CHAPTER 10

Advanced Functions

KEY TERMS

variable number of arguments
nested functions
anonymous functions
function handle
function function
recursive functions
outer function
inner function
recursion
general (inductive) case
base case
infinite recursion

CONTENTS

Functions were introduced in Chapter 3 and then expanded on in Chapter 6. In this chapter, several advanced features of functions and types of functions will be described. All of the functions that we have seen so far have had a well-defined number of input and output arguments, but we will see that it is possible to have a ***variable number of arguments***. ***Nested functions*** are also introduced, which are functions contained within other functions. ***Anonymous functions*** are simple one-line functions that are called using their ***function handle***. Other uses of function handles will also be demonstrated, including ***function functions*** and built-in function functions in the MATLAB® software. Finally, ***recursive functions*** are functions that call themselves. A recursive function can return a value or may simply accomplish a task such as printing.

10.1 VARIABLE NUMBERS OF ARGUMENTS

The functions that we've written thus far have contained a fixed number of input arguments and a fixed number of output arguments. For example, in the following function that we have defined previously, there is one input argument and there are two output arguments:

MATLAB®. https://doi.org/10.1016/B978-0-12-815479-3.00010-6

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
```

However, this is not always the case. It is possible to have a ***variable number of arguments***, both input and output arguments. A built-in cell array **varargin** can be used to store a variable number of input arguments and a built-in cell array **varargout** can be used to store a variable number of output arguments. These are cell arrays because the arguments could be of different types, and cell arrays can store different kinds of values in the various elements. The function **nargin** returns the number of input arguments that were passed to the function, and the function **nargout** determines how many output arguments are expected to be returned from a function.

10.1.1 Variable Number of Input Arguments

For example, the following function *areafori* has a variable number of input arguments, either 1 or 2. The name of the function stands for "area, feet or inches". If only one argument is passed to the function, it represents the radius in feet. If two arguments are passed, the second can be a character 'i', indicating that the result should be in inches (for any other character, the default of feet is assumed). One foot = 12 inches. The function uses the built-in cell array **varargin**, which stores any number of input arguments. The function **nargin** returns the number of input arguments that were passed to the function. In this case, the radius is the first argument passed, so it is stored in the first element in **varargin**. If a second argument is passed (if **nargin** is 2), it is a character that specifies the units.

areafori.m

```
function area = areafori(varargin)
% areafori returns the area of a circle in feet
% The radius is passed, and potentially the unit of
% inches is also passed, in which case the result will be
% given in inches instead of feet
% Format: areafori(radius) or areafori(radius,'i')

n = nargin; % number of input arguments
radius = varargin{1}; % Given in feet by default
if n == 2
```

```
    unit = varargin{2};
    % if inches is specified, convert the radius
    if unit == 'i'
        radius = radius * 12;
    end
end
area = pi * radius .^2;
end
```

Note
Curly braces are used to refer to the elements in the cell array **varargin**.

Some examples of calling this function follow:

```
>> areafori(3)
ans =
    28.2743
>> areafori(1,'i')
ans =
    452.3893
```

In this case, it was assumed that the radius will always be passed to the function. The function header can, therefore, be modified to indicate that the radius will be passed, and then a variable number of remaining input arguments (either none or 1):

areafori2.m

```
function area = areafori2(radius, varargin)
% areafori2 returns the area of a circle in feet
% The radius is passed, and potentially the unit of
% inches is also passed, in which case the result will be
% given in inches instead of feet
% Format: areafori2(radius) or areafori2(radius,'i')

n = nargin; % number of input arguments

if n == 2
    unit = varargin{1};
    % if inches is specified, convert the radius
    if unit == 'i'
        radius = radius * 12;
    end
end
area = pi * radius .^2;
end
```

```
>> areafori2(3)
ans =
   28.2743
>> areafori2(1,'i')
ans =
  452.3893
```

Note
nargin returns the total number of input arguments, not just the number of arguments in the cell array **varargin**.

There are basically two formats for the function header with a variable number of input arguments. For a function with one output argument, the options are:

```
function outarg = fnname(varargin)
function outarg = fnname(input arguments, varargin)
```

Either some input arguments are built into the function header and **varargin** stores anything else that is passed, or all of the input arguments go into **varargin**.

PRACTICE 10.1

The sum of a geometric series is given by

$$1 + r + r^2 + r^3 + r^4 + \ldots + r^n$$

Write a function called *geomser* that will receive a value for *r* and calculate and return the sum of the geometric series. If a second argument is passed to the function, it is the value of *n*; otherwise, the function generates a random integer for *n* (in the range from 5 to 30). Note that loops are not necessary to accomplish this. The following examples of calls to this function illustrate what the result should be:

```
>> g = geomser(2,4)    % 1 + 2^1 + 2^2 + 2^3 + 2^4
g =
    31
>> geomser(1)         % 1 + 1^1 + 1^2 + 1^3 + ... ?
ans =
    12
```

Note that in the last example, a random integer was generated for *n* (which must have been 11). Use the following header for the function and fill in the rest:

```
function sgs = geomser(r, varargin)
```

10.1.2 Variable Number of Output Arguments

A variable number of output arguments can also be specified. For example, one input argument is passed to the following function *typesize*. The function will always return a character specifying whether the input argument was a scalar ('s'), vector ('v'), or matrix ('m'). This character is returned through the output argument *arrtype*.

Additionally, if the input argument was a vector, the function returns the length of the vector, and if the input argument was a matrix, the function returns the number of rows and the number of columns of the matrix. The output argument **varargout** is used, which is a cell array. So, for a vector the length is returned through **varargout,** and for a matrix both the number of rows and columns are returned through **varargout**.

typesize.m

```
function [arrtype, varargout] = typesize(inputval)
% typesize returns a character 's' for scalar, 'v'
% for vector, or 'm' for matrix input argument
% also returns length of a vector or dimensions of matrix
% Format: typesize(inputArgument)

[r, c ] = size(inputval);

if r==1 && c==1
     arrtype = 's';
elseif r==1 || c==1
     arrtype = 'v';
     varargout{1} = length(inputval);
else
     arrtype = 'm';
     varargout{1} = r;
     varargout{2} = c;
end
end
```

```
>> typesize(5)
ans =
  '     s'

>> [arrtype, len] = typesize(4:6)
arrtype =
  '     v'
len =
      3

>> [arrtype, r, c] = typesize([4:6;3:5])
arrtype =
  '     m'

r =
      2
c =
      3
```

In the examples shown here, the user must actually know the type of the argument in order to determine how many variables to have on the left-hand side of the assignment statement. An error will result if there are too many variables.

```
>> [arrtype, r, c] = typesize(4:6)
Output argument "varargout{2}" (and maybe others) not assigned during call
to "typesize".
```

The function **nargout** can be called to determine how many output arguments were used to call a function. For example, in the function *mysize* below, a matrix is passed to the function. The function behaves like the built-in function **size** in that it returns the number of rows and columns. However, if three variables are used to store the result of calling this function, it also returns the total number of elements:

mysize.m

```
function [row, col, varargout] = mysize(mat)
% mysize returns dimensions of input argument
% and possibly also total # of elements
% Format: mysize(inputArgument)

[row, col] = size(mat);

if nargout == 3
    varargout{1} = row*col;
end
end
```

```
>> [r, c] = mysize(zeros(3))
r =
     3
c =
     3
>> [r, c, elem] = mysize(zeros(3))
r =
     3
c =
     3
elem =
     9
```

Note

The function **nargout** does not return the number of output arguments in the function header, but the number of output arguments expected from the function (meaning the number of variables in the vector on the left side of the assignment statement when calling the function).

In the first call to the *mysize* function, the value of **nargout** was 2, so the function only returned the output arguments *row* and *col*. In the second call, as there were three variables on the left of the assignment statement, the value of **nargout** was 3; thus, the function also returned the total number of elements.

There are basically two formats for the function header with a variable number of output arguments:

```
function varargout = fnname(input args)
function [output args, varargout] = fnname(input args)
```

Either some output arguments are built into the function header, and **varargout** stores anything else that is returned or all go into **varargout**. The function is called as follows:

```
[variables] = fnname(input args);
```

QUICK QUESTION!

A temperature in degrees Celsius is passed to a function called *converttemp*. How could we write this function so that it converts this temperature to degrees Fahrenheit, and possibly also to degrees Kelvin, depending on the number of output arguments? The conversions are:

$$F = \frac{9}{5}C + 32$$

$$K = C + 273.15$$

Here are possible calls to the function:

```
>> df = converttemp(17)
df =
   62.6000
>> [df, dk] = converttemp(17)
df =
   62.6000
dk =
   290.1500
```

Answer: We could write the function two different ways: one with only **varargout** in the function header, and one that has an output argument for the degrees F and also **varargout** in the function header.

converttemp.m

```
function [degreesF, varargout] = converttemp(degreesC)
% converttemp converts temperature in degrees C
% to degrees F and maybe also K
% Format: converttemp(C temperature)

degreesF = 9/5*degreesC + 32;
n = nargout;
if n == 2
    varargout{1} = degreesC + 273.15;
end
end
```

converttempii.m

