in sequence, to the **case** expressions (*caseexp1*, *caseexp2*, etc.). If the value of the *switch_expression* matches *caseexp1*, for example, then *action1* is executed and the **switch** statement ends. If the value matches *caseexp3*, then *action3* is executed, and in general if the value matches *caseexpi* where *i* can be any integer from 1 to *n*, then *actioni* is executed. If the value of the *switch_expression* does

not match any of the **case** expressions, the action after the word **otherwise** is executed (the *n*th action, *actionn*) if there is an **otherwise** (if not, no action is executed). It is not necessary to have an **otherwise** clause, although it is frequently useful. The *switch_expression* must be either a scalar or a string.

For the previous example, the **switch** statement can be used as follows:

switchletgrade.m

```
function grade = switchletgrade(quiz)
% switchletgrade returns the letter grade corresponding
%    to the integer quiz grade argument using switch
% Format of call: switchletgrade(integerQuiz)
% Returns a character

% First, error-check
if quiz <0 || quiz >10
    grade = 'X';
else
    % If here, it is valid so figure out the
    %  corresponding letter grade using a switch
    switch quiz
        case 10
            grade = 'A';
        case 9
            grade = 'A';
        case 8
            grade = 'B';
        case 7
            grade = 'C';
        case 6
            grade = 'D';
        otherwise
            grade = 'F';
    end
end
end
```

Here are two examples of calling this function:

```
>> quiz = 22;
>> lg = switchletgrade(quiz)
lg =
X

>> switchletgrade(9)
ans =
A
```

Note

that it is assumed that the user will enter an integer value. If the user does not, either an error message will be printed or an incorrect result will be returned. Methods for remedying this will be discussed in Chapter 5.

As the same action of printing 'A' is desired for more than one grade, these can be combined as follows:

```
switch quiz
     case {10,9}
         grade = 'A';
     case 8
         grade = 'B';
       % etc.
```

The curly braces around the **case** expressions 10 and 9 are necessary.

In this example, we error-checked first using an **if-else** statement. Then, if the grade was in the valid range, a **switch** statement was used to find the corresponding letter grade.

Sometimes the **otherwise** clause is used for the error message rather than first using an **if-else** statement. For example, if the user is supposed to enter only a 1, 3, or 5, the script might be organized as follows:

switcherror.m

```
% Example of otherwise for error message

choice = input('Enter a 1, 3, or 5: ');

switch choice
    case 1
        disp('It''s a one!!')
    case 3
        disp('It''s a three!!')
    case 5
        disp('It''s a five!!')
    otherwise
        disp('Follow directions next time!!')
end
```

In this example, actions are taken if the user correctly enters one of the valid options. If the user does not, the **otherwise** clause handles printing an error message. Note the use of two single quotes within the string to print one quote.

```
>> switcherror
Enter a 1, 3, or 5: 4
Follow directions next time!!
```

Note that the order of the case expressions does not matter, except that this is the order in which they will be evaluated.

MATLAB has a built-in function called **menu** that will display a Figure Window with pushbuttons for the options. A script that uses this **menu** function would then use either an **if-else** statement or a **switch** statement to take an appropriate

action based on the button pushed. As of Version R2015b, however, the **menu** function is no longer recommended. Alternates will be found in Chapter 13 when Graphical User Interfaces are covered; in that chapter we will see how to create our own groups of pushbuttons, radio buttons, and other graphical objects.

4.5 THE "IS" FUNCTIONS IN MATLAB

There are a lot of functions that are built into MATLAB that test whether or not something is **true**; these functions have names that begin with the word "is." For example, we have already seen the use of the **isequal** function to compare arrays for equality. As another example, the function called **isletter** returns **logical** 1 if the character argument is a letter of the alphabet, or 0 if it is not:

```
>> isletter('h')
ans =
     1
>> isletter('4')
ans =
     0
```

The **isletter** function will return **logical true** or **false** so it can be used in a condition in an **if** statement. For example, here is the code that would prompt the user for a character, and then print whether or not it is a letter:

```
mychar = input('Please enter a char: ','s');
if isletter(mychar)
   disp('Is a letter')
else
   disp('Not a letter')
end
```

When used in an **if** statement, it is not necessary to test the value to see whether the result from **isletter** is equal to 1 or 0; this is redundant. In other words, in the condition of the **if** statement,

```
isletter(mychar)
```

and

```
isletter(mychar) == 1
```

would produce the same results.

QUICK QUESTION!

How can we write our own function *myisletter* to accomplish the same result as **isletter**?

Answer: The function would compare the character's position within the character encoding.

myisletter.m

```
function outlog = myisletter(inchar)
% myisletter returns true if the input argument
% is a letter of the alphabet or false if not
% Format of call: myisletter(input Character)
% Returns logical 1 or 0

outlog = inchar >= 'a' && inchar <= 'z' ...
         || inchar >= 'A' && inchar <= 'Z';
end
```

Note that it is necessary to check for both lowercase and uppercase letters.

The function **isempty** returns **logical true** if a variable is empty, **logical false** if it has a value, or an error message if the variable does not exist. Therefore, it can be used to determine whether a variable has a value yet or not. For example,

```
>> clear
>> isempty(evec)
Undefined function or variable 'evec'.

>> evec = [];
>> isempty(evec)
ans =
     1

>> evec = [evec 5];
>> isempty(evec)
ans =
     0
```

The **isempty** function will also determine whether or not a string variable is empty. This can be used to determine whether the user entered a string in an **input** function. In the following example, when prompted, the user simply hit the Return key.

```
>> istr = input('Please enter a string: ','s');
Please enter a string:
>> isempty(istr)
ans =
     1
```

PRACTICE 4.5

Prompt the user for a string, and then print either the string that the user entered or an error message if the user did not enter anything.

The **isa** function can be used to determine whether the first argument is a particular type.

```
>> num = 11;
>> isa(num, 'int16')
ans =
     0
>> isa(num,'double')
ans =
     1
```

The function **iskeyword** will determine whether or not a string is the name of a keyword in MATLAB, and therefore something that cannot be used as an identifier name. By itself (with no arguments), it will return the list of all keywords. Note that the names of functions like "sin" are not keywords, so their values can be overwritten if used as an identifier name.

```
>> iskeyword('sin')
ans =
     0
>> iskeyword('switch')
ans =
     1

>> iskeyword
ans =
    'break'
    'case'
    'catch'

     % etc.
```

There are many other "is" functions; the complete list can be found in the Help browser.

■ Explore Other Interesting Features

There are many other "is" functions. As more concepts are covered in the book, more and more of these functions will be introduced. Others that you may want to explore now include **isvarname**, and functions that will tell you whether an argument is a particular type or not (**ischar**, **isfloat**, **isinteger**, **islogical**, **isnumeric**, **isstr**, and **isreal**).

There are "is" functions to determine the type of an array: **isvector**, **isrow**, and **iscolumn**.

The **try/catch** functions are a particular type of **if-else** used to find and avoid potential errors. They may be a bit complicated to understand at this point, but keep them in mind for the future! ■

SUMMARY

COMMON PITFALLS

- Using = instead of == for equality in conditions
- Putting a space in the keyword **elseif**
- Not using quotes when comparing a string variable to a string, such as

  ```
  letter == y
  ```

  instead of

  ```
  letter == 'y'
  ```

- Not spelling out an entire **logical** expression. An example is typing

  ```
  radius || height <= 0
  ```

  instead of

  ```
  radius <= 0 || height <= 0
  ```

  or typing

  ```
  letter == 'y' || 'Y'
  ```

  instead of

  ```
  letter == 'y' || letter == 'Y'
  ```

  Note that these are logically incorrect, but would not result in error messages. Note also that the expression "`letter == 'y' || 'Y'`" will ALWAYS be **true**, regardless of the value of the variable *letter*, as `'Y'` is a nonzero value and therefore a **true** expression.
- Writing conditions that are more complicated than necessary, such as

  ```
  if (x <5) == 1
  ```

  instead of just

  ```
  if x < 5
  ```

  (The "==1" is redundant.)
- Using an **if** statement instead of an **if-else** statement for error-checking; for example,

  ```
  % Wrong method
  if error occurs
       print error message
  end

  continue rest of code
  ```

instead of

```
% Correct method
if error occurs
     print error message
else
     continue rest of code
end
```

In the first example, the error message would be printed but then the program would continue anyway.

PROGRAMMING STYLE GUIDELINES

- Use indentation to show the structure of a script or function. In particular, the actions in an **if** statement should be indented.
- When the **else** clause is not needed, use an **if** statement rather than an **if-else** statement. The following is an example:

```
if unit == 'i'
   len = len * 2.54;
else
   len = len; % this does nothing so skip it!
end
```

Instead, just use:

```
if unit == 'i'
   len = len * 2.54;
end
```

- Do not put unnecessary conditions on **else** or **elseif** clauses. For example, the following prints one thing if the value of a variable *number* is equal to 5, and something else if it is not.

```
if number == 5
   disp('It is a 5')
elseif number ~= 5
   disp('It is not a 5')
end
```

The second condition, however, is not necessary. Either the value is 5 or not, so just the **else** would handle this:

```
if number == 5
   disp('It is a 5')
else
   disp('It is not a 5')
end
```

if	else	case
switch	elseif	otherwise

error	isletter	isa
menu	isempty	iskeyword

Exercises

1. Write a script that tests whether the user can follow instructions. It prompts the user to enter an 'x.' If the user enters anything other than an 'x,' it prints an error message—otherwise, the script does nothing.
2. Write a function *nexthour* that receives one integer argument, which is an hour of the day, and returns the next hour. This assumes a 12-hour clock; so, for example, the next hour after 12 would be 1. Here are two examples of calling this function.

```
>> fprintf('The next hour will be %d.\n',nexthour(3))
The next hour will be 4.
>> fprintf('The next hour will be %d.\n',nexthour(12))
The next hour will be 1.
```

3. The speed of a sound wave is affected by the temperature of the air. At 0 °C, the speed of a sound wave is 331 m/sec. The speed increases by approximately 0.6 m/sec for every degree (in Celsius) above 0; this is a reasonably accurate approximation for 0–50°C. So, our equation for the speed in terms of a temperature C is:

```
speed = 331 + 0.6 * C
```

   Write a script *soundtemp* that will prompt the user for a temperature in Celsius in the range from 0 to 50 inclusive, and will calculate and print the speed of sound at that temperature if the user enters a temperature in that range, or an error message if not. Here are some examples of using the script:

```
>> soundtemp
Enter a temp in the range 0 to 50: -5.7
Error in temperature
>> soundtemp
Enter a temp in the range 0 to 50: 10
For a temperature of 10.0, the speed is 337.0
>> help soundtemp
Calculates and prints the speed of sound given a
temperature entered by the user
```

4. When would you use just an **if** statement and not an **if-else**?

5. Come up with "trigger words" in a problem statement that would tell you when it would be appropriate to use **if**, **if-else**, or **switch** statements.
6. Write a statement that will store **logical true** in a variable named "isit" if the value of a variable "*x*" is in the range from 0 to 10, or **logical false** if not. Do this with just one assignment statement, with no **if** or **if-else** statement!
7. The Pythagorean theorem states that for a right triangle, the relationship between the length of the hypotenuse *c* and the lengths of the other sides *a* and *b* is given by:

   $c^2 = a^2 + b^2$

   Write a script that will prompt the user for the lengths *a* and *c*, call a function *findb* to calculate and return the length of *b*, and print the result. Note that any values of *a* or *c* that are less than or equal to zero would not make sense, so the script should print an error message if the user enters any invalid value. Here is the function *findb*:

   findb.m

   ```
   function b = findb(a,c)
   % Calculates b from a and c
   b = sqrt(c^2 - a^2);
   end
   ```

8. The area A of a rhombus is defined as $A = \frac{d_1 d_2}{2}$, where d_1 and d_2 are the lengths of the two diagonals. Write a script *rhomb* that first prompts the user for the lengths of the two diagonals. If either is a negative number or zero, the script prints an error message. Otherwise, if they are both positive, it calls a function *rhombarea* to return the area of the rhombus, and prints the result. Write the function, also! The lengths of the diagonals, which you can assume are in inches, are passed to the *rhombarea* function.
9. A data file "parttolerance.dat" stores on one line, a part number, and the minimum and maximum values for the valid range that the part could weigh. Write a script "parttol" that will read these values from the file, prompt the user for a weight, and print whether or not that weight is within range.
   For example, IF the file stores the following:

   ```
   >> type parttolerance.dat
   123    44.205    44.287
   ```

   Here might be examples of executing the script:

   ```
   >> parttol
   Enter the part weight: 44.33
   The part 123 is not in range
   >> parttol
   Enter the part weight: 44.25
   The part 123 is within range
   ```

10. Write a script that will prompt the user for a character. It will create an *x*-vector that has 50 numbers, equally spaced between -2π and 2π, and then a *y*-vector

which is cos(x). If the user entered the character 'r,' it will plot these vectors with red *s—otherwise, for any other character it will plot the points with green +s.

11. Simplify this statement:

```
if number > 100
   number = 100;
else
   number = number;
end
```

12. Simplify this statement:

```
if val >= 10
   disp('Hello')
elseif val < 10
   disp('Hi')
end
```

13. The continuity equation in fluid dynamics for steady fluid flow through a stream tube equates the product of the density, velocity, and area at two points that have varying cross-sectional areas. For incompressible flow, the densities are constant so the equation is $A_1V_1 = A_2V_2$. If the areas and V_1 are known, V_2 can be found as $\frac{A_1}{A_2}V_1$. Therefore, whether the velocity at the second point increases or decreases depend on the areas at the two points. Write a script that will prompt the user for the two areas in square feet, and will print whether the velocity at the second point will increase, decrease, or remain the same as at the first point.

14. Write a function *eqfn* that will calculate $f(x) = x^2 + \frac{1}{x}$ for all elements of x. Since division by 0 is not possible, if any element in x is zero, the function will instead return a flag of −99. Here are examples of using this function:

```
>> vec = [5  0  11  2];
>> eqfn(vec)
ans =
   -99
>> result = eqfn(4)
result =
   16.2500
>> eqfn(2:5)
ans =
    4.5000   9.3333  16.2500  25.2000
```

15. In chemistry, the pH of an aqueous solution is a measure of its acidity. The pH scale ranges from 0 to 14, inclusive. A solution with a pH of 7 is said to be *neutral*, a solution with a pH greater than 7 is *basic*, and a solution with a pH less than 7 is *acidic*. Write a script that will prompt the user for the pH of a solution, and will print whether it is neutral, basic, or acidic. If the user enters an invalid pH, an error message will be printed.

