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
main function
local 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 (either in their own or in scripts) 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 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®. https://doi.org/10.1016/B978-0-12-815479-3.00006-4

- 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. Although functions can be stored in script code files, for now we will concentrate on functions that are stored in their own code files with 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

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 hours, 5 minutes, and 15 seconds because 7515 = 3600 * 2 + 60 * 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:

Note

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

functionname.m

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

Statements here
end
```

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 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. 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 (meaning, actually, a character vector) 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 is 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

Assuming that each is stored in a separate code file, there would 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 Local Functions

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 ***main function*** and the function that is called is a ***local function***, or sometimes a ***subfunction***. These functions would both be stored in the same code file, first the main function and then the local function. The name of the code file would be the same as the name of the main 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 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 local function 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 local function 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 main function, *printrectarea*. To view the first comment in the local function, as it is contained within the *printrectarea.m* file, the operator > is used to specify both the main and local functions:

```
>> 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)
```

So, local functions can be in script code files, and also in function code files.

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, each stored in separate code files
- One script that calls two functions; the function that calculates the lengths of the sides will be a local function 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 local functions. 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:

$$e^x \approx 1 + \frac{x^1}{1!} + \frac{x^2}{2!} + \frac{x^3}{3!} + \ldots$$

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 local function *askforn* to prompt the user for an integer *n*.
- Calculate and print the approximate value of *e* using *n*.

The algorithm for the local function *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 local function *appex* to find an approximate value of **exp(x)** using a series with *n* terms.
- Print this approximate value.

The algorithm for the local function *appex* 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 local function 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'

```
>> application
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 if 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
```

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 **clearvars** 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.

The following program demonstrates this. The script calls a function *func1*, which initializes a variable *count* to 0, increments it, and then prints the value. Every time this function is called, the variable is created, initialized to 0, changed to 1, and then cleared when the function exits. The script then calls a function *func2*, which first declares a **persistent** variable *count*. If the variable has not yet been initialized, which will be the case the first time the function is called, it is initialized to 0. Then, like the first function, the variable is incremented and the value is printed. With the second function, however, the variable remains with its value when the function exits, so the next time the function is called the variable is incremented again.

persistex.m

```
% This script demonstrates persistent variables

% The first function has a variable "count"
fprintf('This is what happens with a "normal" variable:\n')
func1
func1

% The second function has a persistent variable "count"
fprintf('\nThis is what happens with a persistent variable:\n')
func2
func2
```

func1.m

```
function func1
% func1 increments a normal variable "count"
% Format func1 or func1()

count = 0;
count = count + 1;
fprintf('The value of count is %d\n',count)
end
```

func2.m

```
function func2
% func2 increments a persistent variable "count"
% Format func2 or func2()

persistent count % Declare the variable

if isempty(count)
   count = 0;
end
count = count + 1;
fprintf('The value of count is %d\n',count)
end
```

The line

```
persistent count
```

declares the variable *count*, which allocates space for it, but does not initialize it. The **if** statement then initializes it (the first time the function is called). In many languages, variables always have to be declared before they can be used; in MATLAB, this is true only for **persistent** variables.

The functions can be called from the script or from the Command Window, as shown. For example, the functions are called first from the script. With the **persistent** variable, the value of *count* is incremented. Then, *func1* is called from the Command Window and *func2* is also called from the Command Window. As the value of the **persistent** variable had the value 2, this time it is incremented to 3.

```
>> persistex
This is what happens with a "normal" variable:
The value of count is 1
The value of count is 1

This is what happens with a persistent variable:
The value of count is 1
The value of count is 2

>> func1
The value of count is 1

>> func2
The value of count is 3
```

As can be seen from this, every time the function *func1* is called, whether from *persistex* or from the Command Window, the value of 1 is printed. However, with *func2* the variable *count* is incremented every time it is called. It is first called in this example from *persistex* twice, so *count* is 1 and then 2. Then, when

called from the Command Window, it is incremented to 3 (so it is counting how many times the function is called).

The way to restart a **persistent** variable is to use the **clear** function. The command

```
>> clear functions
```

will restart all **persistent** variables (see **help clear** for more options). However, as of Version 2015a, a better method has been established which is to clear an individual function rather than all of them, e.g.,