```
function varargout = converttempii(degreesC)
% converttempii converts temperature in degrees C
% to degrees F and maybe also K
% Format: converttempii(C temperature)

varargout{1} = 9/5*degreesC + 32;
n = nargout;
if n == 2
    varargout{2} = degreesC + 273.15;
end
end
```

10.2 NESTED FUNCTIONS

Just as loops can be nested, meaning one inside of another, functions can be nested. The terminology for ***nested functions*** is that an ***outer function*** can have within it ***inner functions***. When functions are nested, every function must have an **end** statement. The general format of a nested function is as follows:

```
outer function header

    body of outer function

    inner function header
        body of inner function
    end % inner function

    more body of outer function

end % outer function
```

The inner function can be in any part of the body of the outer function, so there may be parts of the body of the outer function before and after the inner function. There can be multiple inner functions.

The scope of any variable is the workspace of the outermost function in which it is defined and used. That means that a variable defined in the outer function could be used in an inner function (without passing it).

For example, the following function calculates and returns the volume of a cube. Three arguments are passed to it, for the length and width of the base of the cube, and also the height. The outer function calls a nested function that calculates and returns the area of the base of the cube.

Note

It is not necessary to pass the length and width (*len* and *wid*) to the inner function, as the scope of these variables includes the inner function.

nestedvolume.m

```
function outvol = nestedvolume(len, wid, ht)
% nestedvolume receives the lenght, width, and
% height of a cube and returns the volume; it calls
% a nested function that returns the area of the base
% Format: nestedvolume(length,width,height)

outvol = base * ht;

    function outbase = base
    % returns the area of the base
    outbase = len * wid;
    end % base function

end % nestedvolume function
```

An example of calling this function follows:

```
>> v = nestedvolume(3,5,7)
v =
   105
```

Output arguments are different from variables. The scope of an output argument is just the nested function; it cannot be used in the outer function. In this example, *outbase* can only be used in the *base* function; its value, for example, could not be printed from *nestedvolume*.

A variable defined in the inner function ***could*** be used in the outer function, but if it is not used in the outer function the scope is just the inner function.

Examples of nested functions will be used in the section on Graphical User Interfaces.

10.3 ANONYMOUS FUNCTIONS AND FUNCTION HANDLES

An anonymous function is a very simple, one-line function. The advantage of an anonymous function is that it does not have to be stored in a separate file. This can greatly simplify programs, as often calculations are very simple and the use of anonymous functions reduces the number of code files necessary for a program. Anonymous functions can be created in the Command Window or in any script or user-defined function. The syntax for an anonymous function follows:

```
fnhandlevar = @ (arguments) functionbody;
```

where *fnhandlevar* stores the ***function handle***; it is essentially a way of referring to the function. The handle is returned by the @ operator and then this handle is assigned to the variable *fnhandlevar* on the left. The arguments, in parentheses, correspond to the argument(s) that are passed to the function, just like any other kind of function. The functionbody is the body of the function, which is any valid MATLAB expression. For example, here is an anonymous function that calculates and returns the area of a circle:

```
>> cirarea = @ (radius) pi * radius .^ 2;
```

The function handle variable name is *cirarea*. There is one input argument, *radius*. The body of the function is the expression `pi * radius .^ 2`. The `.^` array operator is used so that a vector of radii can be passed to the function.

The function is then called using the handle and passing argument(s) to it; in this case, the radius or vector of radii. The function call using the function handle looks just like a function call using a function name:

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

```
>> areas = cirarea(1:4)
areas =
    3.1416   12.5664   28.2743   50.2655
```

The type of *cirarea* can be found using the **class** function:

```
>> class(cirarea)
ans =
function_handle
```

Unlike functions stored in code files, if no argument is passed to an anonymous function, the parentheses must still be in the function definition and in the function call. For example, the following is an anonymous function that prints a random real number with two decimal places, as well as a call to this function:

```
>> prtran = @ () fprintf('%.2f\n',rand);
>> prtran()
0.95
```

Typing just the name of the function handle will display its contents, which is the function definition.

```
>> prtran
prtran =
    @ () fprintf('%.2f\n',rand)
```

This is why parentheses must be used to call the function, even though no arguments are passed.

An anonymous function can be saved to a MAT-file and then it can be loaded when needed.