16. Write a function *flipvec* that will receive one input argument. If the input argument is a row vector, the function will reverse the order and return a new row vector. If the input argument is a column vector, the function will reverse the order and return a new column vector. If the input argument is a matrix or a scalar, the function will return the input argument unchanged.
17. In a script, the user is supposed to enter either a 'y' or 'n' in response to a prompt. The user's input is read into a character variable called "letter." The script will print "OK, continuing" if the user enters either a 'y' or 'Y' or it will print "OK, halting" if the user enters a 'n' or 'N' or "Error" if the user enters anything else. Put this statement in the script first:

    ```
    letter = input('Enter your answer: ', 's');
    ```

    Write the script using a single nested **if-else** statement (**elseif** clause is permitted).
18. Write the script from the previous exercise using a **switch** statement instead.
19. In aerodynamics, the Mach number is a critical quantity. It is defined as the ratio of the speed of an object (e.g., an aircraft) to the speed of sound. If the Mach number is less than 1, the flow is subsonic; if the Mach number is equal to 1, the flow is transonic; and if the Mach number is greater than 1, the flow is supersonic. Write a script that will prompt the user for the speed of an aircraft and the speed of sound at the aircraft's current altitude and will print whether the condition is subsonic, transonic, or supersonic.
20. Write a script that will generate one random integer and will print whether the random integer is an even or an odd number. (Hint: an even number is divisible by 2, whereas an odd number is not; so check the remainder after dividing by 2.)

    Global temperature changes have resulted in new patterns of storms in many parts of the world. Tracking wind speeds and a variety of categories of storms is important in understanding the ramifications of these temperature variations. Programs that work with storm data will use selection statements to determine the severity of storms and also to make decisions based on the data.
21. Whether a storm is a tropical depression, tropical storm, or hurricane is determined by the average sustained wind speed. In miles per hour, a storm is a tropical depression if the winds are less than 38 mph. It is a tropical storm if the winds are between 39 and 73 mph, and it is a hurricane if the wind speeds are >= 74 mph. Write a script that will prompt the user for the wind speed of the storm, and will print which type of storm it is.
22. The Beaufort Wind Scale is used to characterize the strength of winds. The scale uses integer values and goes from a force of 0, which is no wind, up to 12, which is a hurricane. The following script first generates a random force value. Then, it prints a message regarding what type of wind that force represents, using a **switch** statement. You are to rewrite this **switch** statement as one nested **if-else** statement that accomplishes exactly the same thing. You may use **else** and/or **elseif** clauses.

```
ranforce = randi([0, 12]);
switch ranforce
    case 0
        disp('There is no wind')
    case {1,2,3,4,5,6}
        disp('There is a breeze')
    case {7,8,9}
        disp('This is a gale')
    case {10,11}
        disp('It is a storm')
    case 12
        disp('Hello, Hurricane!')
end
```

23. Rewrite the following **switch** statement as one nested **if-else** statement (**elseif** clauses may be used). Assume that there is a variable *letter* and that it has been initialized.

```
switch letter
      case 'x'
           disp('Hello')
      case {'y', 'Y'}
           disp('Yes')
      case 'Q'
           disp('Quit')
      otherwise
           disp('Error')
end
```

24. Rewrite the following nested **if-else** statement as a **switch** statement that accomplishes exactly the same thing. Assume that *num* is an integer variable that has been initialized, and that there are functions *f1*, *f2*, *f3*, and *f4*. Do not use any **if** or **if-else** statements in the actions in the **switch** statement, only calls to the four functions.

```
if num < -2 || num > 4
     f1(num)
else
     if num <= 2
          if num >= 0
               f2(num)
          else
               f3(num)
          end
     else
          f4(num)
     end
end
```

25. Write a script *areaMenu* that will print a list consisting of "cylinder," "circle," and "rectangle." It prompts the user to choose one, and then prompts the user

for the appropriate quantities (e.g., the radius of the circle) and then prints its area. If the user enters an invalid choice, the script simply prints an error message. The script should use a nested **if-else** statement to accomplish this. Here are two examples of running it (units are assumed to be inches).

```
>> areaMenu
Menu
1. Cylinder
2. Circle
3. Rectangle
Please choose one: 2
Enter the radius of the circle: 4.1
The area is 52.81

>> areaMenu
Menu
1. Cylinder
2. Circle
3. Rectangle
Please choose one: 3
Enter the length: 4
Enter the width: 6
The area is 24.00
```

26. Modify *the areaMenu* script to use a **switch** statement to decide which area to calculate.
27. Write a script that will prompt the user for a string and then print whether it was empty or not.
28. Simplify this statement:

```
if iskeyword('else') == 1
   disp('Cannot use as a variable name')
end
```

29. Store a value in a variable and then use **isa** to test to see whether or not it is the type **double**.
30. Write a function called "makemat" that will receive two row vectors as input arguments, and from them create and return a matrix with two rows. You may not assume that the length of the vectors is known. Also, the vectors may be of different lengths. If that is the case, add 0's to the end of one vector first to make it as long as the other. For example, a call to the function might be:

```
>>makemat(1:4, 2:7)
ans =
    1    2    3    4    0    0
    2    3    4    5    6    7
```

CHAPTER 5

Loop Statements and Vectorizing Code

KEY TERMS

looping statements
counted loops
conditional loops
action
vectorized code
iterate
loop or iterator variable
echo printing
running sum
running product
preallocate
nested loop
outer loop
inner loop
infinite loop
factorial
sentinel
counting
error-checking

CONTENTS

Consider the problem of calculating the area of a circle with a radius of 0.3 cm. A MATLAB® program certainly is not needed to do that; you'd use your calculator instead, and punch in $\pi * 0.3^2$. However, if a table of circle areas is desired, for radii ranging from 0.1 to 100 cm in steps of 0.05 (e.g., 0.1, 0.15, 0.2, etc.), it would be very tedious to use a calculator and write it all down. One of the great uses of programming languages and software packages such as MATLAB is the ability to repeat a process such as this.

This chapter will cover statements in MATLAB that allow other statement(s) to be repeated. The statements that do this are called *looping statements* or ***loops***. There are two basic kinds of loops in programming: ***counted loops*** and ***conditional loops***. A counted loop is a loop that repeats statements a specified number of times (so, ahead of time it is known how many times the statements are to be repeated). In a counted loop, for example, you might say "repeat these statements 10 times". A conditional loop also repeats statements, but ahead of time it is not known *how many* times the statements will need to be repeated. With a conditional loop, for example, you might say "repeat these statements until this condition becomes false". The statement(s) that are repeated in any loop are called the ***action*** of the loop.

There are two different loop statements in MATLAB: the **for** statement and the **while** statement. In practice, the **for** statement is used as the counted loop, and

MATLAB®. http://dx.doi.org/10.1016/B978-0-12-804525-1.00005-2

the **while** is usually used as the conditional loop. To keep it simple, that is how they will be presented here.

In many programming languages, looping through the elements in a vector or matrix is a very fundamental concept. In MATLAB, however, as it is written to work with vectors and matrices, looping through elements is usually not necessary. Instead, "vectorized code" is used, which means replacing the loops through matrices with the use of built-in functions and operators. Both methods will be described in this chapter. The earlier sections will focus on "the programming concepts", using loops. These will be contrasted with "the efficient methods", using ***vectorized code***. Loops are still relevant and necessary in MATLAB in other contexts, just not normally when working with vectors or matrices.

5.1 THE FOR LOOP

The **for** statement, or the **for** loop, is used when it is necessary to repeat statement(s) in a script or function, and when it *is* known ahead of time how many times the statements will be repeated. The statements that are repeated are called the action of the loop. For example, it may be known that the action of the loop will be repeated five times. The terminology used is that we ***iterate*** through the action of the loop five times.

The variable that is used to iterate through values is called a ***loop variable*** or an ***iterator variable***. For example, the variable might iterate through the integers 1 through 5 (e.g., 1, 2, 3, 4, and then 5). Although, in general, variable names should be mnemonic, it is common in many languages for an iterator variable to be given the name *i* (and if more than one iterator variable is needed, *i*, *j*, *k*, *l*, etc.) This is historical, and is because of the way integer variables were named in Fortran. However, in MATLAB both **i** and **j** are built-in functions that return the value $\sqrt{-1}$, so using either as a loop variable will override that value. If that is not an issue, then it is okay to use *i* as a loop variable.

The general form of the **for** loop is:

```
for loopvar = range
    action
end
```

where *loopvar* is the loop variable, "range" is the range of values through which the loop variable is to iterate, and the action of the loop consists of all statements up to the **end**. Just like with **if** statements, the action is indented to make it easier to see. The range can be specified using any vector, but normally the easiest way to specify the range of values is to use the colon operator.

As an example, we will print a column of numbers from 1 to 5.

THE PROGRAMMING CONCEPT

The loop could be entered in the Command Window, although, like **if** and **switch** statements, loops will make more sense in scripts and functions. In the Command Window, the results would appear after the **for** loop:

```
>> for i = 1:5
       fprintf('%d\n',i)
   end
1
2
3
4
5
```

What the **for** statement accomplished was to print the value of *i* and then the newline character for every value of *i*, from 1 through 5 in steps of 1. The first thing that happens is that *i* is initialized to have the value 1. Then, the action of the loop is executed, which is the **fprintf** statement that prints the value of *i* (1), and then the newline character to move the cursor down. Then, *i* is incremented to have the value of 2. Next, the action of the loop is executed, which prints 2 and the newline. Then, *i* is incremented to 3 and that is printed; then, *i* is incremented to 4 and that is printed; and then, finally, *i* is incremented to 5 and that is printed. The final value of *i* is 5; this value can be used once the loop has finished.

THE EFFICIENT METHOD

Of course, **disp** could also be used to print a column vector, to achieve the same result:

```
>> disp([1:5]')
     1
     2
     3
     4
     5
```

QUICK QUESTION!

How could you print this column of integers (using the programming method):

```
  0
 50
100
150
200
```

Answer: In a loop, you could print these values starting at 0, incrementing by 50 and ending at 200. Each is printed using a field width of 3.

```
>> for i = 0:50:200
       fprintf('%3d\n',i)
   end
```

5.1.1 For Loops that do not Use the Iterator Variable in the Action

In the previous example, the value of the loop variable was used in the action of the **for** loop: it was printed. It is not always necessary to actually use the value of the loop variable, however. Sometimes the variable is simply used to iterate, or repeat, an action a specified number of times. For example,

```
for i = 1:3
    fprintf('I will not chew gum\n')
end
```

produces the output:

```
I will not chew gum
I will not chew gum
I will not chew gum
```

The variable *i* is necessary to repeat the action three times, even though the value of *i* is not used in the action of the loop.

QUICK QUESTION!

What would be the result of the following **for** loop?

```
for i = 4:2:8
    fprintf('I will not chew gum\n')
end
```

Answer: Exactly the same output as above! It doesn't matter that the loop variable iterates through the values 4, then 6, then 8 instead of 1, 2, 3. As the loop variable is not used in the action, this is just another way of specifying that the action should be repeated three times. Of course, using 1:3 makes more sense!

PRACTICE 5.1

Write a **for** loop that will print a column of five *'s.

5.1.2 Input in a for Loop

The following script repeats the process of prompting the user for a number and ***echo printing*** the number (which means simply printing it back out). A **for** loop specifies how many times this is to occur. This is another example in which the loop variable is not used in the action, but instead, just specifies how many times to repeat the action.

forecho.m

```
% This script loops to repeat the action of
% prompting the user for a number and echo-printing it

for iv = 1:3
    inputnum = input('Enter a number: ');
    fprintf('You entered %.1f\n',inputnum)
end
```

```
>> forecho
Enter a number: 33
You entered 33.0
Enter a number: 1.1
You entered 1.1
Enter a number: 55
You entered 55.0
```

In this example, the loop variable *iv* iterates through the values 1–3, so the action is repeated three times. The action consists of prompting the user for a number and echo printing it with one decimal place.

5.1.3 Finding Sums and Products

A very common application of a **for** loop is to calculate sums and products. For example, instead of just echo printing the numbers that the user enters, we could calculate the sum of the numbers. In order to do this, we need to add each value to a ***running sum***. A running sum keeps changing, as we keep adding to it. First, the sum has to be initialized to 0.

As an example, we will write a script *sumnnums* that will sum the *n* numbers entered by the user; *n* is a random integer that is generated. In a script to calculate the sum, we need a loop or iterator variable *i*, and also a variable to store the running sum. In this case we will use a variable *runsum* as the running sum. Every time through the loop, the next value that the user enters is added to the value of *runsum*. This script will print the end result, which is the sum of all the numbers, stored in the variable *runsum*.

sumnnums.m

```
% sumnnums calculates the sum of the n numbers
% entered by the user

n = randi([3 10]);
runsum = 0;
for i = 1:n
    inputnum = input('Enter a number: ');
    runsum = runsum + inputnum;
end
fprintf('The sum is %.2f\n', runsum)
```

Here is an example in which 3 is generated to be the value of the variable *n*; the script calculates and prints the sum of the numbers the user enters, 4 + 3.2 + 1.1, or 8.3:

```
>> sumnnums
Enter a number: 4
Enter a number: 3.2
Enter a number: 1.1
The sum is 8.30
```

Another very common application of a **for** loop is to find a ***running product***. With a product, the running product must be initialized to 1 (as opposed to a running sum, which is initialized to 0).

PRACTICE 5.2

Write a script *prodnnums* that is similar to the *sumnnums* script but will calculate and print the product of the numbers entered by the user.

5.1.4 Preallocating Vectors

When numbers are entered by the user, it is often necessary to store them in a vector. There are two basic methods that could be used to accomplish this. One method is to start with an empty vector and extend the vector by adding each number to it as the numbers are entered by the user. Extending a vector, however, is very inefficient. What happens is that every time a vector is extended, a new "chunk" of memory must be found that is large enough for the new vector, and all of the values must be copied from the original location in memory to the new one. This can take a long time to execute.

A better method is to ***preallocate*** the vector to the correct size and then change the value of each element to be the numbers that the user enters. This method involves referring to each index in the result vector, and placing each number into the next element in the result vector. This method is far superior, if it is known ahead of time how many elements the vector will have. One common method is to use the **zeros** function to preallocate the vector to the correct length.

The following is a script that accomplishes this and prints the resulting vector. The script generates a random integer *n* and repeats the process *n* times. As it is known that the resulting vector will have *n* elements, the vector can be preallocated.

forgenvec.m

```
% forgenvec creates a vector of length n
% It prompts the user and puts n numbers into a vector

n = randi([4   8]);
numvec = zeros(1,n);
for iv = 1:n
    inputnum = input('Enter a number: ');
    numvec(iv) = inputnum;
end
fprintf('The vector is: \n')
disp(numvec)
```

Next is an example of executing this script.

```
>> forgenvec
Enter a number: 44
Enter a number: 2.3
Enter a number: 11
The vector is:
   44.0000    2.3000   11.0000
```

It is very important to notice that the loop variable iv is used as the index into the vector.

QUICK QUESTION!

If you need to just print the sum or average of the numbers that the user enters, would you need to store them in a vector variable?

Answer: No. You could just add each to a running sum as you read them in a loop.

QUICK QUESTION!

What if you wanted to calculate how many of the numbers that the user entered were greater than the average?

Answer: Yes, then you would need to store them in a vector because you would have to go back through them to count how many were greater than the average (or, alternatively, you could go back and ask the user to enter them again!!).

5.1.5 For Loop Example: Subplot

A function that is very useful with all types of plots is **subplot**, which creates a matrix of plots in the current Figure Window. Three arguments are passed to it in the form **subplot(r,c,n)**, where *r* and *c* are the dimensions of the matrix and *n*

is the number of the particular plot within this matrix. The plots are numbered rowwise starting in the upper left corner. In many cases, it is useful to create a **subplot** in a **for** loop so the loop variable can iterate through the integers 1 through *n*.