```
>> clear func2
```

PRACTICE 6.6

The following function *posnum* prompts the user to enter a positive number and loops to error-check. It returns the positive number entered by the user. It calls a subfunction in the loop to print an error message. The subfunction has a **persistent** variable to count the number of times an error has occurred. Here is an example of calling the function:

```
>> enteredvalue = posnum
Enter a positive number: -5
Error # 1 ...   Follow instructions!
Does -5.00 look like a positive number to you?
Enter a positive number: -33
Error # 2 ...   Follow instructions!
Does -33.00 look like a positive number to you?
Enter a positive number: 6
enteredvalue =
     6
```

Fill in the subfunction below to accomplish this.

posnum.m

```
function num = posnum
% Prompt user and error-check until the
% user enters a positive number
% Format posnum or posnum()

num = input('Enter a positive number: ');
while num < 0
      errorsubfn(num)
      num = input('Enter a positive number: ');
end
end

function errorsubfn(num)
% Fill this in

end
```

Of course, the numbering of the error messages will continue if the function is executed again without clearing it first.

6.5 DEBUGGING TECHNIQUES

Any error in a computer program is called a ***bug***. This term is thought to date back to the 1940s, when a problem with an early computer was found to have been caused by a moth in the computer's circuitry! The process of finding errors in a program, and correcting them, is still called ***debugging***.

As we have seen, the **checkcode** function can be used to help find mistakes or potential problems in script and function files.

6.5.1 Types of Errors

There are several different kinds of errors that can occur in a program, which fall into the categories of ***syntax errors***, ***runtime errors***, and ***logical errors***.

Syntax errors are mistakes in using the language. Examples of syntax errors are missing a comma or a quotation mark, or misspelling a word. MATLAB itself will flag syntax errors and give an error message. For example, the following character vector is missing the end quote:

```
>> mystr = 'how are you;
mystr = 'how are you;
        ↑
Error: Character vector is not terminated properly.
```

If this type of error is typed in a script or function using the Editor, the Editor will flag it.

Another common mistake is to spell a variable name incorrectly; MATLAB will also catch this error. Newer versions of MATLAB will typically be able to correct this for you, as in the following:

```
>> value = 5;
>> newvalue = valu + 3;
Unrecognized function or variable 'valu'.

Did you mean:
>> newvalue = value + 3;
```

Runtime, or execution-time, errors are found when a script or function is executing. With most languages, an example of a runtime error would be attempting to divide by zero. However, in MATLAB, this will return the constant Inf. Another example would be attempting to refer to an element in an array that does not exist.

runtimeEx.m

```
% This script shows an execution-time error

vec = 3:5;

for i = 1:4
    disp(vec(i))
end
```

Note

MATLAB gives an explanation of the error, and it gives the line number in the script in which the error occurred.

The previous script initializes a vector with three elements, but then attempts to refer to a fourth. Running it prints the three elements in the vector, and then an error message is generated when it attempts to refer to the fourth element.

```
>> runtimeEx
     3
     4
     5
Index exceeds array bounds.
Error in runtimeEx (line 6)
    disp(vec(i))
```

Logical errors are more difficult to locate because they do not result in any error message. A logical error is a mistake in reasoning by the programmer, but it is not a mistake in the programming language. An example of a logical error would be dividing by 2.54 instead of multiplying to convert inches to centimeters. The results printed or returned would be incorrect, but this might not be obvious.

All programs should be robust and should, wherever possible, anticipate potential errors and guard against them. For example, whenever there is input into a program, the program should error-check and make sure that the input is in the correct range of values. Also, before dividing, any denominator should be checked to make sure that it is not zero.

Despite the best precautions, there are bound to be errors in programs.

6.5.2 Tracing

Many times, when a program has loops and/or selection statements and is not running properly, it is useful in the debugging process to know exactly which statements have been executed. For example, the following is a function that

attempts to display "In middle of range" if the argument passed to it is in the range from 3 to 6, and "Out of range" otherwise.

testifelse.m

```
function testifelse(x)
% testifelse will test the debugger
% Format: testifelse(Number)

if 3 < x < 6
    disp('In middle of range')
else
    disp('Out of range')
end
end
```

However, it seems to print "In middle of range" for all values of *x*:

```
>> testifelse(4)
In middle of range

>> testifelse(7)
In middle of range

>> testifelse(-2)
In middle of range
```

One way of following the flow of the function, or ***tracing*** it, is to use the **echo** function. The **echo** function, which is a toggle, will display every statement as it is executed as well as results from the code. For scripts, just **echo** can be typed, but for functions, the name of the function must be specified. For example, the general form is

```
echo functionname on/off
```

For the *testifelse* function, it can be called as:

```
>> echo testifelse on
>> testifelse(-2)
% This function will test the debugger
if 3 < x < 6
     disp('In middle of range')
In middle of range
end
```

We can see from this result that the action of the **if** clause was executed.

6.5.3 Editor/Debugger

MATLAB has many useful functions for debugging, and debugging can also be done through its Editor, which is more properly called the Editor/Debugger.

Typing **help debug** at the prompt in the Command Window will show some of the debugging functions. Also, in the Help Documentation, typing "debugging" in the Search Documentation will display basic information about the debugging processes.

It can be seen in the previous example that the action of the **if** clause was executed and is printed "In middle of range", but just from that it cannot be determined why this happened. There are several ways to set ***breakpoints*** in a file (script or function) so that the variables or expressions can be examined. These can be done from the Editor/Debugger or commands can be typed from the Command Window. For example, the following **dbstop** command will set a breakpoint in the sixth line of this function (which is the action of the **if** clause), which allows the values of variables and/or expressions to be examined at that point in the execution. The function **dbcont** can be used to continue the execution, and **dbquit** can be used to quit the debug mode. Note that the prompt becomes K>> in debug mode.

```
>> dbstop testifelse 6
>> testifelse(-2)
5        disp('In middle of range')
K>> x
x =
     -2
K>> 3 < x
ans =
     0
K>> 3 < x < 6
ans =
     1
K>> dbcont
In middle of range
end
>>
```

By typing the expressions $3 < x$ and then $3 < x < 6$, we can determine that the expression $3 < x$ will return either 0 or 1. Both 0 and 1 are less than 6, so the expression will always be **true**, regardless of the value of x! Once in the debug mode, instead of using **dbcont** to continue the execution, **dbstep** can be used to step through the rest of the code one line at a time.

Breakpoints can also be set and cleared through the Editor. When a file is open in the Editor, in between the line numbers on the left and the lines of code is a thin gray strip which is the ***breakpoint alley***. In this, there are underscore marks next to the executable lines of code (as opposed to comments, for example). Clicking the mouse in the alley next to a line will create a breakpoint at that line (and then clicking on the red dot that indicates a breakpoint will clear it).

PRACTICE 6.7

The following script is bad code in several ways. Use **checkcode** first to check it for potential problems, and then use the techniques described in this section to set breakpoints and check values of variables.

debugthis.m

```
for i = 1:5
    i = 3;
    disp(i)
end

for j = 2:4
    vec(j) = j
end
```

6.5.4 Function Stubs

Another common debugging technique that is used when there is a script main program that calls many functions is to use ***function stubs***. A function stub is a place holder, used so that the script will work even though that particular function hasn't been written yet. For example, a programmer might start with a script main program, which consists of calls to three functions that accomplish all of the tasks.

mainscript.m

```
% This program gets values for x and y, and
%    calculates and prints z

[x, y] = getvals;
z = calcz(x,y);
printall(x,y,z)
```

The three functions have not yet been written, however, so function stubs are put in place so that the script can be executed and tested. The function stubs consist of the proper function headers, followed by a simulation of what the function will eventually do. For example, the first two functions put arbitrary values in for the output arguments, and the last function prints.

getvals.m