```
>> cirarea = @ (radius) pi * radius .^2;
>> save anonfns cirarea
>> clear
>> load anonfns
>> who
Your variables are:
cirarea
>> cirarea
cirarea =
    @ (radius) pi * radius .^2
```

Other anonymous functions could be appended to this MAT-file. Even though an advantage of anonymous functions is that they do not have to be saved in individual code files, it is frequently useful to save groups of related anonymous functions in a single MAT-file. Anonymous functions that are used frequently can be saved in a MAT-file and then loaded from this MAT-file in every MATLAB Command Window.

PRACTICE 10.2

Create your own anonymous functions to perform some temperature conversions. Store these anonymous function handle variables in a MAT-file called "tempconverters.mat".

10.4 USES OF FUNCTION HANDLES

Function handles can also be created for functions other than anonymous functions, both built-in and user-defined functions. For example, the following would create a function handle for the built-in **factorial** function:

```
>> facth = @factorial;
```

The @ operator gets the handle of the function, which is then stored in a variable *facth*.

The handle could then be used to call the function, just like the handle for the anonymous functions, as in:

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

Using the function handle to call the function instead of using the name of the function does not in itself demonstrate why this is useful, so an obvious question would be why function handles are necessary for functions other than anonymous functions.

10.4.1 Function Functions

One reason for using function handles is to be able to pass functions to other functions—these are called ***function functions***. For example, let's say we have a function that creates an x vector. The y vector is created by evaluating a function at each of the x points, and then these points are plotted.

fnfnexamp.m

```
function fnfnexamp(funh)
% fnfnexamp receives the handle of a function
% and plots that function of x (which is 1:.25:6)
% Format: fnfnexamp(function handle)

x = 1:.25:6;
y = funh(x);
plot(x,y,'ko')
xlabel('x')
ylabel('fn(x)')
title(func2str(funh))
end
```

What we want to do is pass a function to be the value of the input argument *funh*, such as **sin**, **cos**, or **tan**. Simply passing the name of the function does not work:

```
>> fnfnexamp(sin)
Error using sin
Not enough input arguments.
```

Instead, we have to pass the handle of the function:

```
>> fnfnexamp(@sin)
```

This creates the *y* vector as **sin(x)** and then brings up the plot as seen in Figure 10.1. The function **func2str** converts a function handle to a character vector; this is used for the title.

Passing the handle to the **cos** function instead would graph cosine instead of sine:

```
>> fnfnexamp(@cos)
```

We could also pass the handle of any user-defined or anonymous function to the *fnfnexamp* function. Note that if a variable stores a function handle, just the name of the variable would be passed (not the @ operator). For example, for our anonymous function defined previously,

```
>> fnfnexamp(cirarea)
```

The function **func2str** will return the definition of an anonymous function as a character vector that could also be used as a title. For example:

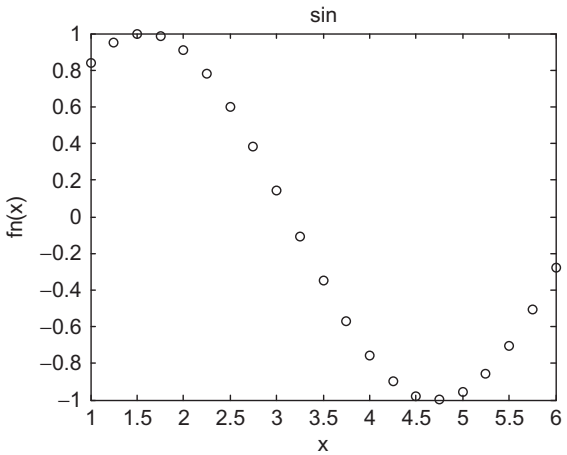

FIGURE 10.1
Plot of **sin** created by passing handle of function to plot.

```
>> cirarea = @ (radius) pi * radius .^2;
>> fnname = func2str(cirarea)
fnname =
@(radius)pi*radius.^2
```

There is also a built-in function **str2func** that will convert a string scalar or character vector to a function handle. A string containing the name of a function could be passed as an input argument, and then converted to a function handle.

fnstrfn.m

```
function fnstrfn(funstr)
% fnstrfn receives the name of a function as a string
% it converts this to a function handle and
% then plots the function of x (which is 1:.25:6)
% Format: fnstrfn(function name as string)
x = 1:.25:6;
funh = str2func(funstr);
y = funh(x);
plot(x,y,'ko')
xlabel('x')
ylabel('fn(x)')
title(funstr)
end
```

This would be called by passing a string to the function and would create the same plot as in Figure 10.1:

```
>> fnstrfn("sin")
```

PRACTICE 10.3

Write a function that will receive as input arguments an *x* vector and a function handle and will create a vector *y* that is the function of *x* (whichever function handle is passed) and will also plot the data from the *x* and *y* vectors with the function name in the title.

MATLAB has some built-in function functions. One built-in function function is **fplot**, which plots a function between limits that are specified. The form of the call to **fplot** is:

```
fplot(fnhandle, [xmin, xmax])
```

For example, to pass the **sin** function to **fplot**, one would pass its handle (see Figure 10.2 for the result).

