Chapter 6

# Basic User-Defined Functions

In Chapter 4, we learned the importance of good program design. The basic technique that we employed was **top-down design**. In top-down design, you start with a statement of the problem to be solved and the required inputs and outputs. Next, you describe the algorithm to be implemented by the program in broad outline, and you apply *decomposition* to break down the algorithm into logical subdivisions called sub-tasks. Then you break down each sub-task until you have many small pieces, each of which does a simple, clearly understandable job. Finally, you turn the individual pieces into MATLAB code.

Although we have followed this design process in our examples, the results have been somewhat restricted because we have had to combine the final MATLAB code generated for each sub-task into a single large program. There has been no way to code, verify, and test each sub-task independently before combining them into the final program.

Fortunately, MATLAB has a special mechanism designed to make sub-tasks easy to develop and debug independently before building the final program. It is possible to code each sub-task as a separate **function**, and each function can be tested and debugged independently of all of the other sub-tasks in the program.

Well-designed functions enormously reduce the effort required on a large programming project. Their benefits include:

1. **Independent testing of sub-tasks.** Each sub-task can be written as an independent unit. The sub-task can be tested separately to ensure that it performs properly by itself before combining it into the larger program. This step is known as **unit testing**. It eliminates a major source of problems before the final program is built.
2. **Reusable code**. In many cases, the same basic sub-task is needed in many parts of a program. For example, it may be necessary to sort a list of values into ascending order many different times within a program, or even in other programs. It is possible to design, code, test, and debug a *single* function to do the sorting, and then to reuse that function whenever sorting is

required. This reusable code has two major advantages: it reduces the total programming effort required, and it simplifies debugging, since the sorting function only needs to be debugged once.

3. **Isolation from unintended side effects.** Functions receive input data from the program that invokes them through a list of variables called an **input argument list**, and return results to the program through an **output argument list**. Each function has its own workspace with its own variables, independent of all other functions and of the calling program. *The only variables in the calling program that can be seen by the function are those in the input argument list, and the only variables in the function that can be seen by the calling program are those in the output argument list.* This is very important, since accidental programming mistakes within a function can only affect the variables within the function in which the mistake occurred.

Once a large program is written and released, it has to be *maintained*. Program maintenance involves fixing bugs and modifying the program to handle new and unforeseen circumstances. The engineer who modifies a program during maintenance is often not the person who originally wrote it. In poorly written programs, it is common for the engineer modifying the program to make a change in one region of the code, and to have that change cause unintended side effects in a totally different part of the program. This happens because variable names are reused in different portions of the program. When the engineer changes the values left behind in some of the variables, those values are accidentally picked up and used in other portions of the code.

The use of well-designed functions minimizes this problem by **data hiding**. The variables in the main program are not visible to the function (except for those in the input argument list), and the variables in the main program cannot be accidentally modified by anything occurring in the function. Therefore, mistakes or changes in the function's variables cannot accidentally cause unintended side effects in other parts of the program.

### Good Programming Practice

Break large program tasks into functions whenever practical to achieve the important benefits of independent component testing, reusability, and isolation from undesired side effects.

## 6.1 Introduction to MATLAB Functions

All of the M-files that we have seen so far have been **script files**. Script files are just collections of MATLAB statements that are stored in a file. When a script file is executed, the result is the same as it would be if all of the commands had been typed directly into the Command Window. Script files share the Command Window's workspace, so any variables that were defined before the script file starts are visible to the script file, and any variables created by the script file remain in the workspace after the script file finishes executing. A script file has no input arguments and returns no results, but script files can communicate with other script files through the data left behind in the workspace.

In contrast, a **MATLAB function** is a special type of M-file that runs in its own independent workspace. It receives input data through an **input argument list**, and returns results to the caller through an **output argument list**. The general form of a MATLAB function is

```
function [outarg1, outarg2, ...] = fname(inarg1, inarg2, ...)
% H1 comment line
% Other comment lines
...
(Executable code)
...
(return)
(end)
```

The `function` statement marks the beginning of the function. It specifies the name of the function and the input and output argument lists. The input argument list appears in parentheses after the function name, and the output argument list appears in brackets to the left of the equal sign. (If there is only one output argument, the brackets can be dropped.)

Each ordinary MATLAB function should be placed in a file with the same name (including capitalization) as the function, and the file extension "`.m`". For example, if a function is named `My_fun`, then that function should be placed in a file named `My_fun.m`.