```
function [x, y] = getvals
x = 33;
y = 11;
end
```

calcz.m

```
function z = calcz(x,y)
z = x + y;
end
```

printall.m

```
function printall(x,y,z)
disp(x)
disp(y)
disp(z)
end
```

Then, the functions can be written and debugged one at a time. It is much easier to write a working program using this method than to attempt to write everything at once—then, when errors occur, it is not always easy to determine where the problem is!

6.6 LIVE SCRIPTS, CODE CELLS, AND PUBLISHING CODE

An entirely new type of script has been introduced in MATLAB as of Version 2016a. The script is called a ***live script*** and is created using the ***Live Editor***. A live script is much more dynamic than a simple script; it can embed equations, images, and hyperlinks in addition to formatted text. Instead of having graphs in separate windows, the graphics are shown next to the code that created them. It is also possible to put the graphs inline, under the code. As of R2016b, equation editors have been added, and equations can be entered in LaTeX format.

The scripts that we have seen so far have been simple scripts, stored in files that have an extension of .m. Live scripts are instead stored using the .mlx file format.

An example of a live script is shown in Figure 6.1. In this live script named "sintest.mlx", there is text, followed by an equation, then code, more text, another equation, and more code. All of these could be in separate sections, but sections can also include the different types of elements. The sections are created by clicking on "code", "text", "equation", and so forth from the "Insert" section, with "section break" in between each section. All output, including error messages if there are any, are shown to the right of the code. In this case, the graphs produced by the code sections are shown to the right of the code. Clicking on the icon above the graphs will move them inline.

There are several ways to create a live script. The simplest is to click on New Live Script, or on New, then Live Script. A simple script can also be transformed into a live script by choosing Save As and then Live Script. Right clicking on a set of

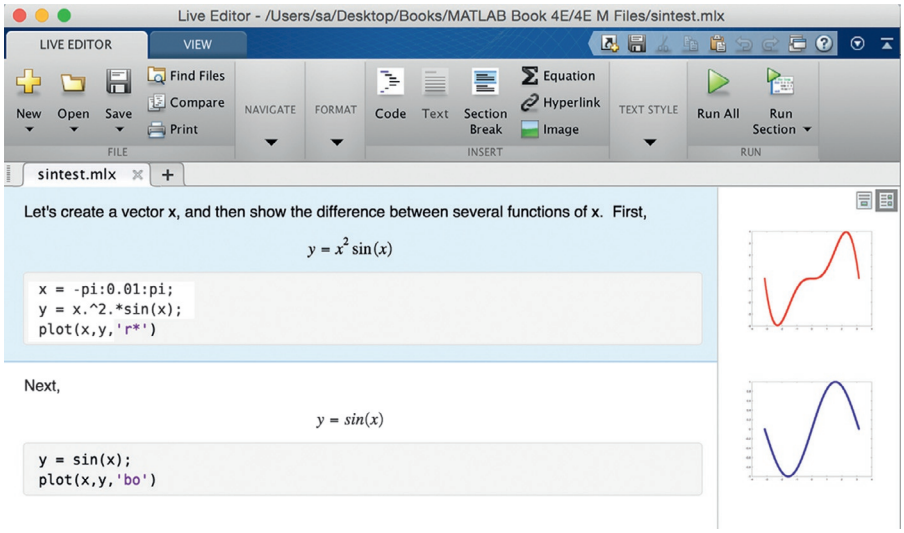

FIGURE 6.1
The Live Editor.

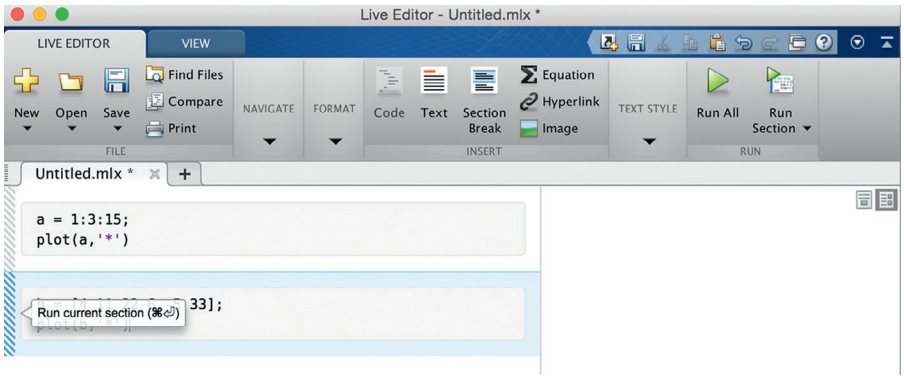

FIGURE 6.2
"Run current section" from Live Editor.

commands from the Command History window also pops up an option to save as a Live Script.

All of the code from the live script can be executed by choosing the Run All button. Alternatively, individual sections can be executed by clicking on the bar to the left of the section as seen in Figure 6.2.

Functions can be included in Live Scripts, as of R2016b. When a function is entered into a code section of a live script, a section break is automatically created before the function, as seen in Figure 6.3. The function must appear after all of the script code.

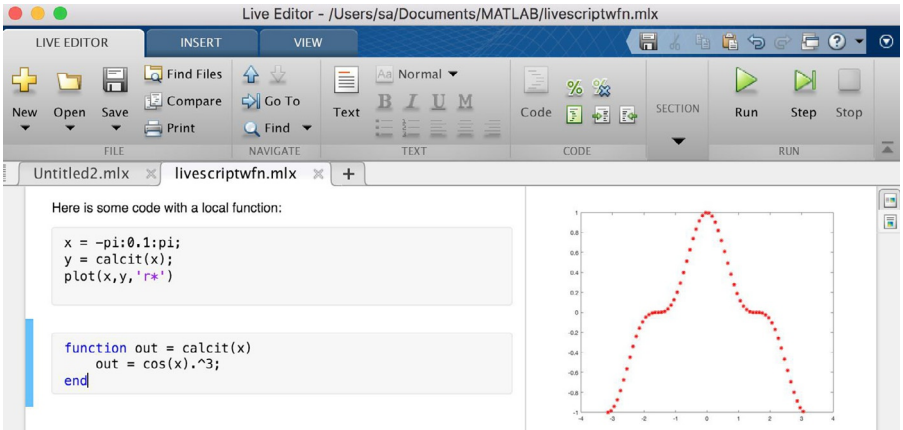

FIGURE 6.3
Function in Live Script.

Once a live script has been completed, it can be shared with others as an .mlx file, or it can be converted to a PDF or HTML format. To do this, click on the down arrow under "Save", and then either "Export to PDF" or "Export to HTML".

Live scripts can also be converted to code files with the .m extension by choosing Save As and then choosing MATLAB Code file from the drop down menu for the Format.

Using the **type** command on a live script will show just the code sections. The entire contents of the .mlx file can be seen from the Live Editor.