For example, if it is desired to have three plots next to each other in one Figure Window, the function would be called as **subplot(1,3,n)**. The matrix dimensions in the Figure Window would be *1x3* in this case, and from left to right the individual plots would be numbered 1, 2, and then 3 (these would be the values of *n*). The first two arguments would always be 1 and 3, as they specify the dimensions of the matrix within the Figure Window.

When the **subplot** function is called in a loop, the first two arguments will always be the same as they give the dimensions of the matrix. The third argument will iterate through the numbers assigned to the elements of the matrix. When the **subplot** function is called, it makes the specified element the "active" plot; then, any plot function can be used, complete with formatting such as axis labeling and titles within that element. Note that the **subplot** function just specifies the dimensions of the matrix in the Figure Window, and which is the "active" element; **subplot** itself does not plot anything.

For example, the following **subplot** shows the difference, in one Figure Window, between using 20 points and 40 points to plot **sin(x)** between 0 and 2 * π. The **subplot** function creates a *1 x 2* row vector of plots in the Figure Window, so that the two plots are shown side by side. The loop variable *i* iterates through the values 1 and then 2.

The first time through the loop, when *i* has the value 1, 20 * 1 or 20 points are used, and the value of the third argument to the **subplot** function is 1. The second time through the loop, 40 points are used and the third argument to **subplot** is 2. The resulting Figure Window with both plots is shown in Fig 5.1.

subplotex.m

```
% Demonstrates subplot using a for loop
for i = 1:2
    x = linspace(0,2*pi,20*i);
    y = sin(x);
    subplot(1,2,i)
    plot(x,y,'ko')
    xlabel('x')
    ylabel('sin(x)')
    title('sin plot')
end
```

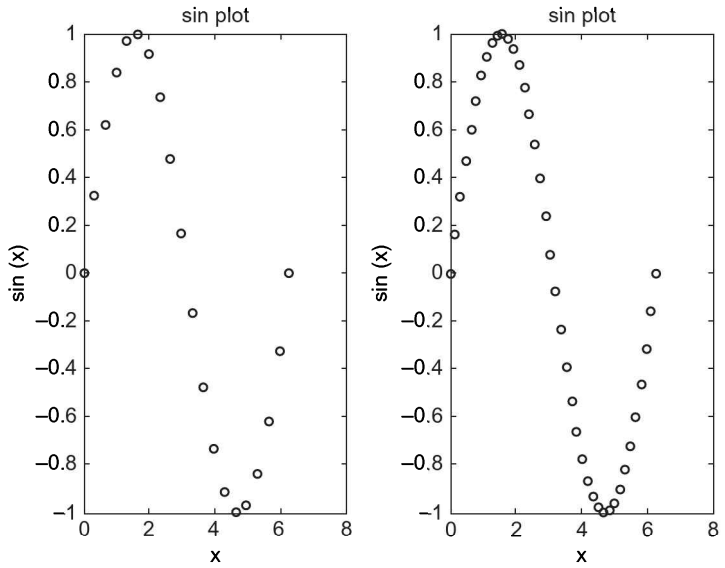

FIGURE 5.1
Subplot to demonstrate a plot using 20 points and 40 points.

Note that once string manipulating functions have been covered in Chapter 7, it will be possible to have customized titles (e.g., showing the number of points).

5.2 NESTED FOR LOOPS

The action of a loop can be any valid statement(s). When the action of a loop is another loop, this is called a *nested loop*.

The general form of a nested **for** loop is as follows:

```
for loopvarone = rangeone          ← outer loop

    % actionone includes the inner loop

    for loopvartwo = rangetwo      ← inner loop
        actiontwo
    end
end
```

The first **for** loop is called the ***outer loop***; the second **for** loop is called the ***inner loop***. The action of the outer loop consists (in part; there could be other statements) of the entire inner loop.

As an example, a nested **for** loop will be demonstrated in a script that will print a box of stars (*). Variables in the script will specify how many rows and

columns to print. For example, if *rows* has the value 3 and *columns* has the value 5, a *3 x 5* box would be printed. As lines of output are controlled by printing the newline character, the basic algorithm is as follows.

- For every row of output:
  Print the required number of stars
  Move the cursor down to the next line (print '\n')

printstars.m

```
% Prints a box of stars
% How many will be specified by two variables
%  for the number of rows and columns

rows = 3;
columns = 5;
% loop over the rows
for i=1:rows
    %  for every row loop to print *'s and then one \n
    for j=1:columns
        fprintf('*')
    end
    fprintf('\n')
end
```

Executing the script displays the output:

```
>> printstars
*****
*****
*****
```

The variable *rows* specifies the number of rows to print, and the variable *columns* specifies how many stars to print in each row. There are two loop variables: *i* is the loop variable over the rows and *j* is the loop variable over the columns. As the number of rows is known and the number of columns is known (given by the variables *rows* and *columns*), **for** loops are used. There is one **for** loop to loop over the rows and another to print the required number of stars for every row.

The values of the loop variables are not used within the loops, but are used simply to iterate the correct number of times. The first **for** loop specifies that the action will be repeated "rows" times. The action of this loop is to print stars and then the newline character. Specifically, the action is to loop to print *columns* stars (e.g., five stars) across on one line. Then, the newline character is printed after all five stars to move the cursor down to the next line.

In this case, the outer loop is over the rows and the inner loop is over the columns. The outer loop must be over the rows because the script is printing a certain number of rows of output. For each row, a loop is necessary to print the required number of stars; this is the inner **for** loop.

When this script is executed, first the outer loop variable *i* is initialized to 1. Then, the action is executed. The action consists of the inner loop and then printing the newline character. So, while the outer loop variable has the value 1, the inner loop variable *j* iterates through all of its values. As the value of *columns* is 5, the inner loop will print a single star five times. Then, the newline character is printed and then the outer loop variable *i* is incremented to 2. The action of the outer loop is then executed again, meaning the inner loop will print five stars, and then the newline character will be printed. This continues, and in all, the action of the outer loop will be executed *rows* times.

Notice that the action of the outer loop consists of two statements (the **for** loop and an **fprintf** statement). The action of the inner loop, however, is only a single **fprintf** statement.

The **fprintf** statement to print the newline character must be separate from the other **fprintf** statement that prints the star character. If we simply had

```
fprintf('*\n')
```

as the action of the inner loop, this would print a long column of 15 stars, not a *3 x 5* box.

QUICK QUESTION!

How could this script be modified to print a triangle of stars instead of a box such as the following:

```
*
**
***
```

Answer: In this case, the number of stars to print in each row is the same as the row number (e.g., one star is printed in row 1, two stars in row 2, and so on). The inner **for** loop does not loop to columns, but to the value of the row loop variable (so we do not need the variable *columns*):

printtristars.m

```
% Prints a triangle of stars
% How many will be specified by a variable
%  for the number of rows
rows = 3;
for i=1:rows
    % inner loop just iterates to the value of i
    for j=1:i
        fprintf('*')
    end
    fprintf('\n')
end
```

In the previous examples, the loop variables were just used to specify the number of times the action is to be repeated. In the next example, the actual values of the loop variables will be printed.

printloopvars.m

```
% Displays the loop variables
for i = 1:3
    for j = 1:2
        fprintf('i=%d, j=%d\n',i,j)
    end
    fprintf('\n')
end
```

Executing this script would print the values of both *i* and *j* on one line every time the action of the inner loop is executed. The action of the outer loop consists of the inner loop and printing a newline character, so there is a separation between the actions of the outer loop:

```
>> printloopvars
i=1, j=1
i=1, j=2

i=2, j=1
i=2, j=2

i=3, j=1
i=3, j=2
```

Now, instead of just printing the loop variables, we can use them to produce a multiplication table, by multiplying the values of the loop variables.

The following function *multtable* calculates and returns a matrix which is a multiplication table. Two arguments are passed to the function, which are the number of rows and columns for this matrix.

multtable.m

```
function outmat = multtable(rows, columns)
% multtable returns a matrix which is a
% multiplication table
% Format: multtable(nRows, nColumns)

% Preallocate the matrix
outmat = zeros(rows,columns);
for i = 1:rows
    for j = 1:columns
        outmat(i,j) = i*j;
    end
end
end
```

In the following example of calling this function, the resulting matrix has three rows and five columns:

```
>> multtable(3,5)
ans =
     1     2     3     4     5
     2     4     6     8    10
     3     6     9    12    15
```

Note that this is a function that returns a matrix. It preallocates the matrix to zeros, and then replaces each element. As the number of rows and columns are known, **for** loops are used. The outer loop loops over the rows and the inner loop loops over the columns. The action of the nested loop calculates i * j for all values of *i* and *j*. *Just like with vectors, it is again important to notice that the loop variables are used as the indices into the matrix.*

First, when *i* has the value 1, *j* iterates through the values 1 through 5, so first we are calculating 1*1, then 1*2, then 1*3, then 1*4, and finally, 1*5. These are the values in the first row (first in element (1,1), then (1,2), then (1,3), then (1,4), and finally (1,5)). Then, when *i* has the value 2, the elements in the second row of the output matrix are calculated, as *j* again iterates through the values from 1 through 5. Finally, when *i* has the value 3, the values in the third row are calculated (3*1, 3*2, 3*3, 3*4, and 3*5).

This function could be used in a script that prompts the user for the number of rows and columns, calls this function to return a multiplication table, and writes the resulting matrix to a file:

createmulttab.m

```
% Prompt the user for rows and columns and
%  create a multiplication table to store in
%  a file "mymulttable.dat"

num_rows = input('Enter the number of rows: ');
num_cols = input('Enter the number of columns: ');
multmatrix = multtable(num_rows, num_cols);
save mymulttable.dat multmatrix -ascii
```

The following is an example of running this script, and then loading from the file into a matrix in order to verify that the file was created:

```
>> createmulttab
Enter the number of rows: 6
Enter the number of columns: 4

>> load mymulttable.dat
```

```
>> mymulttable
mymulttable =
     1     2     3     4
     2     4     6     8
     3     6     9    12
     4     8    12    16
     5    10    15    20
     6    12    18    24
```

PRACTICE 5.3

For each of the following (they are separate), determine what would be printed. Then, check your answers by trying them in MATLAB.

```
mat = [7   11   3;   3:5];
[r, c] = size(mat);
for i = 1:r
    fprintf('The sum is %d\n', sum(mat(i,:)))
end
----------------------------------------
for i = 1:2
    fprintf('%d: ', i)
    for j = 1:4
        fprintf('%d ', j)
    end
    fprintf('\n')
end
```

5.2.1 Combining Nested for Loops and if Statements

The statements inside of a nested loop can be any valid statements, including any selection statement. For example, there could be an **if** or **if-else** statement as the action, or part of the action, in a loop.

As an example, assume there is a file called "datavals.dat" containing results recorded from an experiment. However, some were recorded erroneously. The numbers are all supposed to be positive. The following script reads from this file into a matrix. It prints the sum from each row of only the positive numbers. We will assume that the file contains integers, but will not assume how many lines are in the file nor how many numbers per line (although we will assume that there are the same number of integers on every line).

sumonlypos.m

```
% Sums only positive numbers from file
% Reads from the file into a matrix and then
%   calculates and prints the sum of only the
%    positive numbers from each row

load datavals.dat
[r c] = size(datavals);

for row = 1:r
    runsum = 0;
    for col = 1:c
        if datavals(row,col) >= 0
            runsum = runsum+datavals(row,col);
        end
    end
    fprintf('The sum for row %d is %d\n',row,runsum)
end
```

For example, *if* the file contains:

```
33  -11    2
 4    5    9
22    5   -7
 2   11    3
```

the output from the program would look like this:

```
>> sumonlypos
The sum for row 1 is 35
The sum for row 2 is 18
The sum for row 3 is 27
The sum for row 4 is 16
```

The file is loaded and the data are stored in a matrix variable. The script finds the dimensions of the matrix and then loops through all of the elements in the matrix by using a nested loop; the outer loop iterates through the rows and the inner loop iterates through the columns. This is important; as an action is desired for every row, the outer loop has to be over the rows. For each element an **if-else** statement determines whether the element is positive or not. It only adds the positive values to the row sum. As the sum is found for each row, the *runsum* variable is initialized to 0 for every row, meaning inside of the outer loop.

QUICK QUESTION!

Would it matter if the order of the loops was reversed in this example, so that the outer loop iterates over the columns and the inner loop over the rows?

Answer: Yes, as we want a sum for every row the outer loop must be over the rows.

QUICK QUESTION!

What would you have to change in order to calculate and print the sum of only the positive numbers from each column instead of each row?

Answer: You would reverse the two loops, and change the sentence to say "The sum of column...". That is all that would change. The elements in the matrix would still be referenced as datavals(row,col). The row index is always given first, then the column index – regardless of the order of the loops.

PRACTICE 5.4

Write a function *mymatmin* that finds the minimum value in each column of a matrix argument and returns a vector of the column minimums. An example of calling the function follows:

```
>> mat = randi(20,3,4)
mat =
    15    19    17     5
     6    14    13    13
     9     5     3    13

>> mymatmin(mat)
ans =
     6     5     3     5
```

QUICK QUESTION!

Would the function *mymatmin* in Practice 5.4 also work for a vector argument?

Answer: Yes, it should, as a vector is just a subset of a matrix. In this case, one of the loop actions would be executed only one time (for the rows if it is a row vector or for the columns if it is a column vector).

5.3 WHILE LOOPS

The **while** statement is used as the conditional loop in MATLAB; it is used to repeat an action when ahead of time it is *not known how many* times the action will be repeated. The general form of the **while** statement is:

```
while condition
    action
end
```

The action, which consists of any number of statement(s), is executed as long as the condition is true.

The way it works is that first the condition is evaluated. If it is logically **true**, the action is executed. So, to begin with, the **while** statement is just like an **if** statement. However, at that point, the condition is evaluated again. If it is still **true**, the action is executed again. Then, the condition is evaluated again. If it is still **true**, the action is executed again. Then, the condition is....eventually, this has to stop! Eventually, something in the action has to change something in the condition so it becomes **false**. The condition must eventually become **false** to avoid an ***infinite loop***. (If this happens, Ctrl-C will exit the loop.)

As an example of a conditional loop, we will write a function that will find the first ***factorial*** that is greater than the input argument *high*. For an integer n, the factorial of n, written as n!, is defined as $n! = 1 \times 2 \times 3 \times 4 \times \ldots \times n$. To calculate a factorial, a **for** loop would be used. However, in this case we do not know the value of n, so we have to keep calculating the next factorial until a level is reached, which means using a **while** loop.

The basic algorithm is to have two variables: one that iterates through the values 1,2, 3, and so on; and one that stores the factorial of the iterator at each step. We start with 1 and 1 factorial, which is 1. Then, we check the factorial. If it is not greater than *high*, the iterator variable will then increment to 2 and find its factorial (2). If this is not greater than *high*, the iterator will then increment to 3 and the function will find its factorial (6). This continues until we get to the first factorial that is greater than *high*.

So, the process of incrementing a variable and finding its factorial is repeated until we get to the first value greater than **high**. This is implemented using a **while** loop:

factgthigh.m

```
function facgt = factgthigh(high)
% factgthigh returns the first factorial > input
% Format: factgthigh(inputInteger)

i=0;
fac=1;
while fac <= high
     i=i+1;
     fac = fac * i;
end
facgt = fac;
end
```

An example of calling the function, passing 5000 for the value of the input argument *high*, follows:

```
>> factgthigh(5000)
ans =
        5040
```

The iterator variable *i* is initialized to 0, and the running product variable *fac*, which will store the factorial of each value of *i*, is initialized to 1. The first time the **while** loop is executed, the condition is **true**: 1 is less than or equal to 5000. So, the action of the loop is executed, which is to increment *i* to 1 and *fac* becomes 1 (1×1).