```
>> fplot(@sin, [-pi, pi])
```

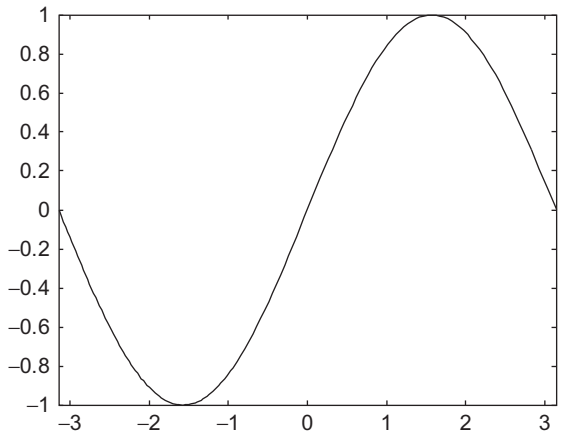

FIGURE 10.2
Plot of **sin** created using **fplot**.

The **fplot** function is a nice shortcut—it is not necessary to create x and y vectors, and it plots a continuous curve rather than discrete points.

QUICK QUESTION!

Could you pass an anonymous function to the **fplot** function?

Answer: Yes, as in:

```
>> cirarea = @ (radius) pi * radius .^2;
>> fplot(cirarea, [1, 5])
>> title(func2str(cirarea))
```

Note that, in this case, the **@** operator is not used in the call to **fplot**, as *cirarea* already stores the function handle.

The function function **feval** will evaluate a function handle and execute the function for the specified argument. For example, the following is equivalent to **sin(3.2)**:

```
>> feval(@sin, 3.2)
ans =
   -0.0584
```

Another built-in function function is **fzero**, which finds a zero of a function near a specified value. For example:

```
>> fzero(@cos,4)
ans =
    4.7124
```

10.4.2 Timing Functions

The function **timeit** (introduced in R2013b) can be used to time functions and is more robust than using **tic** and **toc**. The **timeit** function takes one input argument, which is a function handle; this can be the handle of any type of function. The time is returned in seconds.

```
>> fh = @() prod(1:10000000);
>> timeit(fh)
ans =
    0.0308
```

A warning message is thrown if the function is too fast.

```
>> fh = @() prod(1:100);
>> timeit(fh)
Warning: The measured time for F may be inaccurate because it is running too
fast. Try measuring something that takes longer.
> In timeit (line 158)
ans =
    1.3993e-06
```

10.5 RECURSIVE FUNCTIONS

Recursion is when something is defined in terms of itself. In programming, a ***recursive function*** is a function that calls itself. Recursion is used very commonly in programming, although many simple examples (including some shown in this section) are actually not very efficient and can be replaced by iterative methods (loops or vectorized code in MATLAB). Nontrivial examples go beyond the scope of this book, so the concept of recursion is simply introduced here.

The first example will be of a factorial. Normally, the factorial of an integer n is defined iteratively:

```
n! = 1 * 2 * 3 * ... * n
```

For example, `4! = 1 * 2 * 3 * 4`, or 24.

Another, recursive, definition is:

```
n! = n * (n - 1)!        general case
1! = 1                   base case
```

This definition is recursive because a factorial is defined in terms of another factorial. There are two parts to any recursive definition: the ***general (or inductive) case***, and the ***base case***. We say that, in general, the factorial of n is defined as n multiplied by the factorial of $(n-1)$, but the base case is that the factorial of 1 is just 1. The base case stops the recursion.

For example:

```
3! = 3 * 2!
       2! = 2 * 1!
              1! = 1
         = 2
   = 6
```

The way this works is that 3! is defined in terms of another factorial, as 3 * 2!. This expression cannot yet be evaluated because, first, we have to find out the value of 2!. So, in trying to evaluate the expression 3 * 2!, we are interrupted by the recursive definition. According to the definition, 2! is 2 * 1!. Again, the expression 2 * 1! cannot yet be evaluated because, first, we have to find the value of 1!. According to the definition, 1! is 1. As we now know what 1! is, we can continue with the expression that was just being evaluated; now we know that 2 * 1! is 2 * 1, or 2. Thus, we can now finish the previous expression that was being evaluated; now we know that 3 * 2! is 3 * 2, or 6.

This is the way that recursion always works. With recursion, the expressions are put on hold with the interruption of the general case of the recursive definition. This keeps happening until the base case of the recursive definition applies. This finally stops the recursion, and then the expressions that were put on hold are evaluated in the reverse order. In this case, first the evaluation of 2 * 1! was completed, and then 3 * 2!.

There must always be a base case to end the recursion, and the base case must be reached at some point. Otherwise, ***infinite recursion*** would occur (theoretically, although MATLAB will stop the recursion eventually).

We have already seen the built-in function **factorial** in MATLAB to calculate factorials, and we have seen how to implement the iterative definition using a running product. Now we will instead write a recursive function called *fact*. The function will receive an integer *n*, which we will for simplicity assume is a positive integer, and will calculate *n!* using the recursive definition given previously.

fact.m