The input argument list is a list of names representing values that will be passed from the caller to the function. These names are called **dummy arguments**. They are just placeholders for actual values that are passed from the caller when the function is invoked. Similarly, the output argument list contains a list of dummy arguments that are placeholders for the values returned to the caller when the function finishes executing.

A function is invoked by naming it in an expression together with a list of **actual arguments**. A function can be invoked by typing its name directly in the Command Window, or by including it in a script file or another function. The name in the calling program must *exactly match* the function name (including capitalization).[1] When the function is invoked, the value of the first actual argument is used in place of the first dummy argument, and so forth for each other actual argument/dummy argument pair.

[1]For example, suppose that a function has been declared with the name `My_Fun`, and placed in file `My_Fun.m`. Then this function should be called with the name `My_Fun`, not `my_fun` or `MY_FUN`. If the capitalization fails to match, MATLAB will look for the most similar function name and ask if you want to run that function.

Execution begins at the top of the function and ends when a `return` statement, an `end` statement, or the end of the function is reached. Because execution stops at the end of a function, the `return` statement is not actually required in most functions and is rarely used. Each item in the output argument list must appear on the left side of at least one assignment statement in the function. When the function returns, the values stored in the output argument list are returned to the caller and may be used in further calculations.

The use of an `end` statement to terminate a function is a new feature as of MATLAB 7.0. It is optional unless a file includes nested functions, which we describe in Chapter 7. We will not use the `end` statement to terminate a function unless it is actually needed, so you will rarely see it used in this book.

The initial comment lines in a function serve a special purpose. The first comment line after the function statement is called the **H1 comment line**. It should always contain a one-line summary of the purpose of the function. The special significance of this line is that it is searched and displayed by the `lookfor` command. The remaining comment lines from the H1 line until the first blank line or the first executable statement are displayed by the `help` command. They should contain a brief summary of how to use the function.

A simple example of a user-defined function is shown next. Function `dist2` calculates the distance between points $(x_1, y_1)$ and $(x_2, y_2)$ in a Cartesian coordinate system.

```
function distance = dist2 (x1, y1, x2, y2)
%DIST2 Calculate the distance between two points
% Function DIST2 calculates the distance between
% two points (x1,y1) and (x2,y2) in a Cartesian
% coordinate system.
%
% Calling sequence:
%    distance = dist2(x1, y1, x2, y2)

% Define variables:
%   x1       -- x-position of point 1
%   y1       -- y-position of point 1
%   x2       -- x-position of point 2
%   y2       -- y-position of point 2
%   distance -- Distance between points

%  Record of revisions:
%      Date       Programmer          Description of change
%      ====       ==========          =====================
%    02/01/18    S. J. Chapman        Original code

% Calculate distance.
distance = sqrt((x2-x1).^2 + (y2-y1).^2);
```

This function has four input arguments and one output argument. A simple script file using this function is shown next.

```
%  Script file: test_dist2.m
%
%  Purpose:
%    This program tests function dist2.
%
%  Record of revisions:
%      Date       Programmer          Description of change
%      ====       ==========          =====================
%    02/01/18    S. J. Chapman        Original code
%
% Define variables:
%   ax     -- x-position of point a
%   ay     -- y-position of point a
%   bx     -- x-position of point b
%   by     -- y-position of point b
%   result -- Distance between the points

% Get input data.
disp('Calculate the distance between two points:');
ax = input('Enter x value of point a:   ');
ay = input('Enter y value of point a:   ');
bx = input('Enter x value of point b:   ');
by = input('Enter y value of point b:   ');

% Evaluate function
result = dist2 (ax, ay, bx, by);

% Write out result.
fprintf('The distance between points a and b is %f\n',result);
```

When this script file is executed, the results are:

```
» test_dist2
Calculate the distance between two points:
Enter x value of point a:   1
Enter y value of point a:   1
Enter x value of point b:   4
Enter y value of point b:   5
The distance between points a and b is 5.000000
```

These results are correct, as we can verify from simple hand calculations.