After the execution of the action of the loop, the condition is evaluated again. As it will still be **true**, the action is executed: *i* is incremented to 2 and *fac* will get the value 2 (1×2). The value 2 is still <= 5000, so the action will be executed again: *i* will be incremented to 3 and *fac* will get the value 6 (2×3). This continues, until the first value of *fac* is found that is greater than 5000. As soon as *fac* gets to this value, the condition will be **false** and the **while** loop will end. At that point the factorial is assigned to the output argument, which returns the value.

The reason that *i* is initialized to 0 rather than 1 is that the first time the loop action is executed, *i* becomes 1 and *fac* becomes 1, so we have 1 and 1!, which is 1.

5.3.1 Multiple Conditions in a while Loop

In the *factgthigh* function, the condition in the **while** loop consisted of one expression, which tested whether or not the variable *fac* was less than or equal to the variable *high*. In many cases, however, the condition will be more complicated than that and could use either the **or** operator || or the **and** operator &&. For example, it may be that it is desired to stay in a **while** loop as long as a variable *x* is in a particular range:

```
while x >= 0 && x <= 100
```

As another example, continuing the action of a loop may be desired as long as at least one of two variables is in a specified range:

```
while x < 50||y < 100
```

5.3.2 Reading from a File Using a while Loop

The following example illustrates reading from a data file using a **while** loop. Data from an experiment has been recorded in a file called "experd.dat". The file has some weights followed by a −99 and then more weights, all on the same

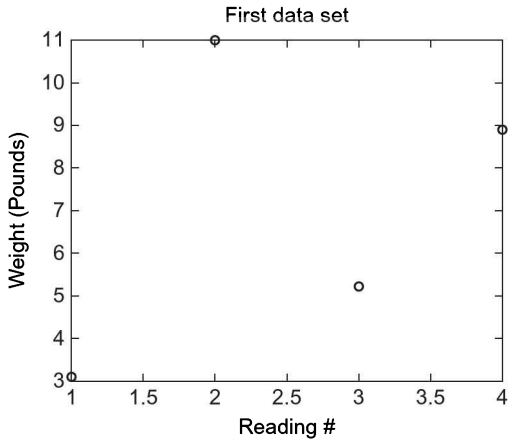

FIGURE 5.2
Plot of some (but not all) data from a file.

line. The only data values that we are interested in, however, are those before −99. The −99 is an example of a ***sentinel***, which is a marker in between data sets.

The algorithm for the script is as follows:

- Read the data from the file into a vector.
- Create a new vector variable *newvec* that only has the data values up to but not including the −99.
- Plot the new vector values, using black circles.

For example, *if* the file contains the following:

```
3.1  11  5.2  8.9  -99  4.4 62
```

The plot produced would look like Fig 5.2.

For simplicity, we will assume that the file format is as specified. Using **load** will create a vector with the name *experd*, which contains the values from the file.

THE PROGRAMMING CONCEPT

Using the programming method, we would loop through the vector until the −99 is found, creating the new vector by storing each element from *experd* in the vector *newvec*.

Continued

THE PROGRAMMING CONCEPT—CONT'D

findvalwhile.m

```
% Reads data from a file, but only plots the numbers
% up to a flag of -99. Uses a while loop.

load experd.dat

i = 1;
while experd(i) ~= -99
    newvec(i) = experd(i);
    i = i + 1;
end

plot(newvec,'ko')
xlabel('Reading #')
ylabel('Weight (pounds)')
title('First Data Set')
```

Note that this extends the vector *newvec* every time the action of the loop is executed.

THE EFFICIENT METHOD

Using the **find** function, we can locate the index of the element that stores the −99. Then, the new vector comprises all of the original vector from the first element to the index *before* the index of the element that stores the −99.

findval.m

```
% Reads data from a file, but only plots the numbers
% up to a flag of -99.Uses find and the colon operator

load experd.dat

Where = find(experd == -99);
newvec = experd(1:where-1);

plot(newvec,'ko')
xlabel('Reading #')
ylabel('Weight (pounds)')
title('First Data Set')
```

5.3.3 Input in a while Loop

Sometimes a **while** loop is used to process input from the user as long as the user is entering data in a correct format. The following script repeats the process of prompting the user, reading in a positive number, and echo printing it, as long as the user correctly enters positive numbers when prompted. As soon as the user types in a negative number, the script will print "OK" and end.

whileposnum.m

```
% Prompts the user and echo prints the numbers entered
% until the user enters a negative number

inputnum=input('Enter a positive number: ');
while inputnum >= 0
    fprintf('You entered a %d.\n\n',inputnum)
    inputnum = input('Enter a positive number: ');
end
fprintf('OK!\n')
```

When the script is executed, the input/output might look like this:

```
>> whileposnum
Enter a positive number: 6
You entered a 6.

Enter a positive number: -2
OK!
```

Note that the prompt is repeated in the script: once before the loop, and then again at the end of the action. This is done so that every time the condition is evaluated, there is a new value of *inputnum* to check. If the user enters a negative number the first time, no values would be echo printed:

Note

This example illustrates a very important feature of ***while*** *loops: it is possible that the action will not be executed at all, if the value of the condition is false the first time it is evaluated.*

```
>> whileposnum
Enter a positive number: -33
OK!
```

As we have seen previously, MATLAB will give an error message if a character is entered rather than a number.

```
>> whileposnum
Enter a positive number: a
Error using input
Undefined function or variable 'a'.

Error in whileposnum (line 4)
inputnum=input('Enter a positive number: ');

Enter a positive number: -4
OK!
```

However, if the character is actually the name of a variable, it will use the value of that variable as the input. For example:

```
>> a = 5;
>> whileposnum
Enter a positive number: a
You entered a 5.

Enter a positive number: -4
OK!
```

EXTENDING A VECTOR

If it is desired to store all of the positive numbers that the user enters, we would store them one at a time in a vector. However, as we do not know ahead of time how many elements we will need, we cannot preallocate to the correct size. The two methods of extending a vector one element at a time are shown here. We can start with an empty vector and concatenate each value to the vector, or we can increment an index.

```
numvec = [];
inputnum=input('Enter a positive number: ');
while inputnum >= 0
     numvec = [numvec inputnum];
     inputnum = input('Enter a positive number: ');
end

% OR:

i = 0;
inputnum=input('Enter a positive number: ');
while inputnum >= 0
     i = i+1;
     numvec(i) = inputnum;
     inputnum = input('Enter a positive number: ');
end
```

Keep in mind that this is inefficient and should be avoided if the array can be preallocated.

5.3.4 Counting in a While Loop

Although **while** loops are used when the number of times the action will be repeated is not known ahead of time, it is often useful to know how many times the action was, in fact, repeated. In that case, it is necessary to *count* the number of times the action is executed. The following variation on the previous script counts the number of positive numbers that the user successfully enters.

countposnum.m

```
% Prompts the user for positive numbers and echo prints as
% long as the user enters positive numbers

% Counts the positive numbers entered by the user
counter=0;
inputnum=input('Enter a positive number: ');
while inputnum >= 0
     fprintf('You entered a %d.\n\n',inputnum)
     counter = counter+1;
     inputnum = input('Enter a positive number: ');
end
fprintf('Thanks, you entered %d positive numbers.\n',counter)
```

The script initializes a variable *counter* to 0. Then, in the **while** loop action, every time the user successfully enters a number, the script increments the counter variable. At the end of the script it prints the number of positive numbers that were entered.

```
>> countposnum
Enter a positive number: 4
You entered a 4.

Enter a positive number: 11
You entered a 11.

Enter a positive number: -4
Thanks, you entered 2 positive numbers.
```

PRACTICE 5.5

Write a script *avenegnum* that will repeat the process of prompting the user for negative numbers, until the user enters a zero or positive number, as just shown. Instead of echo printing them, however, the script will print the average (of just the negative numbers). If no negative numbers are entered, the script will print an error message instead of the average. Use the programming method. Examples of executing this script follow:

```
>> avenegnum
Enter a negative number: 5
No negative numbers to average.

>> avenegnum
Enter a negative number: -8
Enter a negative number: -3
Enter a negative number: -4
Enter a negative number: 6
The average was -5.00
```

5.3.5 Error-Checking User Input in a while loop

In most applications, when the user is prompted to enter something, there is a valid range of values. If the user enters an incorrect value, rather than having the program carry on with an incorrect value, or just printing an error message, the program should repeat the prompt. The program should keep prompting the user, reading the value, and checking it until the user enters a value that is in the correct range. This is a very common application of a conditional loop: looping until the user correctly enters a value in a program. This is called *error-checking*.

For example, the following script prompts the user to enter a positive number, and loops to print an error message and repeat the prompt until the user finally enters a positive number.

readonenum.m

```
% Loop until the user enters a positive number

inputnum=input('Enter a positive number: ');
while inputnum < 0
    inputnum = input('Invalid! Enter a positive number: ');
end
fprintf('Thanks, you entered a %.1f \n',inputnum)
```

An example of running this script follows:

Note
MATLAB itself catches the character input and prints an error message, and repeats the prompt when the *c* was entered.

```
>> readonenum
Enter a positive number: -5
Invalid! Enter a positive number: -2.2
Invalid! Enter a positive number: c
Error using input
Undefined function or variable 'c'.
Error in readonenum (line 5)
 inputnum = input('Invalid! Enter a positive number: ');
Invalid! Enter a positive number: 44
Thanks, you entered a 44.0
```

QUICK QUESTION!

How could we vary the previous example so that the script asks the user to enter positive numbers *n* times, where *n* is an integer defined to be 3?

Answer: Every time the user enters a value, the script checks and in a **while** loop keeps telling the user that it's invalid until a valid positive number is entered. By putting the error-check in a **for** loop that repeats n times, the user is forced eventually to enter three positive numbers, as shown in the following.

readnnums.m

```
% Loop until the user enters n positive numbers
n=3;
fprintf('Please enter %d positive numbers\n\n',n)
for i=1:n
    inputnum=input('Enter a positive number: ');
    while inputnum < 0
        inputnum = input('Invalid! Enter a positive number: ');
    end
    fprintf('Thanks, you entered a %.1f \n',inputnum)
end
```

QUICK QUESTION!—CONT'D

```
>> readnnums
Please enter 3 positive numbers

Enter a positive number: 5.2
Thanks, you entered a 5.2
Enter a positive number: 6
Thanks, you entered a 6.0
Enter a positive number: -7.7
Invalid! Enter a positive number: 5
Thanks, you entered a 5.0
```

5.3.5.1 Error-Checking for Integers

As MATLAB uses the type **double** by default for all values, to check to make sure that the user has entered an integer, the program has to convert the input value to an integer type (e.g., **int32**) and then check to see whether that is equal to the original input. The following examples illustrate the concept.

If the value of the variable *num* is a real number, converting it to the type **int32** will round it, so the result is not the same as the original value.

```
>> num = 3.3;
>> inum = int32(num)
inum =
           3

>> num == inum
ans =
     0
```

If, however, the value of the variable *num* is an integer, converting it to an integer type will not change the value.

```
>> num = 4;
>> inum = int32(num)
inum =
           4

>> num == inum
 ans =
      1
```

The following script uses this idea to error-check for integer data; it loops until the user correctly enters an integer.

Note
This assumes that the user enters something. Use the **isempty** function to be sure.

readoneint.m

```
% Error-check until the user enters an integer
inputnum = input('Enter an integer: ');
num2 = int32(inputnum);
while num2 ~= inputnum
    inputnum = input('Invalid! Enter an integer: ');
    num2 = int32(inputnum);
end
fprintf('Thanks, you entered a %d \n',inputnum)
```

Examples of running this script are:

```
>> readoneint
Enter an integer: 9.5
Invalid! Enter an integer: 3.6
Invalid! Enter an integer: -11
Thanks, you entered a -11

>> readoneint
Enter an integer: 5
Thanks, you entered a 5
```

Putting these ideas together, the following script loops until the user correctly enters a positive integer. There are two parts to the condition, the value must be positive and must be an integer.

readoneposint.m

```
% Error checks until the user enters a positive integer
inputnum = input('Enter a positive integer: ');
num2 = int32(inputnum);
while num2 ~= inputnum || num2 < 0
    inputnum = input('Invalid! Enter a positive integer: ');
    num2 = int32(inputnum);
end
fprintf('Thanks, you entered a %d \n',inputnum)
```

```
>> readoneposint
Enter a positive integer: 5.5
Invalid! Enter a positive integer: -4
Invalid! Enter a positive integer: 11
Thanks, you entered a 11
```

PRACTICE 5.6

Modify the script *readoneposint* to read *n* positive integers, instead of just one.

5.4 LOOPS WITH VECTORS AND MATRICES; VECTORIZING

In most programming languages when performing an operation on a vector, a **for** loop is used to loop through the entire vector, using the loop variable as the index into the vector. In general, in MATLAB, assuming there is a vector variable *vec*, the indices range from 1 to the length of the vector, and the **for** statement loops through all of the elements performing the same operation on each one:

```
for i = 1:length(vec)
    % do something with vec(i)
end
```

In fact, this is one reason to store values in a vector. Typically, values in a vector represent "the same thing", so, typically in a program the same operation would be performed on every element.

Similarly, for an operation on a matrix, a nested loop would be required, and the loop variables over the rows and columns are used as the subscripts into the matrix. In general, assuming a matrix variable *mat*, we use **size** to return separately the number of rows and columns, and use these variables in the **for** loops. If an action is desired for every row in the matrix, the nested for loop would look like this:

```
[r, c] = size(mat);
for row = 1:r
   for col = 1:c
      % do something with mat(row,col)
   end
end
```

If, instead, an action is desired for every column in the matrix, the outer loop would be over the columns. (Note, however, that the reference to a matrix element always refers to the row index first and then the column index.)

```
[r, c] = size(mat);
for col = 1:c
   for row = 1:r
      % do something with mat(row,col)
   end
end
```

Typically, this is not necessary in MATLAB! Although **for** loops are very useful for many other applications in MATLAB, they are not typically used for operations on vectors or matrices; instead, the efficient method is to use built-in functions and/or operators. This is called vectorized code. The use of loops and selection statements with vectors and matrices is a basic programming concept with many other languages, and so both "the programming concept" and "the efficient method" are highlighted in this section and, to some extent, throughout the rest of this book.

5.4.1 Vectorizing Sums and Products

For example, let's say that we want to perform a scalar multiplication, in this case multiplying every element of a vector v by 3, and store the result back in v, where v is initialized as follows:

```
>> v= [3   7   2   1];
```

THE PROGRAMMING CONCEPT

To accomplish this, we can loop through all of the elements in the vector and multiply each element by 3. In the following, the output is suppressed in the loop, and then the resulting vector is shown:

```
>> for i = 1:length(v)
       v(i) = v(i) *3;
   end
>> v
v =
   9    21    6    3
```

THE EFFICIENT METHOD

```
>> v = v * 3
```

How could we calculate the factorial of n, n! = 1 * 2 * 3 * 4 *… * n?