```
>> type sintest.mlx
x = -pi:0.01:pi;
y = x.^2.*sin(x);
plot(x,y,'r*')
y = sin(x);
plot(x,y,'bo')
```

PRACTICE 6.8

If you have R2016a or later, try creating a live script with text, equations, and code to produce at least one plot.

6.6.1 Code Cells

With simple code file scripts, one can break the code into sections, called ***code cells***. With code cells, you can run one cell at a time and you can also publish the code in an HTML format with plots embedded and with formatted equations.

To break code into cells, create comment lines that begin with two % symbols; these become the cell titles. For example, a script from Chapter 3 that plots sin and cos has been modified to have two cells: one that creates vectors for sin(x) and cos(x) and plots them, and a second that adds a legend, title, and axis labels to the plot.

sinncosCells.m

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

%% Create vectors and plot
clf
x = 0: 2*pi/40: 2*pi;
y = sin(x);
plot(x,y,'ro')
hold on
y = cos(x);
plot(x,y,'b+')

%% Add legends, axis labels, and title
legend('sin', 'cos')
xlabel('x')
ylabel('sin(x) or cos(x)')
title('sin and cos on one graph')
```

When viewing this script in the Editor, the individual cells can be chosen by clicking the mouse anywhere within the cell. This will highlight the cell with a background color. Then, from the Editor tab, you can choose "Run Section" to run just that one code cell and remain within that cell, or you can choose "Run and Advance" to run that code cell and then advance to the next.

By choosing the "Publish" tab and then "Publish", the code is published by default in HTML document. For the *sinncosCells* script, this creates a document in which there is a table of contents (consisting of the two cell titles); the first code block which plots, followed by the actual plot, and then the second code block that annotates the Figure Window, followed by the modified plot.

■ Explore Other Interesting Features

From the Command Window, type **help debug** to find out more about the debugging, and **help dbstop** in particular to find out more options for stopping code. Breakpoints can be set at specified locations in a file, only when certain condition(s) apply, and when errors occur.