Function `dist2` also supports the MATLAB help subsystem. If we type "`help dist2`", the results are:

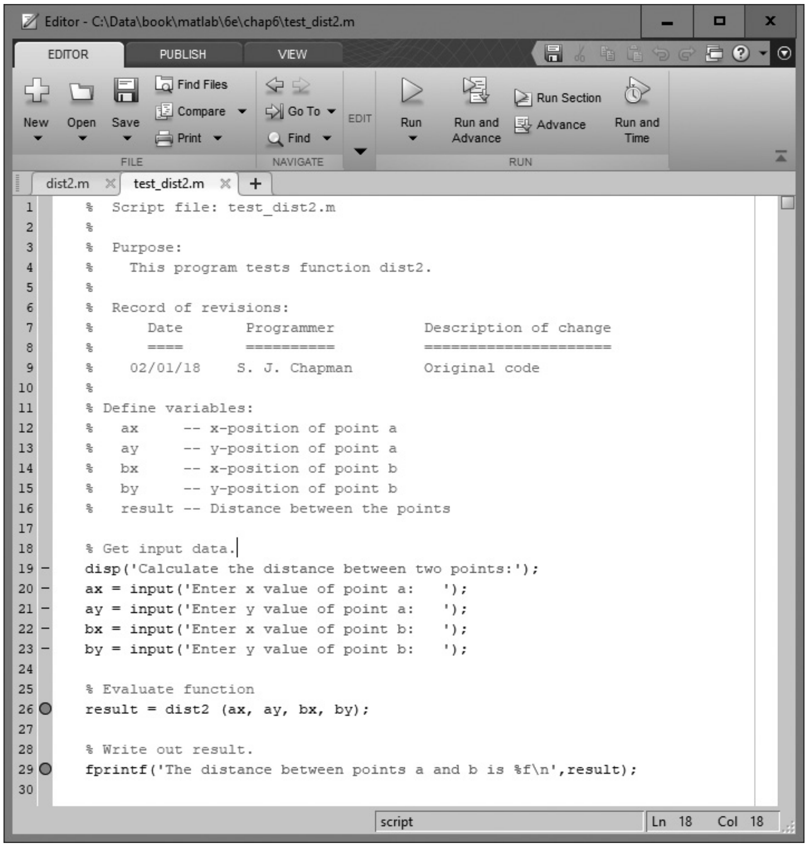
```
» help dist2
DIST2 Calculate the distance between two points
    Function DIST2 calculates the distance between
```

```
        two points (x1,y1) and (x2,y2) in a Cartesian
        coordinate system.

        Calling sequence:
         res = dist2(x1, y1, x2, y2)
```

Similarly, "`lookfor distance`" produces the result

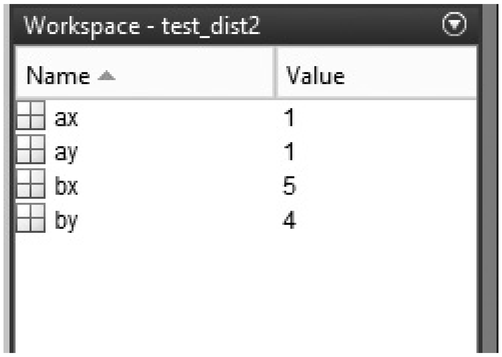
```
» lookfor distance
dist2        - Calculate the distance between two points
turningdist  - Find the turning distance of two polyshapes
```

To observe the behavior of the MATLAB workspace before, during, and after the function is executed, we will load function `dist2` and the script file `test_dist2` into the MATLAB debugger, and set breakpoints before, during, and after the function call (see Figure 6.1). When the program stops at the breakpoint *before* the function call, the workspace is as shown in Figure 6.2a. Note that variables `ax`, `ay`, `bx`,



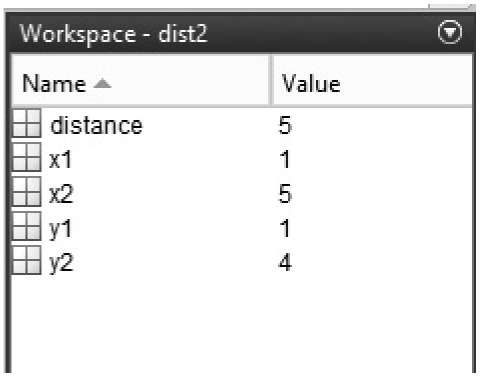
```
%  Script file: test_dist2.m
%
%  Purpose:
%    This program tests function dist2.
%
%  Record of revisions:
%      Date       Programmer          Description of change
%      ====       ==========          =====================
%    02/01/18    S. J. Chapman        Original code
%
% Define variables:
%   ax     -- x-position of point a
%   ay     -- y-position of point a
%   bx     -- x-position of point b
%   by     -- y-position of point b
%   result -- Distance between the points

% Get input data.
disp('Calculate the distance between two points:');
ax = input('Enter x value of point a:   ');
ay = input('Enter y value of point a:   ');
bx = input('Enter x value of point b:   ');
by = input('Enter y value of point b:   ');

% Evaluate function
result = dist2 (ax, ay, bx, by);

% Write out result.
fprintf('The distance between points a and b is %f\n',result);
```

**Figure 6.1** M-file `test_dist2` and function `dist2` are loaded into the debugger, with breakpoints set before, during, and after the function call.