THE PROGRAMMING CONCEPT

The basic algorithm is to initialize a running product to 1 and multiply the running product by every integer from 1 to n. This is implemented in a function:

myfact.m

```
function runprod = myfact(n)
% myfact returns n!
% Format of call: myfact(n)

runprod = 1;
for i = 1:n
    runprod = runprod*i;
end
end
```

THE PROGRAMMING CONCEPT—CONT'D

Any positive integer argument could be passed to this function, and it will calculate the factorial of that number. For example, if 5 is passed, the function will calculate and return 1*2*3*4*5, or 120:

```
>> myfact(5)
ans =
   120
```

THE EFFICIENT METHOD

MATLAB has a built-in function, **factorial**, that will find the factorial of an integer n. The **prod** function could also be used to find the product of the vector 1:5.

```
>> factorial(5)
ans =
   120
>> prod(1:5)
ans =
   120
```

QUICK QUESTION!

MATLAB has a **cumsum** function that will return a vector of all of the running sums of an input vector. However, many other languages do not, so how could we write our own?

Answer: Essentially, there are two programming methods that could be used to simulate the **cumsum** function. One method is to start with an empty vector and extend the vector by adding each running sum to it as the running sums are calculated. A better method is to preallocate the vector to the correct size and then change the value of each element to be successive running sums.

myveccumsum.m

```
function outvec = myveccumsum(vec)
% myveccumsum imitates cumsum for a vector
% It preallocates the output vector
% Format: myveccumsum(vector)

outvec = zeros(size(vec));
runsum = 0;
for i = 1:length(vec)
    runsum = runsum + vec(i);
    outvec(i) = runsum;
end
end
```

An example of calling the function follows:

```
>> myveccumsum([5   9   4])
ans =
     5   14   18
```

PRACTICE 5.7

Write a function that imitates the **cumprod** function. Use the method of preallocating the output vector.

QUICK QUESTION!

How would we sum each individual column of a matrix?

Answer: The programming method would require a nested loop in which the outer loop is over the columns. The function will sum each column and return a row vector containing the results.

matcolsum.m

```
function outsum = matcolsum(mat)
% matcolsum finds the sum of every column in a matrix
% Returns a vector of the column sums
% Format: matcolsum(matrix)

[row, col] = size(mat);

% Preallocate the vector to the number of columns
outsum = zeros(1,col);

% Every column is being summed so the outer loop
% has to be over the columns
for i = 1:col
    % Initialize the running sum to 0 for every column
    runsum = 0;
    for j = 1:row
        runsum = runsum+mat(j,i);
    end
    outsum(i) = runsum;
end
end
```

Note that the output argument will be a row vector containing the same number of columns as the input argument matrix. Also, as the function is calculating a sum for each column, the runsum variable must be initialized to 0 for every column, so it is initialized inside of the outer loop.

```
>> mat = [3:5; 2 5 7]
mat =
     3     4     5
     2     5     7
>> matcolsum(mat)
ans =
     5     9    12
```

Of course, the built-in **sum** function in MATLAB would accomplish the same thing, as we have already seen.

PRACTICE 5.8

Modify the function *matcolsum*. Create a function *matrowsum* to calculate and return a vector of all of the row sums instead of column sums. For example, calling it and passing the *mat* variable above would result in the following:

```
>> matrowsum(mat)
ans =
    12      14
```

5.4.2 Vectorizing Loops with Selection Statements

In many applications, it is useful to determine whether numbers in a matrix are positive, zero, or negative.

THE PROGRAMMING CONCEPT

A function *signum* follows that will accomplish this:

signum.m

```
function outmat = signum(mat)
% signum imitates the sign function
% Format: signum(matrix)

[r, c] = size(mat);
for i = 1:r
    for j = 1:c
        if mat(i,j) > 0
            outmat(i,j) = 1;
        elseif mat(i,j) == 0
            outmat(i,j) = 0;
        else
            outmat(i,j) = -1;
        end
    end
end
end
```

Here is an example of using this function:

```
>> mat = [0  4  -3;  -1  0  2]
mat =
     0     4    -3
    -1     0     2
```

Continued

THE PROGRAMMING CONCEPT—CONT'D

```
>> signum(mat)
ans =
     0     1    -1
    -1     0     1
```

THE EFFICIENT METHOD

Close inspection reveals that the function accomplishes the same task as the built-in sign function!

```
>> sign(mat)
ans =
     0     1    -1
    -1     0     1
```

Another example of a common application on a vector is to find the minimum and/or maximum value in the vector.

THE PROGRAMMING CONCEPT

For instance, the algorithm to find the minimum value in a vector is as follows:

- The working minimum (the minimum that has been found so far) is the first element in the vector to begin with.
- Loop through the rest of the vector (from the second element to the end).
  - If any element is less than the working minimum, then that element is the new working minimum.

The following function implements this algorithm and returns the minimum value found in the vector.

myminvec.m

```
function outmin = myminvec(vec)
% myminvec returns the minimum value in a vector
% Format: myminvec(vector)

outmin = vec(1);
for i = 2:length(vec)
    if vec(i) < outmin
        outmin = vec(i);
    end
end
end
```

THE PROGRAMMING CONCEPT—CONT'D

```
>> vec = [3   8   99   -1];
>> myminvec(vec)
ans =
    -1

>> vec = [3   8   99   11];
>> myminvec(vec)
ans =
     3
```

Note

An **if** statement is used in the loop rather than an **if-else** statement. If the value of the next element in the vector is less than *outmin*, then the value of *outmin* is changed; otherwise, no action is necessary.

THE EFFICIENT METHOD

Use the **min** function:

```
>> vec = [5   9   4];
>> min(vec)
ans =
     4
```

QUICK QUESTION!

Determine what the following function accomplishes:

xxx.m

```
function logresult = xxx(vec)
% QQ for you - what does this do?

logresult = false;
i = 1;
while i <= length(vec) && logresult == false
    if vec(i) ~= 0
       logresult = true;
    end
    i = i+1;
end
end
```

Answer: The output produced by this function is the same as the **any** function for a vector. It initializes the output argument to **false**. It then loops through the vector and, if any element is nonzero, changes the output argument to **true**. It loops until either a nonzero value is found or it has gone through all elements.

QUICK QUESTION!

Determine what the following function accomplishes.

yyy.m

```
function logresult = yyy(mat)
% QQ for you - what does this do?

count = 0;
[r, c] = size(mat);
for i = 1:r
    for j = 1:c
        if mat(i,j) ~= 0
            count = count + 1;
        end
    end
end

logresult = count == numel(mat);
end
```

Answer: The output produced by this function is the same as the **all** function.

As another example, we will write a function that will receive a vector and an integer as input arguments and will return a logical vector that stores **logical true** only for elements of the vector that are greater than the integer and **false** for the other elements.Note that as the vector was preallocated to **false**, the **else** clause is not necessary.

THE PROGRAMMING CONCEPT

The function receives two input arguments: the vector, and an integer *n* with which to compare. It loops through every element in the input vector, and stores in the result vector either **true** or **false** depending on whether vec(i) > n is **true** or **false**.

testvecgtn.m

```
function outvec = testvecgtn(vec,n)
% testvecgtn tests whether elements in vector
%     are greater than n or not
% Format: testvecgtn(vector, n)

% Preallocate the vector to logical false
outvec = false(size(vec));
for i = 1:length(vec)
    % If an element is > n, change to true
    if vec(i) > n
        outvec(i) = true;
    end
end
end
```

THE PROGRAMMING CONCEPT—CONT'D

```
>> ov = testvecgtn([44 2 11 -3 5 8], 6)
ov =
     1     0     1     0     0     1
>> class(ov)
ans =
logical
```

THE EFFICIENT METHOD

As we have seen, the relational operator > will automatically create a **logical** vector.

testvecgtnii.m

```
function outvec = testvecgtnii(vec,n)
% testvecgtnii tests whether elements in vector
%     are greater than n or not with no loop
% Format: testvecgtnii(vector, n)

outvec = vec > n;
end
```

PRACTICE 5.9

Call the function *testvecgtnii*, passing a vector and a value for n. Use MATLAB code to count how many values in the vector were greater than n.

5.4.3 TIPS FOR WRITING EFFICIENT CODE

To be able to write efficient code in MATLAB, including vectorizing, there are several important features to keep in mind:

- Scalar and array operations
- Logical vectors
- Built-in functions
- Preallocation of vectors

There are many functions in MATLAB that can be utilized instead of code that uses loops and selection statements. These functions have been demonstrated already but it is worth repeating them to emphasize their utility:

- **sum** and **prod**: find the sum or product of every element in a vector or column in a matrix

- **cumsum** and **cumprod**: return a vector or matrix of the cumulative (running) sums or products
- **min** and **max**: find the minimum value in a vector or in every column of a matrix
- **any**, **all**, **find**: work with logical expressions
- "is" functions, such as **isletter** and **isequal**: return **logical** values

In almost all cases, code that is faster to write by the programmer is also faster for MATLAB to execute. So, "efficient code" means that it is both efficient for the programmer and for MATLAB.

PRACTICE 5.10

Vectorize the following (rewrite the code efficiently):

```
i = 0;
for inc = 0: 0.5: 3
    i = i+1;
    myvec(i) = sqrt(inc);
end
----------------------------------------
[r c] = size(mat);
newmat = zeros(r,c);
for i = 1:r
    for j = 1:c
        newmat(i,j) = sign(mat(i,j));
    end
end
```

MATLAB has a built-in function **checkcode** that can detect potential problems within scripts and functions. Consider, for example, the following script that extends a vector within a loop:

badcode.m

```
for j = 1:4
    vec(j) = j
end
```

The function **checkcode** will flag this, as well as the good programming practice of suppressing output within scripts:

```
>> checkcode('badcode')
L 2 (C 5-7): The variable 'vec' appears to change size on every loop
iteration (within a script). Consider preallocating for speed.
L 2 (C 12): Terminate statement with semicolon to suppress output (within
a script).
```

The same information is shown in Code Analyzer Reports, which can be produced within MATLAB for one file (script or function) or for all code files within

a folder. Clicking on the down arrow for the Current Folder, and then choosing Reports and then Code Analyzer Report will check the code for all files within the Current Folder. When viewing a file within the Editor, click on the down arrow and then Show Code Analyzer Report for a report on just that one file.

5.5 TIMING

MATLAB has built-in functions that determine how long it takes code to execute. One set of related functions is **tic/toc**. These functions are placed around code, and will print the time it took for the code to execute. Essentially, the function **tic** turns a timer on, and then **toc** evaluates the timer and prints the result. Here is a script that illustrates these functions.

fortictoc.m

```
tic
mysum = 0;
for i = 1:20000000
    mysum = mysum+i;
end
toc
```

```
>> fortictoc
Elapsed time is 0.087294 seconds.
```

Here is an example of a script that demonstrates how much preallocating a vector speeds up the code.

Note
When using timing functions such as **tic/toc**, be aware that other processes running in the background (e.g., any web browser) will affect the speed of your code.

tictocprealloc.m

```
% This shows the timing difference between
% preallocating a vector vs. not

Clear
disp('No preallocation')
tic
for i = 1:10000
    x(i) = sqrt(i);
end
toc

disp('Preallocation')
tic
y = zeros(1,10000)
for i = 1:10000
    y(i) = sqrt(i);
end
toc
```

```
>> tictocprealloc
No preallocation
Elapsed time is 0.005070 seconds.
Preallocation
Elapsed time is 0.000273 seconds.
```

QUICK QUESTION!

Preallocation can speed up code, but to preallocate it is necessary to know the desired size. What if you do not know the eventual size of a vector (or matrix)? Does that mean that you have to extend it rather than preallocating?

Answer: If you know the maximum size that it could possibly be, you can preallocate to a size that is larger than necessary and then delete the "unused" elements. To do that, you would have to count the number of elements that are actually used. For example, if you have a vector *vec* that has been preallocated and a variable *count* that stores the number of elements that were actually used, this will trim the unnecessary elements:

```
vec = vec(1:count)
```

MATLAB also has a Profiler that will generate detailed reports on execution time of codes. In newer versions of MATLAB, from the Editor, click on Run and Time; this will bring up a report in the Profile Viewer. Choose the function name to see a very detailed report, including a Code Analyzer Report. From the Command Window, this can be accessed using **profile on** and **profile off**, and **profile viewer**.

```
>> profile on
>> tictocprealloc
No preallocation
Elapsed time is 0.047721 seconds.
Preallocation
Elapsed time is 0.040621 seconds.
>> profile viewer
>> profile off
```

■ Explore Other Interesting Features

Explore what happens when you use a matrix rather than a vector to specify the range in a **for** loop. For example,

```
for i = mat
    disp(i)
end
```

Take a guess before you investigate!

Try the **pause** function in loops.

Investigate the **vectorize** function.

The **tic** and **toc** functions are in the **timefun** help topic. Type **help timefun** to investigate some of the other timing functions. ■

SUMMARY

COMMON PITFALLS

- Forgetting to initialize a running sum or count variable to 0
- Forgetting to initialize a running product variable to 1
- In cases where loops are necessary, not realizing that if an action is required for every row in a matrix, the outer loop must be over the rows (and if an action is required for every column, the outer loop must be over the columns)
- Not realizing that it is possible that the action of a **while** loop will never be executed
- Not error-checking input into a program
- Forgetting to vectorize code whenever possible. If it is not necessary to use loops in MATLAB, don't!
- Forgetting that **subplot** numbers the plots rowwise rather than columnwise.
- Not realizing that the **subplot** function just creates a matrix within the Figure Window. Each part of this matrix must then be filled with a plot, using any type of plot function.

PROGRAMMING STYLE GUIDELINES

- Use loops for repetition only when necessary
  for statements as counted loops
  while statements as conditional loops
- Do not use *i* or *j* for iterator variable names if the use of the built-in constants **i** and **j** is desired.
- Indent the action of loops.
- If the loop variable is just being used to specify how many times the action of the loop is to be executed, use the colon operator 1:n where *n* is the number of times the action is to be executed.
- Preallocate vectors and matrices whenever possible (when the size is known ahead of time).
- When data are read in a loop, only store them in an array if it will be necessary to access the individual data values again.

MATLAB Reserved Words
for
while
end

MATLAB Functions and Commands	
subplot	profile
factorial	
checkcode	
tic/toc	

Exercises

1. Write a **for** loop that will print the column of real numbers from 2.7 to 3.5 in steps of 0.2.
2. In the Command Window, write a **for** loop that will iterate through the integers from 32 to 255. For each, show the corresponding character from the character encoding. Play with this! Try printing characters beyond the standard ASCII, in small groups. For example, print the characters that correspond to integers from 300 to 340.
3. Prompt the user for an integer *n* and print "I love this stuff!" n times.
4. When would it matter if a **for** loop contained `for i = 1:4` vs. `for i = [3 5 2 6]`, and when would it not matter?
5. Write a function *sumsteps2* that calculates and returns the sum of 1 to *n* in steps of 2, where *n* is an argument passed to the function. For example, if 11 is passed, it will return 1+3+5+7+9+11. Do this using a **for** loop. Calling the function will look like this:

```
>> sumsteps2(11)
ans =
    36
```

6. Write a function *prodby2* that will receive a value of a positive integer *n* and will calculate and return the product of the odd integers from 1 to *n* (or from 1 to *n-1* if *n* is even). Use a **for** loop.
7. Write a script that will:
   - generate a random integer in the inclusive range from 2 to 5
   - loop that many times to
     prompt the user for a number
     print the sum of the numbers entered so far with one decimal place
8. Write a script that will load data from a file into a matrix. Create the data file first, and make sure that there is the same number of values on every line in the file so that it can be loaded into a matrix. Using a **for** loop, it will then create a subplot

for every row in the matrix, and will plot the numbers from each row element in the Figure Window.