Investigate the **dbstatus** function.

Explore the use of the functions **mlock** and **munlock** to block functions from being cleared using **clear**.

As of R2017a, it is possible to interactively edit figures in the Live Editor. Create Live script with a plot. Click on the plot, zoom in, and click on the "Update Code" button to modify the x and y limits.

It is also possible to create code cells in functions. Investigate this. ■

SUMMARY

COMMON PITFALLS

- Not matching up arguments in a function call with the input arguments in a function header.
- Not having enough variables in an assignment statement to store all of the values returned by a function through the output arguments.
- Attempting to call a function that does not return a value from an assignment statement, or from an output statement.
- Not using the same name for the function and the file in which it is stored.
- Not thoroughly testing functions for all possible inputs and outputs.
- Forgetting that **persistent** variables are updated every time the function in which they are declared is called—whether from a script or from the Command Window.

PROGRAMMING STYLE GUIDELINES

- If a function is calculating one or more values, return these value(s) from the function by assigning them to output variable(s).
- Give the function and the file in which it is stored the same name.
- Function headers and function calls must correspond. The number of arguments passed to a function must be the same as the number of input arguments in the function header. If the function returns values, the number of variables in the left side of an assignment statement should match the number of output arguments returned by the function.
- If arguments are passed to a function in the function call, do not replace these values by using **input** in the function itself.
- Functions that calculate and return value(s) will not normally also print them.
- Functions should not normally be longer than one page in length.
- Do not declare variables in the Command Window and then use them in a script, or vice versa.

- Pass all values to be used in functions to input arguments in the functions.
- When writing large programs with many functions, start with the main program script and use function stubs, filling in one function at a time while debugging.

MATLAB Reserved Words	
global	persistent

MATLAB Functions and Commands	
echo	dbquit
dbstop	dbstep
dbcont	

MATLAB Operator	
> path for subfunction	%% code cell title

Exercises

1. Given the following function header:

```
function [x, y] = calcem(a, b, c)
```

Which of the following function calls would be valid—and why?

```
[num, val] = calcem(4, 11, 2)
result = calcem(3, 5, 7)
```

2. Write a function that will receive as an input argument a number of kilometers (K). The function will convert the kilometers to miles and to US nautical miles, and return both results. The conversions are: 1K = 0.621 miles and 1 US nautical mile = 1.852K.
3. Write a function that receives an input argument *x* and returns *y* and *z*, which should be calculated as follows:

$y = 2x^2$

$z = 3x - 4y$

4. Write a function *splitem* that will receive one vector of numbers as an input argument and will return two vectors: one with the positive (>=0) numbers from the original vector, and the second, the negative numbers from the original vector. Use vectorized code (no loops) in your function.

5. Given the following function header:

```
function doit(a, b)
```

Which of the following function calls would be valid—and why?

```
fprintf('The result is %.1f\n', doit(4,11))
```

```
doit(5, 2, 11.11)
```

```
x = 11;
y = 3.3;
doit(x,y)
```

6. Write a function that receives an input argument and returns its length and first element.
7. We are going to write a program to create a word puzzle. For now, what we need is a function *promptletnum* that will prompt the user for one letter and an integer in the range from 1 to 5 (inclusive), error-checking until the user enters valid values, and then return them. The function will convert upper case letters to lower case. This function might be called from the following script *wordpuz*:

wordpuz.m

```
[l, n] = promptletnum;
fprintf('We will find words that contain the \n')
fprintf('letter ''%c'' %d times.\n', l, n)
```

8. Write a function that prints the area and circumference of a circle for a given radius. Only the radius is passed to the function. The function does not return any values. The area is given by πr^2 and the circumference is $2 \pi r$.
9. Write a function that will receive an integer *n* and a character as input arguments and will print the character *n* times.
10. Write a function that receives a matrix as an input argument and prints a random column from the matrix.
11. Write a function that receives a count as an input argument and prints the value of the count in a sentence that would read "It happened 1 time." if the value of the count is 1, or "It happened xx times." if the value of count (xx) is greater than 1.
12. Write a function that receives an *x* vector, a minimum value, and a maximum value and plots **sin(x)** from the specified minimum to the specified maximum.
13. Write a function that prompts the user for a value of an integer *n* and returns the value of *n*. No input arguments are passed to this function. Error-check to make sure that an integer is entered.
14. A cubit is an ancient unit of length, roughly the distance between one's fingers and elbow. The definition of one cubit is 18 inches. Write a program consisting of one script and one function. The script does the following:

- prompts the user for the number of cubits
- calls a function *cubit2inch* to convert this number to inches
- prints the result

15. Write a script that will:
    - Call a function to prompt the user for an angle in degrees
    - Call a function to calculate and return the angle in radians . (Note: π radians = 180 degrees)
    - Call a function to print the result

    Write all of the functions, also. Put the script and all functions in separate code files.
16. Modify the program in Exercise 15 so that the function to calculate the angle is a local function to the function that prints.
17. In 3D space, the Cartesian coordinates (*x*,*y*,*z*) can be converted to spherical coordinates (radius *r*, inclination θ, azimuth ϕ) by the following equations:

$$r = \sqrt{x^2 + y^2 + z^2}, \quad \theta = \cos^{-1}\left(\frac{z}{r}\right), \quad \phi = \tan^{-1}\left(\frac{y}{x}\right)$$

A program is being written to read in Cartesian coordinates, convert to spherical, and print the results. So far, a script *pracscript* has been written that calls a function *getcartesian* to read in the Cartesian coordinates and a function *printspherical* that prints the spherical coordinates. Assume that the *getcartesian* function exists and reads the Cartesian coordinates from a file. The function *printspherical* calls a local function *convert2spher* that converts from Cartesian to spherical coordinates. You are to write the *printspherical* function. Here is an example:

```
>> pracscript
The radius is 5.46
The inclination angle is 1.16
The azimuth angle is 1.07
```

pracscript.m

```
[x,y,z] = getcartesian();
printspherical(x,y,z)
```

getcartesian.m

```
function [x,y,z] = getcartesian()
% Assume this gets x,y,z from a file
end
```

18. The lump sum S to be paid when interest on a loan is compounded annually is given by $S = P(1 + i)^n$, where *P* is the principal invested, i is the interest rate, and *n* is the number of years. Write a program that will plot the amount *S* as it increases through the years from 1 to *n*. The main script will call a function to prompt the user for the number of years (and error-check to make sure that the user enters a positive integer). The script will then call a function that will plot *S* for years 1 through *n*. It will use 0.05 for the interest rate and $10,000 for *P*.
19. Write a program to write a length conversion chart to a file. It will print lengths in feet from 1 to an integer specified by the user in one column, and the corresponding length in meters (1 foot = 0.3048 m) in a second column. The main script will call one function that prompts the user for the maximum length in feet; this function must error-check to make sure that the user enters a valid positive integer. The script then calls a function to write the lengths to a file.
20. The script *circscript* loops n times to prompt the user for the circumference of a circle (where n is a random integer). Error-checking is ignored to focus on functions in this program. For each, it calls one function to calculate the radius and area of that circle, and then calls another function to print these values. The formulas are $r = c/(2\pi)$ and $a = \pi r^2$ where r is the radius, c is the circumference, and a is the area. Write the two functions.

circscript.m

```
n = randi(4);
for i = 1:n
    circ = input('Enter the circumference of the circle: ');
    [rad, area] = radarea(circ);
   dispra(rad,area)
end
```

21. The distance between any two points (x_1,y_1) and (x_2,y_2) is given by:

$$\text{distance} = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$$

The area of a triangle is:

$$\text{area} = \sqrt{s^*(s-a)^*(s-b)^*(s-c)}$$

where *a*, *b*, and *c* are the lengths of the sides of the triangle, and s is equal to half the sum of the lengths of the three sides of the triangle. Write a script that will prompt the user to enter the coordinates of three points that determine a triangle (e.g., the *x* and *y* coordinates of each point). The script

will then calculate and print the area of the triangle. It will call one function to calculate the area of the triangle. This function will call a local function that calculates the length of the side formed by any two points (the distance between them).