```
function facn = fact(n)
% fact recursively finds n!
% Format: fact(n)
if n == 1
    facn = 1;
else
    facn = n * fact(n-1);
end
end
```

The function calculates one value, using an **if-else** statement to choose between the base and general cases. If the value passed to the function is 1, the function

returns 1 as 1! is equal to 1. Otherwise, the general case applies. According to the definition, the factorial of n, which is what this function is calculating, is defined as n multiplied by the factorial of $(n-1)$. So, the function assigns `n*fact(n-1)` to the output argument.

How does this work? Exactly the way the example was sketched previously for 3!. Let's trace what would happen if the integer 3 is passed to the function:

```
fact(3) tries to assign 3 * fact(2)
                    fact(2) tries to assign 2 * fact(1)
                                        fact(1) assigns 1
                    fact(2) assigns 2
fact(3) assigns 6
```

When the function is first called, 3 is not equal to 1, so the statement

```
facn = n * fact(n-1);
```

is executed. This will attempt to assign the value of 3 * fact(2) to *facn*, but this expression cannot be evaluated yet and therefore a value cannot be assigned yet because first the value of fact(2) must be found.

Thus, the assignment statement has been interrupted by a recursive call to the *fact* function. The call to the function fact(2) results in an attempt to assign 2 * fact(1), but, again, this expression cannot yet be evaluated. Next, the call to the function fact(1) results in a complete execution of an assignment statement as it assigns just 1. Once the base case has been reached, the assignment statements that were interrupted can be evaluated, in the reverse order.

Calling this function yields the same result as the built-in **factorial** function, as follows:

```
>> fact(5)
ans =
   120
>> factorial(5)
ans =
   120
```

The recursive factorial function is a very common example of a recursive function. It is somewhat of a lame example, however, as recursion is not necessary to find a factorial. A **for** loop can be used just as well in programming (or, of course, the built-in function in MATLAB).

Another, better, example is of a recursive function that does not return anything, but simply prints. The following function *prtwords* receives a sentence and prints the words in the sentence in reverse order. The algorithm for the *prtwords* function follows:

- Receive a sentence as an input argument.
- Use **strtok** to break the sentence into the first word and the rest of the sentence.

- If the rest of the sentence is not empty (in other words, if there is more to it), recursively call the *prtwords* function and pass to it the rest of the sentence.
- Print the word.

The function definition follows:

prtwords.m

```
function prtwords(sent)
% prtwords recursively prints the words in a
%  sentence in reverse order
% Format: prtwords(sentence)

[word, rest] = strtok(sent);
if ~isempty(rest)
   prtwords(rest);
end
disp(word)
end
```

Here is an example of calling the function, passing the sentence "what does this do":

```
>> prtwords('what does this do')
do
this
does
what
```

An outline of what happens when the function is called follows:

```
The function receives 'what does this do'
It breaks it into word = 'what', rest = 'does this do'
Since "rest" is not empty, calls prtwords, passing "rest"
    The function receives 'does this do'
    It breaks it into word = 'does', rest = 'this do'
    Since "rest" is not empty, calls prtwords, passing "rest"
        The function receives 'this do'
        It breaks it into word = 'this', rest = 'do'
        Since "rest" is not empty, calls prtwords, passing "rest"
            The function receives 'do'
            It breaks it into word = 'do', rest = ''
            "rest" is empty so no recursive call
            Print 'do'
        Print 'this'
    Print 'does'
Print 'what'
```

In this example, the base case is when the rest of the sentence is empty; in other words, the end of the original sentence has been reached. Every time the function is called, the execution of the function is interrupted by a recursive call to the function until the base case is reached. When the base case is reached, the entire function can be executed, including printing the word (in the base case, the word 'do').

Once that execution of the function is completed, the program returns to the previous version of the function in which the word was 'this,' and finishes the execution by printing the word 'this.' This continues; the versions of the function are finished in the reverse order, so the program ends up printing the words from the sentence in the reverse order.

PRACTICE 10.4

For the following function,

recurfn.m