9. Write the code that will prompt the user for 4 numbers, and store them in a vector. Make sure that you preallocate the vector!
10. Write a **for** loop that will print the elements from a vector variable in sentence format, regardless of the length of the vector. For example, if this is the vector:

```
>> vec = [5.5  11  3.45];
```

this would be the result:

```
Element 1 is 5.50.
Element 2 is 11.00.
Element 3 is 3.45.
```

The **for** loop should work regardless of how many elements are in the vector.

11. Execute this script and be amazed by the results! You can try more points to get a clearer picture, but it may take a while to run.

```
clear
clf
x = rand;
y = rand;
plot(x,y)

hold on
for it = 1:10000
   choic = round(rand*2);
   if choic == 0
      x = x/2;
      y = y/2;
   elseif choic == 1
      x = (x+1)/2;
      y = y/2;
   else
      x = (x+0.5)/2;
      y = (y+1)/2;
   end
   plot(x,y)

   hold on
end
```

12. A machine cuts N pieces of a pipe. After each cut, each piece of pipe is weighed and its length is measured; these 2 values are then stored in a file called *pipe.dat* (first the weight and then the length on each line of the file). Ignoring units, the weight is supposed to be between 2.1 and 2.3, inclusive, and the length is supposed to be between 10.3 and 10.4, inclusive. The following is just the beginning of what will be a long script to work with these data. For now, the script will just count how many rejects there are. A reject is any piece of pipe that has an invalid

weight and/or length. For a simple example, if N is 3 (meaning three lines in the file) and the file stores:

```
2.14 10.30
2.32 10.36
2.20 10.35
```

There is only one reject, the second one, as it weighs too much. The script would print:
There were 1 rejects.

13. Come up with "trigger" words in a problem statement that would tell you when it's appropriate to use **for** loops and/or nested **for** loops.
14. With a matrix, when would:
    your outer loop be over the rows
    your outer loop be over the columns
    not matter which is the outer and which is the inner loop?
15. Write a function *myones* that will receive two input arguments *n* and *m* and will return an *nxm* matrix of all ones. Do NOT use any built-in functions (so, yes, the code will be inefficient). Here is an example of calling the function:
16. Write a script that will print the following multiplication table:

```
1
2  4
3  6  9
4  8  12  16
5  10  15  20  25
```

17. Write a function that will receive a matrix as an input argument, and will calculate and return the overall average of all numbers in the matrix. Use loops, not built-in functions, to calculate the average.
18. Write an algorithm for an ATM program. Think about where there would be selection statements, menus, loops (counted vs. conditional), etc. – but – don't write MATLAB code, just an algorithm (pseudo-code).
19. Trace this to figure out what the result will be, and then type it into MATLAB to verify the results.

```
count = 0;
number = 8;
while number > 3
     fprintf('number is %d\n', number)
     number = number - 2;
     count = count+1;
end
fprintf('count is %d\n', count)
```

20. The inverse of the mathematical constant *e* can be approximated as follows:

$$\frac{1}{e} \approx \left(1 - \frac{1}{n}\right)^n$$

Write a script that will loop through values of *n* until the difference between the approximation and the actual value is less than 0.0001. The script should then print out the built-in value of e^{-1} and the approximation to 4 decimal places, and also print the value of *n* required for such accuracy.

21. Write a script that will generate random integers in the range from 0 to 50, and print them, until one is finally generated that is greater than 25. The script should print how many attempts it took.
22. Write a script that will prompt the user for a keyword in MATLAB, error-checking until a keyword is entered.
23. A blizzard is a massive snowstorm. Definitions vary, but for our purposes we will assume that a blizzard is characterized by both winds of 30 mph or higher and blowing snow that leads to visibility of 0.5 miles or less, sustained for at least four hours. Data from a storm one day has been stored in a file *stormtrack.dat*. There are 24 lines in the file, one for each hour of the day. Each line in the file has the wind speed and visibility at a location. Create a sample data file. Read this data from the file and determine whether blizzard conditions were met during this day or not.
24. Given the following loop:

```
while x < 10
     action
end
```

    For what values of the variable *x* would the action of the loop be skipped entirely?

    If the variable *x* is initialized to have the value of 5 before the loop, what would the action have to include in order for this to not be an infinite loop?

25. Write a script called *prtemps* that will prompt the user for a maximum Celsius value in the range from −16 to 20; error-check to make sure it is in that range. Then, print a table showing degrees Fahrenheit and degrees Celsius until this maximum is reached. The first value that exceeds the maximum should not be printed. The table should start at 0 degrees Fahrenheit, and increment by 5 degrees Fahrenheit until the max (in Celsius) is reached. Both temperatures should be printed with a field width of 6 and one decimal place. The formula is C = 5/9 (F-32).
26. Vectorize this code! Write *one* assignment statement that will accomplish exactly the same thing as the given code (assume that the variable *vec* has been initialized):

```
result = 0;
for i = 1:length(vec)
    result = result+vec(i);
end
```

27. Vectorize this code! Write *one* assignment statement that will accomplish exactly the same thing as the given code (assume that the variable *vec* has been initialized):

```
newv = zeros(size(vec));
myprod = 1;
for i = 1:length(vec)
   myprod = myprod*vec(i);
   newv(i) = myprod;
end
newv % Note: this is just to display the value
```

28. The following code was written by somebody who does not know how to use MATLAB efficiently. Rewrite this as a single statement that will accomplish exactly the same thing for a matrix variable *mat* (e.g., vectorize this code):

```
[r c] = size(mat);
for i = 1:r
    for j = 1:c
        mat(i,j) = mat(i,j)*2;
   end
end
```

29. Vectorize the following code. Write one assignment statement that would accomplish the same thing. Assume that *mat* is a matrix variable that has been initialized.

```
[r,c] = size(mat);
val = mat(1,1);
for i = 1:r
    for j = 1:c
        if mat(i,j) < val
            val = mat(i,j);
        end
    end
end
val % just for display
```

30. Vectorize the following code. Write statement(s) that accomplish the same thing, eliminating the loop. Assume that there is a vector v that has a negative number in it, e.g:

```
>> v = [4 11 22 5 33 -8 3 99 52];
```

```
newv = [];
i = 1;
while v(i) >= 0
   newv(i) = v(i);
   i = i+1;
end
newv % Note: just to display
```

31. Give some examples of when you would need to use a counted loop in MATLAB, and when you would not.
32. For each of the following, decide whether you would use a **for** loop, a **while** loop, a nested loop (and if so what kind, e.g., a **for** loop inside of another **for** loop, a **while** loop inside of a **for** loop, etc.), or no loop at all. DO NOT WRITE THE ACTUAL CODE.

    sum the integers 1 through 50:
    add 3 to all numbers in a vector:
    prompt the user for a string and keep doing this until the string that the user enters is a keyword in MATLAB:
    find the minimum in every column of a matrix:
    prompt the user for 5 numbers and find their sum:
    prompt the user for 10 numbers, find the average and also find how many of the numbers were greater than the average:
    generate a random integer n in the range from 10 to 20. Prompt the user for n positive numbers, error-checking to make sure you get n positive numbers (and just echo print each one):
    prompt the user for positive numbers until the user enters a negative number. Calculate and print the average of the positive numbers, or an error message if none are entered:
33. Write a script that will prompt the user for a quiz grade and error-check until the user enters a valid quiz grade. The script will then echo print the grade. For this case, valid grades are in the range from 0 to 10 in steps of 0.5. Do this by creating a vector of valid grades and then use **any** or **all** in the condition in the **while** loop.
34. Which is faster: using **false** or using **logical**(0) to preallocate a matrix to all **logical** zeros? Write a script to test this.
35. Which is faster: using a **switch** statement or using a nested **if-else**? Write a script to test this.
36. Write a script *beautyofmath* that produces the following output. The script should iterate from 1 to 9 to produce the expressions on the left, perform the specified operation to get the results shown on the right and print exactly in the format shown here.

```
>> beautyofmath
1 x 8+1 = 9
12 x 8+2 = 98
123 x 8+3 = 987
1234 x 8+4 = 9876
12345 x 8+5 = 98765
123456 x 8+6 = 987654
1234567 x 8+7 = 9876543
12345678 x 8+8 = 98765432
123456789 x 8+9 = 987654321
```

37. The Wind Chill Factor (WCF) measures how cold it feels with a given air temperature T (in degrees Fahrenheit) and wind speed V (in miles per hour). One formula for WCF is

$$WCF = 35.7 + 0.6\,T - 35.7\left(V^{0.16}\right) + 0.43\,T\left(V^{0.16}\right)$$

Write a function to receive the temperature and wind speed as input arguments, and return the WCF. Using loops, print a table showing wind chill factors for temperatures ranging from −20 to 55 in steps of 5, and wind speeds ranging from 0 to 55 in steps of 5. Call the function to calculate each wind chill factor.

38. Instead of printing the WCFs in the previous problem, create a matrix of WCFs and write them to a file. Use the programming method, using nested loops.
39. Write a script that will prompt the user for N integers, and then write the positive numbers (>=0) to an ASCII file called *pos.dat* and the negative numbers to an ASCII file called *neg.dat*. Error-check to make sure that the user enters N integers.
40. Write a script to add two 30-digit numbers and print the result. This is not as easy as it might sound at first, because integer types may not be able to store a value this large. One way to handle large integers is to store them in vectors, where each element in the vector stores a digit of the integer. Your script should initialize two 30-digit integers, storing each in a vector, and then add these integers, also storing the result in a vector. Create the original numbers using the **randi** function. Hint: add 2 numbers on paper first, and pay attention to what you do!
41. Write a "Guess My Number Game" program. The program generates a random integer in a specified range, and the user (the player) has to guess the number. The program allows the user to play as many times as he/she would like; at the conclusion of each game, the program asks whether the player wants to play again.
    The basic algorithm is:
    1. The program starts by printing instructions on the screen.
    2. For every game:
        the program generates a new random integer in the range from MIN to MAX. Treat MIN and MAX like constants; start by initializing them to 1 and 100
        loop to prompt the player for a guess until the player correctly guesses the integer
        for each guess, the program prints whether the player's guess was too low, too high, or correct
        at the conclusion (when the integer has been guessed):
        print the total number of guesses for that game
        print a message regarding how well the player did in that game (e.g the player took way too long to guess the number, the player was awesome, etc.). To do this, you will have to decide on ranges for your messages and give a rationale for your decision in a comment in the program.

3. After all games have been played, print a summary showing the average number of guesses.

42. A CD changer allows you to load more than one CD. Many of these have random buttons, which allow you to play random tracks from a specified CD, or play random tracks from random CDs. You are to simulate a play list from such a CD changer using the **randi** function. The CD changer that we are going to simulate can load 3 different CDs. You are to assume that three CDs have been loaded. To begin with, the program should "decide" how many tracks there are on each of the three CDs, by generating random integers in the range from MIN to MAX. You decide on the values of MIN and MAX (look at some CDs; how many tracks do they have? What's a reasonable range?). The program will print the number of tracks on each CD. Next, the program will ask the user for his or her favorite track; the user must specify which track and which CD it's on. Next, the program will generate a "playlist" of the N random tracks that it will play, where N is an integer. For each of the N songs, the program will first randomly pick one of the 3 CDs, and then randomly pick one of the tracks from that CD. Finally, the program will print whether the user's favorite track was played or not. The output from the program will look something like this depending on the random integers generated and the user's input:

```
There are 15 tracks on CD 1.
There are 22 tracks on CD 2.
There are 13 tracks on CD 3.

What's your favorite track?
Please enter the number of the CD: 4
Sorry, that's not a valid CD.
Please enter the number of the CD: 1
Please enter the track number: 17
Sorry, that's not a valid track on CD 1.
Please enter the track number: -5
Sorry, that's not a valid track on CD 1.
Please enter the track number: 11

Play List:
CD 2 Track 20
CD 3 Track 11
CD 3 Track 8
CD 2 Track 1
CD 1 Track 7
CD 3 Track 8
CD 1 Track 3
CD 1 Track 15
CD 3 Track 12
CD 1 Track 6
Sorry, your favorite track was not played.
```

43. Write your own code to perform matrix multiplication. Recall that to multiply two matrices, the inner dimensions must be the same.

$$[A]_{m\,x\,n}[B]_{n\,x\,p} = [C]_{m\,x\,p}$$

Every element in the resulting matrix C is obtained by:

$$c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}.$$

So, three nested loops are required.

CHAPTER 6

MATLAB Programs

KEY TERMS

functions that return more than one value
functions that do not return any values
side effects
call-by-value
modular programs
main program
primary function
subfunction
menu-driven program
variable scope
base workspace
local variable
main function
global variable
persistent variable
declaring variables
bug
debugging
syntax errors
run-time errors
logical errors
tracing
breakpoints
breakpoint alley
function stubs
live script
code cells

CONTENTS

Chapter 3 introduced scripts and user-defined functions. In that chapter, we saw how to write scripts, which are sequences of statements that are stored in MATLAB code files and then executed. We also saw how to write user-defined functions, also stored in MATLAB code files that calculate and return a single value. In this chapter, we will expand on these concepts and introduce other kinds of user-defined functions. We will show how MATLAB® programs consist of combinations of scripts and user-defined functions. The mechanisms for interactions of variables in code files and the Command Window will be explored. Techniques for finding and fixing mistakes in programs will be reviewed. Finally, the use of live scripts created by the Live Editor (new as of R2016a), and using code cells in scripts will be introduced.

6.1 MORE TYPES OF USER-DEFINED FUNCTIONS

We have already seen how to write a user-defined function, stored in a code file, that calculates and returns one value. This is just one type of function. It is also possible for a function to return multiple values and it is possible for a function to return nothing. We will categorize functions as follows:

MATLAB®. http://dx.doi.org/10.1016/B978-0-12-804525-1.00006-4

- Functions that calculate and return one value
- Functions that calculate and return more than one value
- Functions that just accomplish a task, such as printing, without returning any values

Thus, although many functions calculate and return values, some do not. Instead, some functions just accomplish a task. Categorizing the functions as above is somewhat arbitrary, but there are differences between these three types of functions, including the format of the function headers and also the way in which the functions are called. Regardless of what kind of function it is, all functions must be defined and all function definitions consist of the ***header*** and the ***body***. Also, the function must be called for it to be utilized. All functions are stored in code files that have an extension of .m.

In general, any function in MATLAB consists of the following:

- The function header (the first line); this has:
  the reserved word **function**
  if the function ***returns*** values, the name(s) of the output argument(s), followed by the assignment operator (=)
  the name of the function (important: this should be the same as the name of the file in which this function is stored to avoid confusion)
  the input arguments in parentheses, if there are any (separated by commas if there is more than one).
- A comment that describes what the function does (this is printed if **help** is used).
- The body of the function, which includes all statements, including putting values in all output arguments if there are any.
- **end** at the end of the function.