22. Write a program to write a temperature conversion chart to a file. The main script will:
    - call a function that explains what the program will do
    - call a function to prompt the user for the minimum and maximum temperatures in degrees Fahrenheit, and return both values. This function checks to make sure that the minimum is less than the maximum and calls a local function to swap the values if not.
    - call a function to write temperatures to a file: the temperature in degrees F from the minimum to the maximum in one column and the corresponding temperature in degrees Celsius in another column. The conversion is C = (F – 32) * 5/9.
23. Modify the function *func2* from Section 6.4.1 that has a **persistent** variable *count*. Instead of having the function print the value of *count*, the value should be returned.
24. Write a function *per2* that receives one number as an input argument. The function has a **persistent** variable that sums the values passed to it. Here are the first two times the function is called:

```
>> per2(4)
ans =

     4
>> per2(6)
ans =

    10
```

25. Assume a matrix variable *mat*, as in the following example:

```
mat =

     4     2     4     3     2

     1     3     1     0     5

     2     4     4     0     2
```

The following **for** loop

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

prints this result:

```
The sum is now 15
The sum is now 25
The sum is now 37
```

Write the function *sumprint*.

26. The following script *land* calls functions to:
    - prompt the user for a land area in acres
    - calculate and return the area in hectares and in square miles
    - print the results

    One acre is 0.4047 hectares. One square mile is 640 acres. Assume that the last function, that prints, exists—you do not have to do anything for that function. You are to write the entire function that calculates and returns the area in hectares and in square miles, and write just a function stub for the function that prompts the user and reads. Do not write the actual contents of this function; just write a stub!

land.m

```
inacres = askacres;
[sqmil, hectares] = convacres(inacres);
dispareas(inacres, sqmil, hectares) % Assume this exists
```

27. The braking distance of a car depends on its speed as the brakes are applied and on the car's braking efficiency. A formula for the braking distance is

$$b_d = \frac{s^2}{2Rg}$$

where b_d is the braking distance, s is the car's speed, R is the braking efficiency, and g is the acceleration due to gravity (9.81). A script has been written that calls a function to prompt the user for s and R, calls another function to calculate the braking distance, and calls a third function to print the braking distance in a sentence format with one decimal place. You are to write a function stub for the function that prompts for s and R, and the actual function definitions for the other two functions.

```
[s, R] = promptSandR;
brakDist = calcbd(s, R);
printbd(brakDist)
```

28. Write a menu-driven program to convert a time in seconds to other units (minutes, hours, and so on). The main script will loop to continue until the user chooses to exit. Each time in the loop, the script will generate a random time in seconds, call a function to present a menu of options, and print the converted time. The conversions must be made by individual functions (e.g., one to convert from seconds to minutes). All user-entries must be error-checked.
29. Write a menu-driven program to investigate the constant π. Model it after the program that explores the constant *e*. Pi (π) is the ratio of a circle's circumference to its diameter. Many mathematicians have found ways to approximate π. For example, Machin's formula is:

$$\frac{\pi}{4} = 4 \arctan\left(\frac{1}{5}\right) - \arctan\left(\frac{1}{239}\right)$$

Leibniz found that π can be approximated by:

$$\pi = \frac{4}{1} - \frac{4}{3} + \frac{4}{5} - \frac{4}{7} + \frac{4}{9} - \frac{4}{11} + \ldots$$

This is called a sum of a series. There are six terms shown in this series. The first term is 4, the second term is –4/3, the third term is 4/5, and so forth. For example, the menu-driven program might have the following options:

- Print the result from Machin's formula.
- Print the approximation using Leibniz' formula, allowing the user to specify how many terms to use.
- Print the approximation using Leibniz' formula, looping until a "good" approximation is found.
- Exit the program.

30. Write a program to calculate the position of a projectile at a given time *t*. For an initial velocity v_0 and angle of departure θ_0, the position is given by *x* and *y* coordinates as follows (note: the gravity constant *g* is 9.81 m/s^2):

$$x = v_0 \cos(\theta_0)t$$

$$y = v_0 \sin(\theta_0)t - \frac{1}{2}gt^2$$

The program should initialize the variables for the initial velocity, time, and angle of departure. It should then call a function to find the *x* and *y* coordinates, and then another function to print the results

31. Write the program for Problem 30 as a Live Script. Plot the results.