```
function outvar = recurfn(num)
% Format: recurfn(number)

if num < 0
   outvar = 2;
else
   outvar = 4 + recurfn(num-1);
end
end
```

what would be returned by the call to the function `recurfn(3.5)`? Think about it, and then type in the function and test it.

Explore Other Interesting Features

Other function functions and Ordinary Differential Equation (ODE) solvers can be found using **help funfun**.

The function function **bsxfun**. Look at the example in the documentation page of subtracting the column mean from every element in each column of a matrix.

The ODE solvers include **ode45** (which is used most often), **ode23**, and several others. Error tolerances can be set with the **odeset** function.

Investigate the use of the functions **narginchk** and **nargoutchk**.

The function **nargin** can be used not just when using **varargin**, but also for error-checking for the correct number of input arguments into a function. Explore examples of this. ■

SUMMARY

COMMON PITFALLS

- Thinking that **nargin** is the number of elements in **varargin** (it may be, but not necessarily; **nargin** is the total number of input arguments).
- Trying to pass just the name of a function to a function function; instead, the function handle must be passed.
- Forgetting the base case for a recursive function.

PROGRAMMING STYLE GUIDELINES

- If some inputs and/or outputs will always be passed to/from a function, use standard input arguments/output arguments for them. Use **varargin** and **varargout** only when it is not known ahead of time whether other input/output arguments will be needed.
- Use anonymous functions whenever the function body consists of just a simple expression.
- Store related anonymous functions together in one MAT-file
- Use iteration instead of recursion when possible.

MATLAB Reserved Words
end (for functions)

MATLAB Functions and Commands	
varargin	str2func
varargout	fplot
nargin	feval
nargout	fzero
func2str	timeit

MATLAB Operators
handle of functions @

Exercises

1. Write a function that will print a random integer. If no arguments are passed to the function, it will print an integer in the inclusive range from 1 to 100. If one argument is passed, it is the max and the integer will be in the inclusive range from 1 to max. If two arguments are passed, they represent the min and max and it will print an integer in the inclusive range from min to max.
2. Write a function *numbers* that will create a matrix in which every element stores the same number *num*. Either two or three arguments will be passed to the function. The first argument will always be the number *num*. If there are two arguments, the second will be the size of the resulting square (n × n) matrix. If there are three arguments, the second and third will be the number of rows and columns of the resulting matrix.
3. The overall electrical resistance of *n* resistors in parallel is given as:

$$R_T = \left(\frac{1}{R_1} + \frac{1}{R_2} + \frac{1}{R_3} + \ldots + \frac{1}{R_n}\right)^{-1}$$

   Write a function *Req* that will receive a variable number of resistance values and will return the equivalent electrical resistance of the resistor network.
4. Write a function that will receive the radius *r* of a sphere. It will calculate and return the volume of the sphere ($4/3\ \pi\ r^3$). If the function call expects two output arguments, the function will also return the surface area of the sphere ($4\ \pi\ r^2$).
5. Write a function *writel* that will write the length of a side of a square. If only one argument is passed to the function, it is the length, and the output will go to the Command Window. If two arguments are passed, however, the first argument will be the length and the second argument will be the file identifier for an output file to which the length should be printed.
6. Most lap swimming pools have lanes that are either 25 yards long or 25 meters long; there's not much of a difference. A function "convyards" is to be written to help swimmers calculate how far they swam. The function receives as input the number of yards. It calculates and returns the equivalent number of meters and, if (and only if) two output arguments are expected, it also returns the equivalent number of miles. The relevant conversion factors are:

```
1 meter = 1.0936133 yards
1 mile = 1760 yards
```

7. Write a function that will receive as an input argument a temperature in degrees Fahrenheit and will return the temperature in degrees Celsius and, if two outputs are expected, also in Kelvin. The conversion factors are: C = (F − 32) * 5/9 and K = C + 273.15.
8. Write a function *unwind* that will receive a matrix as an input argument. It will return a row vector created columnwise from the elements in the matrix. If the number of expected output arguments is two, it will also return this as a column vector.

9. Write a function *areaperim* that will calculate both the area and perimeter of a polygon. The radius r will be passed as an argument to the function. If a second argument is passed to the function, it represents the number of sides n. If, however, only one argument is passed, the function generates a random value for n (an integer in the range from 3 to 8). For a polygon with *n* sides inscribed in a circle with a radius of *r*, the area *a* and perimeter *p* of the polygon can be found by

$$a = \frac{1}{2} n r^2 \sin\left(\frac{2\pi}{n}\right), \quad p = 2\pi r \sin\left(\frac{\pi}{n}\right)$$

10. Information on some hurricanes is stored in a vector of structures; the name of the vector variable is *hurricanes*. For example, one of the structures might be initialized as follows:

```
struct('Name','Bettylou', 'Avespeed',18,...
       'Size', struct('Width',333,'Eyewidth',22));
```

Write a function *printHurr* that will receive a vector of structures in this format as an input argument. It will print, for every hurricane, its *Name* and *Width* in a sentence format to the screen. If a second argument is passed to the function, it is a file identifier for an output file (which means that the file has already been opened), and the function will print in the same format to this file (and does not close it).

11. The built-in function **date** returns a character vector containing the day, month, and year. Write a function (using the **date** function) that will always return the current day. If the function call expects two output arguments, it will also return the month. If the function call expects three output arguments, it will also return the year.
12. List some built-in functions to which you pass a variable number of input arguments (Note: this is not asking for **varargin**, which is a built-in cell array, or **nargin**.)
13. List some built-in functions that have a variable number of output arguments (or, at least one!).
14. Write a function that will receive a variable number of input arguments: the length and width of a rectangle, and possibly also the height of a box that has this rectangle as its base. The function should return the rectangle area if just the length and width are passed, or also the volume if the height is also passed.
15. Write a function to calculate the volume of a cone. The volume V is V = AH, where A is the area of the circular base ($A = \pi r^2$ where r is the radius) and H is the height. Use a nested function to calculate A.
16. The two real roots of a quadratic equation $ax^2 + bx + c = 0$ (where a is nonzero) are given by

$$\frac{-b \pm \sqrt{D}}{2*a}$$

where the discriminant D = b^2 − 4*a*c. Write a function to calculate and return the roots of a quadratic equation. Pass the values of a, b, and c to the function. Use a nested function to calculate the discriminant.

17. The velocity of sound in air is 49.02 $\sqrt{T}$ feet per second where T is the air temperature in degrees Rankine. Write an anonymous function that will calculate this. One argument, the air temperature in degrees R, will be passed to the function and it will return the velocity of sound.
18. Create a set of anonymous functions to do length conversions and store them in a file named *lenconv.mat*. Call each a descriptive name, such as *cmtoinch,* to convert from centimeters to inches.
19. An approximation for a factorial can be found using Stirling's formula:

$$n! \approx \sqrt{2\pi n}\left(\frac{n}{e}\right)^n$$

Write an anonymous function to implement this.

20. Why would you want to use an anonymous function?
21. Write an anonymous function to implement the following quadratic: `3x^2-2x+5`. Then, use **fplot** to plot the function in the range from −6 to 6.
22. Write a function that will receive data in the form of x and y vectors, and a handle to a plot function, and will produce the plot. For example, a call to the function would look like `wsfn(x,y,@bar)`.
23. Write a function *plot2fnhand* that will receive two function handles as input arguments and will display in two Figure Windows plots of these functions, with the function names in the titles. The function will create an x vector that ranges from 1 to n (where n is a random integer in the inclusive range from 4 to 10). For example, if the function is called as follows

```
>> plot2fnhand(@sqrt, @exp)
```

and the random integer is 5, the first Figure Window would display the **sqrt** function of x = 1:5, and the second Figure Window would display **exp(x)** for x = 1:5.

24. Use **feval** as an alternative way to accomplish the following function calls:

`abs(-4)`
`size(zeros(4))` Use **feval** twice for this one!

25. There is a built-in function function called **cellfun** that evaluates a function for every element of a cell array. Create a cell array, then call the **cellfun** function, passing the handle of the **length** function and the cell array to determine the length of every element in the cell array.
26. A recursive definition of a^n where a is an integer and n is a non-negative integer follows:

$$a^n = 1 \quad \text{if } n == 0$$
$$= a * a^{n-1} \quad \text{if } n > 0$$

Write a recursive function called *mypower*, which receives *a* and *n* and returns the value of a^n by implementing the previous definition. Note: The program should NOT use the ^ operator anywhere; this is to be done recursively instead! Test the function.

27. What does this function do:

```
function outvar = mystery(x,y)
if y == 1
  outvar = x;
else
  outvar = x + mystery(x,y-1);
end
```

Give one word to describe what this function does with its two arguments.

The Fibonacci numbers is a sequence of numbers 0, 1, 1, 2, 3, 5, 8, 13, 21, 34... The sequence starts with 0 and 1. All other Fibonacci numbers are obtained by adding the previous two Fibonacci numbers. The higher up in the sequence that you go, the closer the fraction of one Fibonacci number divided by the previous is to the golden ratio. The Fibonacci numbers can be seen in an astonishing number of examples in nature, for example, the arrangement of petals on a sunflower.

28. The Fibonacci numbers is a sequence of numbers F_i:

```
0  1  1  2  3  5  8  13  21  34 ...
```

where F_0 is 0, F_1 is 1, F_2 is 1, F_3 is 2, and so on. A recursive definition is:

$F_0 = 0$

$F_1 = 1$

$F_n = F_{n-2} + F_{n-1}$ if $n > 1$

Write a recursive function to implement this definition. The function will receive one integer argument n, and it will return one integer value that is the nth Fibonacci number. Note that in this definition, there is one general case but two base cases. Then, test the function by printing the first 20 Fibonacci numbers.

29. Use **fgets** to read character vectors from a file and recursively print them backwards.