6.1.1 Functions That Return More Than One Value

Functions that return one value have one output argument, as we saw previously. Functions that return more than one value must, instead, have more than one output argument in the function header in square brackets. That means that in the body of the function, values must be put in all output arguments listed in the function header. The general form of a function definition for a function that calculates and ***returns more than one value*** looks like this:

functionname.m

```
function [output arguments] = functionname(input arguments)
% Comment describing the function
% Format of function call

Statements here; these must include putting values in all of the output
arguments listed in the header

end
```

In the vector of output arguments, the output argument names are by convention separated by commas.

Choosing New, then Function brings up a template in the Editor that can then be filled in:

```
function [ output_args ] = untitled( input_args )
%UNTITLED Summary of this function goes here
%    Detailed explanation goes here


end
```

If this is not desired, it may be easier to start with New Script.

For example, here is a function that calculates two values, both the area and the circumference of a circle; this is stored in a file called *areacirc.m*:

areacirc.m

```
function [area, circum] = areacirc(rad)
% areacirc returns the area and
% the circumference of a circle
% Format: areacirc(radius)

area = pi * rad .* rad;
circum = 2 * pi * rad;
end
```

As this function is calculating two values, there are two output arguments in the function header (*area* and *circum*), which are placed in square brackets []. Therefore, somewhere in the body of the function, values have to be stored in both.

As the function is returning two values, it is important to capture and store these values in separate variables when the function is called. In this case, the first value returned, the area of the circle, is stored in a variable *a* and the second value returned is stored in a variable *c*:

```
>> [a, c] = areacirc(4)
a =
   50.2655
c =
   25.1327
```

If this is not done, only the first value returned is retained—in this case, the area:

```
>> disp(areacirc(4))
   50.2655
```

Note that in capturing the values, the order matters. In this example, the function first returns the area and then the circumference of the circle. The order in which values are assigned to the output arguments within the function, however, does not matter.

QUICK QUESTION!

What would happen if a vector of radii was passed to the function?

Answer: As the .* operator is used in the function to multiply *rad* by itself, a vector can be passed to the input argument *rad*. Therefore, the results will also be vectors, so the variables on the left side of the assignment operator would become vectors of areas and circumferences.

```
>> [a, c] = areacirc(1:4)
a =
   3.1416   12.5664   28.2743   50.2655
c =
   6.2832   12.5664   18.8496   25.1327
```

QUICK QUESTION!

What if you want only the second value that is returned?

Answer: Function outputs can be ignored using the tilde:

```
>> [~, c] = areacirc(1:4)
c =
  6.2832  12.5664  18.8496  25.1327
```

The **help** function shows the comment listed under the function header:

```
>> help areacirc
   This function calculates the area and
   the circumference of a circle
   Format: areacirc(radius)
```

The *areacirc* function could be called from the Command Window as shown here, or from a script. Here is a script that will prompt the user for the radius of just one circle, call the *areacirc* function to calculate and return the area and circumference of the circle, and print the results:

calcareacirc.m

```
% This script prompts the user for the radius of a circle,
%  calls a function to calculate and return both the area
%  and the circumference, and prints the results
% It ignores units and error-checking for simplicity

radius = input('Please enter the radius of the circle: ');
[area, circ] = areacirc(radius);
fprintf('For a circle with a radius of %.1f,\n', radius)
fprintf('the area is %.1f and the circumference is %.1f\n',...
    area, circ)
```

```
>> calcareacirc
Please enter the radius of the circle: 5.2
For a circle with a radius of 5.2,
the area is 84.9 and the circumference is 32.7
```

PRACTICE 6.1

Write a function *perimarea* that calculates and returns the perimeter and area of a rectangle. Pass the length and width of the rectangle as input arguments. For example, this function might be called from the following script:

calcareaperim.m

```
% Prompt the user for the length and width of a rectangle,
%  call a function to calculate and return the perimeter
%  and area, and print the result
% For simplicity it ignores units and error-checking
length = input('Please enter the length of the rectangle: ');
width = input('Please enter the width of the rectangle: ');
[perim, area] = perimarea(length, width);
fprintf('For a rectangle with a length of %.1f and a', length)
fprintf(' width of %.1f,\nthe perimeter is %.1f,', width, perim)
fprintf(' and the area is %.1f\n', area)
```

As another example, consider a function that calculates and returns three output arguments. The function will receive one input argument representing a total number of seconds and returns the number of hours, minutes, and remaining seconds that it represents. For example, 7515 total seconds is 2 h, 5 min, and 15 s because $7515 = 3600 \times 2 + 60 \times 5 + 15$.

The algorithm is as follows.

- Divide the total seconds by 3600, which is the number of seconds in an hour. For example, 7515/3600 is 2.0875. The integer part is the number of hours (e.g., 2).
- The remainder of the total seconds divided by 3600 is the remaining number of seconds; it is useful to store this in a local variable.
- The number of minutes is the remaining number of seconds divided by 60 (again, the integer part).
- The number of seconds is the remainder of the previous division.

breaktime.m

```
function [hours, minutes, secs] = breaktime(totseconds)
% breaktime breaks a total number of seconds into
% hours, minutes, and remaining seconds
% Format: breaktime(totalSeconds)

hours = floor(totseconds/3600);
remsecs = rem(totseconds, 3600);
minutes = floor(remsecs/60);
secs = rem(remsecs,60);
end
```

An example of calling this function is:

```
>> [h, m, s] = breaktime(7515)
h =
     2
m =
     5
s =
    15
```

As before, it is important to store all values that the function returns by using three separate variables.

6.1.2 Functions That Accomplish a Task Without Returning Values

Many functions do not calculate values but rather accomplish a task, such as printing formatted output. As these functions do not return any values, there are no output arguments in the function header.

The general form of a function definition for a ***function that does not return any values*** looks like this:

functionname.m

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

Statements here
end
```

Note what is missing in the function header: there are no output arguments and no assignment operator.

For example, the following function just prints the two arguments, numbers, passed to it in a sentence format:

printem.m

```
function printem(a,b)
% printem prints two numbers in a sentence format
% Format: printem(num1, num2)

fprintf('The first number is %.1f and the second is %.1f\n',a,b)
end
```

As this function performs no calculations, there are no output arguments in the function header and no assignment operator (=). An example of a call to the *printem* function is:

```
>> printem(3.3, 2)
The first number is 3.3 and the second is 2.0
```

Note that as the function does not return a value, it cannot be called from an assignment statement. Any attempt to do this would result in an error, such as the following:

```
>> x = printem(3, 5) % Error!!
Error using printem
Too many output arguments.
```

We can therefore think of the call to a function that does not return values as a statement by itself, in that the function call cannot be imbedded in another statement such as an assignment statement or an output statement.

The tasks that are accomplished by functions that do not return any values (e.g., output from an **fprintf** statement or a **plot**) are sometimes referred to as ***side effects***. Some standards for commenting functions include putting the side effects in the block comment.

PRACTICE 6.2

Write a function that receives a vector as an input argument and prints the individual elements from the vector in a sentence format.

```
>> printvecelems([5.9   33   11])
Element 1 is 5.9
Element 2 is 33.0
Element 3 is 11.0
```

6.1.3 Functions That Return Values Versus Printing

A function that calculates and ***returns*** values (through the output arguments) does not normally also print them; that is left to the calling script or function. It is a good programming practice to separate these tasks.

If a function just prints a value, rather than returning it, the value cannot be used later in other calculations. For example, here is a function that just prints the circumference of a circle:

calccircum1.m

```
function calccircum1(radius)
% calccircum1 displays the circumference of a circle
%    but does not return the value
% Format: calccircum1(radius)

disp(2 * pi * radius)
end
```

Calling this function prints the circumference, but there is no way to store the value so that it can be used in subsequent calculations:

```
>> calccircum1(3.3)
   20.7345
```

Since no value is returned by the function, attempting to store the value in a variable would be an error:

```
>> c = calccircum1(3.3)
Error using calccircum1
Too many output arguments.
```

By contrast, the following function calculates and returns the circumference, so that it can be stored and used in other calculations. For example, if the circle is the base of a cylinder and we wish to calculate the surface area of the cylinder, we would need to multiply the result from the *calccircum2* function by the height of the cylinder.

calccircum2.m

```
function circle_circum = calccircum2(radius)
%  calccircum2 calculates and returns the
%     circumference of a circle
%  Format: calccircum2(radius)

circle_circum = 2 * pi * radius;
end
```

```
>> circumference = calccircum2(3.3)
circumference =
   20.7345

>> height = 4;
>> surf_area = circumference * height
surf_area =
   82.9380
```

One possible exception to this rule of not printing when returning is to have a function return a value if possible but throw an error if not.

6.1.4 Passing Arguments to Functions

In all function examples presented thus far, at least one argument was passed in the function call to be the value(s) of the corresponding input argument(s) in the function header. The ***call-by-value*** method is the term for this method of passing the values of the arguments to the input arguments in the functions.

In some cases, however, it is not necessary to pass any arguments to the function. Consider, for example, a function that simply prints a random real number with two decimal places:

printrand.m

```
function printrand()
% printrand prints one random number
% Format: printrand or printrand()

fprintf('The random # is %.2f\n',rand)
end
```

Here is an example of calling this function:

```
>> printrand()
The random # is 0.94
```

As nothing is passed to the function, there are no arguments in the parentheses in the function call and none in the function header, either. The parentheses are not even needed in either the function or the function call, either. The following works as well:

printrandnp.m

```
function printrandnp
% printrandnp prints one random number
% Format: printrandnp or printrandnp()

fprintf('The random # is %.2f\n',rand)
end
```

```
>> printrandnp
The random # is 0.52
```

In fact, the function can be called with or without empty parentheses, whether or not there are empty parentheses in the function header.

This was an example of a function that did not receive any input arguments nor did it return any output arguments; it simply accomplished a task.

The following is another example of a function that does not receive any input arguments, but in this case, it does return a value. The function prompts the user for a string and returns the value entered.

stringprompt.m

```
function outstr = stringprompt
% stringprompt prompts for a string and returns it
% Format stringprompt or stringprompt()

disp('When prompted, enter a string of any length.')
outstr = input('Enter the string here: ', 's');
end
```

```
>> mystring = stringprompt
When prompted, enter a string of any length.
Enter the string here: Hi there

mystring =
Hi there
```

PRACTICE 6.3

Write a function that will prompt the user for a string of at least one character, loop to error-check to make sure that the string has at least one character and return the string.

QUICK QUESTION!

It is important that the number of arguments passed in the call to a function must be the same as the number of input arguments in the function header, even if that number is zero. Also, if a function returns more than one value, it is important to "capture" all values by having an equivalent number of variables in a vector on the left side of an assignment statement. Although it is not an error if there aren't enough variables, some of the values returned will be lost. The following question is posed to highlight this.

Given the following function header (note that this is just the function header, not the entire function definition):

```
function [outa, outb] = qq1(x, y, z)
```

Which of the following proposed calls to this function would be valid?

```
(a) [var1, var2] = qq1(a, b, c);
(b) answer = qq1(3, y, q);
(c) [a, b] = myfun(x, y, z);
(d) [outa, outb] = qq1(x, z);
```

Answer: The first proposed function call, (a), is valid. There are three arguments that are passed to the three input arguments in the function header, the name of the function is *qq1*, and there are two variables in the assignment statement to store the two values returned from the function. Function call (b) is valid, although only the first value returned from the function would be stored in *answer*; the second value would be lost. Function call (c) is invalid because the name of the function is given incorrectly. Function call (d) is invalid because only two arguments are passed to the function, but there are three input arguments in the function header.

6.2 MATLAB PROGRAM ORGANIZATION

Typically, a MATLAB program consists of a script that calls functions to do the actual work.

6.2.1 Modular Programs

A ***modular program*** is a program in which the solution is broken down into modules, and each is implemented as a function. The script that calls these functions is typically called the ***main program***.

To demonstrate the concept, we will use the very simple example of calculating the area of a circle. In Section 6.3 a much longer example will be given. For this example, there are three steps in the algorithm to calculate the area of a circle:

- Get the input (the radius)
- Calculate the area
- Display the results

In a modular program, there would be one main script (or, possibly a function instead) that calls three separate functions to accomplish these tasks:

- A function to prompt the user and read in the radius
- A function to calculate and return the area of the circle
- A function to display the results

As scripts and functions are all stored in code files that have an extension of .m, there would therefore be four separate code files altogether for this program; one script file and three function code files, as follows:

calcandprintarea.m

```
% This is the main script to calculate the
%   area of a circle
% It calls 3 functions to accomplish this
radius = readradius;
area = calcarea(radius);
printarea(radius,area)
```

readradius.m

```
function radius = readradius
% readradius prompts the user and reads the radius
% Ignores error-checking for now for simplicity
% Format: readradius or readradius()

disp('When prompted, please enter the radius in inches.')
radius = input('Enter the radius: ');
end
```

calcarea.m

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

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

printarea.m

```
function printarea(rad,area)
% printarea prints the radius and area
% Format: printarea(radius, area)

fprintf('For a circle with a radius of %.2f inches,\n',rad)
fprintf('the area is %.2f inches squared.\n',area)
end
```

When the program is executed, the following steps will take place:

- the script *calcandprintarea* begins executing
- *calcandprintarea* calls the *readradius* function
  readradius executes and returns the radius

- *calcandprintarea* resumes executing and calls the *calcarea* function, passing the radius to it
  calcarea executes and returns the area
- *calcandprintarea* resumes executing and calls the *printarea* function, passing both the radius and the area to it
  printarea executes and prints
- the script finishes executing

Running the program would be accomplished by typing the name of the script; this would call the other functions:

```
>> calcandprintarea
When prompted, please enter the radius in inches.
Enter the radius: 5.3
For a circle with a radius of 5.30 inches,
the area is 88.25 inches squared.
```

Note how the function calls and the function headers match up. For example:

readradius function:

```
function call: radius = readradius;
function header: function radius = readradius
```

In the *readradius* function call, no arguments are passed so there are no input arguments in the function header. The function returns one output argument so that is stored in one variable.

calcarea function:

```
function call: area = calcarea(radius);
function header: function area = calcarea(rad)
```

In the *calcarea* function call, one argument is passed in parentheses so there is one input argument in the function header. The function returns one output argument so that is stored in one variable.

printarea function:

```
function call: printarea(radius,area)
function header: function printarea(rad,area)
```

In the *printarea* function call, there are two arguments passed, so there are two input arguments in the function header. The function does not return anything, so the call to the function is a statement by itself; it is not in an assignment or output statement.

PRACTICE 6.4

Modify the *readradius* function to error-check the user's input to make sure that the radius is valid. The function should ensure that the radius is a positive number by looping to print an error message until the user enters a valid radius.

6.2.2 Subfunctions

Thus far, every function has been stored in a separate code file. However, it is possible to have more than one function in a given file. For example, if one function calls another, the first (calling) function would be the ***primary function*** and the function that is called is a ***subfunction***. These functions would both be stored in the same code file, first the primary function and then the subfunction. The name of the code file would be the same as the name of the primary function, to avoid confusion.

To demonstrate this, a program that is similar to the previous one, but calculates and prints the area of a rectangle, is shown here. The script, or main program, first calls a function that reads the length and width of the rectangle, and then calls a function to print the results. This function calls a subfunction to calculate the area.

rectarea.m

```
% This program calculates & prints the area of a rectangle

% Call a fn to prompt the user & read the length and width
[length, width] = readlenwid;
% Call a fn to calculate and print the area
printrectarea(length, width)
```

readlenwid.m

```
function [l,w] = readlenwid
% readlenwid reads & returns the length and width
% Format: readlenwid or readlenwid()

l = input('Please enter the length: ');
w = input('Please enter the width: ');
end
```

printrectarea.m

```
function printrectarea(len, wid)
% printrectarea prints the rectangle area
% Format: printrectarea(length, width)

% Call a subfunction to calculate the area
area = calcrectarea(len,wid);
fprintf('For a rectangle with a length of %.2f\n',len)
fprintf('and a width of %.2f, the area is %.2f\n', ...
     wid, area);
end

function area = calcrectarea(len, wid)
% calcrectarea returns the rectangle area
% Format: calcrectarea(length, width)
area = len * wid;
end
```

An example of running this program follows:

```
>> rectarea
Please enter the length: 6
Please enter the width: 3
For a rectangle with a length of 6.00
and a width of 3.00, the area is 18.00
```

Note how the function calls and function headers match up. For example:

readlenwid function:

```
function call: [length, width] = readlenwid;
function header: function [l,w] = readlenwid
```

In the *readlenwid* function call, no arguments are passed so there are no input arguments in the function header. The function returns two output arguments so there is a vector with two variables on the left side of the assignment statement in which the function is called.

printrectarea function:

```
function call: printrectarea(length, width)
function header: function printrectarea(len, wid)
```

In the *printrectarea* function call, there are two arguments passed, so there are two input arguments in the function header. The function does not return anything, so the call to the function is a statement by itself; it is not in an assignment or output statement.

calcrectarea subfunction:

```
function call: area = calcrectarea(len,wid);
function header: function area = calcrectarea(len, wid)
```

In the *calcrectarea* function call, two arguments are passed in parentheses so there are two input arguments in the function header. The function returns one output argument so that is stored in one variable.

The **help** command can be used with the script *rectarea*, the function *readlenwid*, and with the primary function, *printrectarea*. To view the first comment in the subfunction, as it is contained within the *printrectarea.m* file, the operator > is used to specify both the primary and subfunctions:

```
>> help rectarea
This program calculates & prints the area of a rectangle

>> help printrectarea
printrectarea prints the rectangle area
Format: printrectarea(length, width)

>> help printrectarea>calcrectarea
calcrectarea returns the rectangle area
Format: calcrectarea(length, width)
```

PRACTICE 6.5

For a right triangle with sides *a*, *b*, and *c*, where *c* is the hypotenuse and θ is the angle between sides *a* and *c*, the lengths of sides *a* and *b* are given by:

```
a = c * cos(θ)
b = c * sin(θ)
```

Write a script *righttri* that calls a function to prompt the user and read in values for the hypotenuse and the angle (in radians), and then calls a function to calculate and return the lengths of sides *a* and *b* and a function to print out all values in a sentence format. For simplicity, ignore units. Here is an example of running the script; the output format should be exactly as shown here:

```
>> righttri
Enter the hypotenuse: 5
Enter the angle: .7854
For a right triangle with hypotenuse 5.0
 and an angle 0.79 between side a & the hypotenuse,
 side a is 3.54 and side b is 3.54
```

For extra practice, do this using two different program organizations:

- One script that calls three separate functions
- One script that calls two functions; the function that calculates the lengths of the sides will be a subfunction to the function that prints

6.3 APPLICATION: MENU-DRIVEN MODULAR PROGRAM

Many longer, more involved programs that have interaction with the user are ***menu-driven***, which means that the program prints a menu of choices and then continues to loop to print the menu of choices until the user chooses to end the program. A modular menu-driven program would typically have a function that presents the menu and gets the user's choice, as well as functions to implement the action for each choice. These functions may have subfunctions. Also, the functions would error-check all user input.

As an example of such a menu-driven program, we will write a program to explore the constant *e*.

The constant *e*, called the natural exponential base, is used extensively in mathematics and engineering. There are many diverse applications of this constant. The value of the constant e is approximately 2.7183... Raising e to the power of

x, or e^x, is so common that this is called the exponential function. In MATLAB, as we have seen, there is a function for this, **exp**.

One way to determine the value of e is by finding a limit.

$$e = \lim_{n \to \infty} \left(1 + \frac{1}{n}\right)^n$$

As the value of n increases toward infinity, the result of this expression approaches the value of e.

An approximation for the exponential function can be found using what is called a Maclaurin series:

$$e^x \approx 1 + \frac{x^1}{1!} + \frac{x^2}{2!} + \frac{x^3}{3!} + \cdots$$

We will write a program to investigate the value of e and the exponential function. It will be menu-driven. The menu options will be:

- Print an explanation of e.
- Prompt the user for a value of n and then find an approximate value for e using the expression $(1 + 1/n)^n$.
- Prompt the user for a value for x. Print the value of **exp(x)** using the built-in function. Find an approximate value for e^x using the Maclaurin series just given.
- Exit the program.

The algorithm for the script main program follows:

- Call a function *eoption* to display the menu and return the user's choice.
- Loop until the user chooses to exit the program. If the user has not chosen to exit, the action of the loop is to:
  Depending on the user's choice, do one of the following:
  - Call a function *explaine* to print an explanation of *e*.
  - Call a function *limite* that will prompt the user for n and calculate an approximate value for e
  - Prompt the user for x and call a function *expfn* that will print both an approximate value for e^x and the value of the built-in **exp(x)**. Note that because any value for x is acceptable, the program does not need to error-check this value.

  Call the function *eoption* to display the menu and return the user's choice again.

The algorithm for the *eoption* function follows:

- Display the four choices.
- Error-check by looping to display the menu until the user chooses one of the four options.
- Return the integer value corresponding to the choice.

The algorithm for the *explaine* function is:

- Print an explanation of e, the **exp** function, and how to find approximate values.

The algorithm for the *limite* function is:

- Call a subfunction *askforn* to prompt the user for an integer *n*.
- Calculate and print the approximate value of e using *n*.

The algorithm for the subfunction *askforn* is:

- Prompt the user for a positive integer for *n*.
- Loop to print an error message and reprompt until the user enters a positive integer.
- Return the positive integer *n*.

The algorithm for the *expfn* function is:

- Receive the value of *x* as an input argument.
- Print the value of **exp(x)**.
- Assign an arbitrary value for the number of terms *n* (an alternative method would be to prompt the user for this).
- Call a subfunction *appex* to find an approximate value of **exp(x)** using a series with *n* terms.
- Print this approximate value.

The algorithm for the *appex* subfunction is:

- Receive *x* and *n* as input arguments.
- Initialize a variable for the running sum of the terms in the series (to 1 for the first term) and for a running product that will be the factorials in the denominators.
- Loop to add the *n* terms to the running sum.
- Return the resulting sum.

The entire program consists of the following script file and four function code files:

eapplication.m

```
% This script explores e and the exponential function

% Call a function to display a menu and get a choice
choice = eoption;

% Choice 4 is to exit the program
while choice ~= 4
   switch choice
       case 1
           % Explain e
           explaine;
       case 2
           % Approximate e using a limit
           limite;
       case 3
           % Approximate exp(x) and compare to exp
           x = input('Please enter a value for x: ');
           expfn(x);
   end
   % Display menu again and get user's choice
   choice = eoption;
end
```

eoption.m

```
function choice = eoption
% eoption prints a menu of options and error-checks
%   until the user chooses one of the options
% Format: eoption or eoption()

printchoices
choice = input('');
while ~any(choice == 1:4)
      disp('Error - please choose one of the options.')
      printchoices
      choice = input('');
end
end

function printchoices
fprintf('Please choose an option:\n\n');
fprintf('1) Explanation\n')
fprintf('2) Limit\n')
fprintf('3) Exponential function\n')
fprintf('4) Exit program\n\n')
end
```

explaine.m

```
function explaine
% explaine explains a little bit about e
% Format: explaine or explaine()

fprintf('The constant e is called the natural')
fprintf(' exponential base.\n')
fprintf('It is used extensively in mathematics and')
fprintf(' engineering.\n')
fprintf('The value of the constant e is ~ 2.7183\n')
fprintf('Raising e to the power of x is so common that')
fprintf(' this is called the exponential function.\n')
fprintf('An approximation for e is found using a limit.\n')
fprintf('An approximation for the exponential function')
fprintf(' can be found using a series.\n')
end
```

limite.m

```
function limite
% limite returns an approximate of e using a limit
% Format: limite or limite()

% Call a subfunction to prompt user for n
n = askforn;
fprintf('An approximation of e with n = %d is %.2f\n', ...
    n, (1 + 1/n) ^n)
end

function outn = askforn
% askforn prompts the user for n
% Format askforn or askforn()
% It error-checks to make sure n is a positive integer

inputnum = input('Enter a positive integer for n: ');
num2 = int32(inputnum);
while num2 ~= inputnum || num2 < 0
     inputnum = input('Invalid! Enter a positive integer: ');
     num2 = int32(inputnum);
end
outn = inputnum;
end
```

expfn.m

```
function expfn(x)
% expfn compares the built-in function exp(x)
%  and a series approximation and prints
% Format expfn(x)

fprintf('Value of built-in exp(x) is %.2f\n',exp(x))

% n is arbitrary number of terms
n = 10;
fprintf('Approximate exp(x) is %.2f\n', appex(x,n))
end

function outval = appex(x,n)
% appex approximates e to the x power using terms up to
%  x to the nth power
% Format appex(x,n)

% Initialize the running sum in the output argument
% outval to 1 (for the first term)
outval = 1;

for i = 1:n
   outval = outval + (x^i)/factorial(i);
end
end
```

Running the script will bring up the menu of options.

```
>> eapplication
Please choose an option:

(1) Explanation
(2) Limit
(3) Exponential function
(4) Exit program
```

Then, what happens will depend on which option(s) the user chooses. Every time the user chooses, the appropriate function will be called and then this menu will appear again. This will continue until the user chooses 4 for 'Exit Program.' Examples will be given of running the script, with different sequences of choices.

In the following example, the user

- Chose 1 for 'Explanation'
- Chose 4 for 'Exit Program'

```
>> eapplication
Please choose an option:
```

```
(1) Explanation
(2) Limit
(3) Exponential function
(4) Exit program

1
The constant e is called the natural exponential base.
It is used extensively in mathematics and engineering.
The value of the constant e is ~ 2.7183
Raising e to the power of x is so common that this is called the exponential
function.
An approximation for e is found using a limit.
An approximation for the exponential function can be found using a series.
Please choose an option:

(1) Explanation
(2) Limit
(3) Exponential function
(4) Exit program

4
```

In the following example, the user

- Chose 2 for 'Limit'
  When prompted for n, entered two invalid values before finally entering a valid positive integer
- Chose 4 for 'Exit Program'

```
>> eapplication
Please choose an option:

(1) Explanation
(2) Limit
(3) Exponential function
(4) Exit program

2
Enter a positive integer for n: -4
Invalid! Enter a positive integer: 5.5
Invalid! Enter a positive integer: 10
An approximation of e with n = 10 is 2.59
Please choose an option:

(1) Explanation
(2) Limit
(3) Exponential function
(4) Exit program

4
```

To see the difference in the approximate value for e as n increases, the user kept choosing 2 for 'Limit,' and entering larger and larger values each time in the following example (the menu is not shown for simplicity):

```
>> eapplication
Enter a positive integer for n: 4
An approximation of e with n = 4 is 2.44
Enter a positive integer for n: 10
An approximation of e with n = 10 is 2.59
Enter a positive integer for n: 30
An approximation of e with n = 30 is 2.67
Enter a positive integer for n: 100
An approximation of e with n = 100 is 2.70
```

In the following example, the user

- Chose 3 for 'Exponential function'
  When prompted, entered 4.6 for x
- Chose 3 for 'Exponential function' again
  When prompted, entered -2.3 for x
- Chose 4 for 'Exit Program'

Again, for simplicity, the menu options and choices are not shown.

```
>> eapplication
Please enter a value for x: 4.6
Value of built-in exp(x) is 99.48
Approximate exp(x) is 98.71
Please enter a value for x: -2.3
Value of built-in exp(x) is 0.10
Approximate exp(x) is 0.10
```

6.4 VARIABLE SCOPE

The ***scope*** of any variable is the workspace in which it is valid. The workspace created in the Command Window is called the ***base workspace***.

As we have seen before, if a variable is defined in any function, it is a ***local variable*** to that function, which means that it is only known and used within that function. Local variables only exist while the function is executing; they cease to exist when the function stops executing. For example, in the following function that calculates the sum of the elements in a vector, there is a local loop variable i.

mysum.m

```
function runsum = mysum(vec)
% mysum returns the sum of a vector
% Format: mysum(vector)

runsum = 0;
for i=1:length(vec)
    runsum = runsum + vec(i);
end
end
```

Running this function does not add any variables to the base workspace, as demonstrated in the following:

```
>> clear
>> who
>> disp(mysum([5  9  1]))
    15
>> who
>>
```

In addition, variables that are defined in the Command Window cannot be used in a function (unless passed as arguments to the function).

However, scripts (as opposed to functions) *do* interact with the variables that are defined in the Command Window. For example, the previous function is changed to be a script *mysumscript*.

mysumscript.m

```
% This script sums a vector
vec = 1:5;
runsum = 0;
for i=1:length(vec)
    runsum = runsum + vec(i);
end
disp(runsum)
```

The variables defined in the script do become part of the base workspace:

```
>> clear
>> who
>> mysumscript
    15
>> who
Your variables are:
i    runsum    vec
```

Variables that are defined in the Command Window can be used in a script, but cannot be used in a function. For example, the vector *vec* could be defined in the Command Window (instead of in the script), but then used in the script:

mysumscriptii.m

```
% This script sums a vector from the Command Window

runsum = 0;
for i=1:length(vec)
    runsum = runsum + vec(i);
end
disp(runsum)
```

```
>> clear
>> vec = 1:7;
>> who
Your variables are:
vec

>> mysumscriptii
    28
>> who
Your variables are:
i     runsum     vec
```

Note
This however, is very poor programming style. It is much better to pass the vector *vec* to a function.

Because variables created in scripts and in the Command Window both use the base workspace, many programmers begin scripts with a **clear** command to eliminate variables that may have already been created elsewhere (either in the Command Window or in another script).

Instead of a program consisting of a script that calls other functions to do the work, in some cases programmers will write a ***main function*** to call the other functions. So, the program consists of all functions rather than one script and the rest functions. The reason for this is again because both scripts and the Command Window use the base workspace. By using only functions in a program, no variables are added to the base workspace.

It is possible in MATLAB as well in other languages, to have ***global variables*** that can be shared by functions without passing them. Although there are some cases in which using **global** variables is efficient, it is generally regarded as poor programming style and therefore will not be explained further here.

6.4.1 Persistent Variables

Normally, when a function stops executing, the local variables from that function are cleared. That means that every time a function is called, memory is allocated and used while the function is executing, but released when it ends. With variables that are declared as ***persistent variables***, however, the value is not cleared so the next time the function is called, the variable still exists and retains its former value.