Workspace - test_dist2

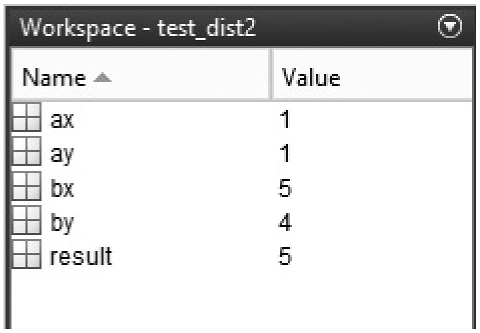
| Name | Value |
|---|---|
| ax | 1 |
| ay | 1 |
| bx | 5 |
| by | 4 |

(a)


Workspace - dist2

| Name | Value |
|---|---|
| distance | 5 |
| x1 | 1 |
| x2 | 5 |
| y1 | 1 |
| y2 | 4 |

(b)


Workspace - test_dist2

| Name | Value |
|---|---|
| ax | 1 |
| ay | 1 |
| bx | 5 |
| by | 4 |
| result | 5 |

(c)

Figure 6.2 (a) The workspace before the function call. (b) The workspace during the function call. (c) The workspace after the function call.

and `by` are defined in the workspace with the values that we have entered. When the program stops at the breakpoint *within* the function call, the function's workspace is active. It is as shown in Figure 6.2b. Note that variables `x1`, `x2`, `y1`, `y2`, and `distance` are defined in the function's workspace, and the variables defined in the calling M-file are not present. When the program stops in the calling program at the breakpoint *after* the function call, the workspace is as shown in Figure 6.2c. Now the original variables are back, with the variable `result` added to contain the value returned by the function. These figures show that the workspace of the function is different from the workspace of the calling M-file.

## 6.2 Variable Passing in MATLAB: The Pass-by-Value Scheme

MATLAB programs communicate with their functions using a **pass-by-value** scheme. When a function call occurs, MATLAB makes a *copy* of the actual arguments and passes them to the function. This copying is very significant because it means that even if the function modifies the input arguments, it won't affect the original data in the caller. This feature helps to prevent unintended side effects, in which an error in the function might unintentionally modify variables in the calling program.

This behavior is illustrated in the function shown next. This function has two input arguments: `a` and `b`. During its calculations, it modifies both input arguments.

```
function out = sample(a, b)
fprintf('In     sample: a = %f, b = %f %f\n',a,b);
a = b(1) + 2*a;
b = a .* b;
out = a + b(1);
fprintf('In     sample: a = %f, b = %f %f\n',a,b);
```

A simple test program to call this function is shown next.

```
a = 2; b = [6 4];
fprintf('Before sample: a = %f, b = %f %f\n',a,b);
out = sample(a,b);
fprintf('After  sample: a = %f, b = %f %f\n',a,b);
fprintf('After  sample: out = %f\n',out);
```

When this program is executed, the results are:

```
» test_sample
Before sample: a = 2.000000, b = 6.000000 4.000000
In     sample: a = 2.000000, b = 6.000000 4.000000
In     sample: a = 10.000000, b = 60.000000 40.000000
After  sample: a = 2.000000, b = 6.000000 4.000000
After  sample: out = 70.000000
```

Note that `a` and `b` were both changed inside function `sample`, but those changes had *no effect on the values in the calling program.*

Users of the C language will be familiar with the pass-by-value scheme, since C uses it for scalar values passed to functions. However, C does *not* use the pass-by-value scheme when passing arrays, so an unintended modification to a dummy array in a C function can cause side-effects in the calling program. MATLAB improves on this by using the pass-by-value scheme for both scalars and arrays.[2]

## ► Example 6.1—Rectangular-to-Polar Conversion

The location of a point in a Cartesian plane can be expressed in either the rectangular coordinates $(x,y)$ or the polar coordinates $(r,\theta)$, as shown in Figure 6.3. The relationships among these two sets of coordinates are given by the following equations:

$$x = r\cos\theta \tag{6.1}$$

$$x = r\sin\theta \tag{6.2}$$

$$r = \sqrt{x^2 + y^2} \tag{6.3}$$

$$\theta = \tan^{-1}\frac{y}{x} \tag{6.4}$$

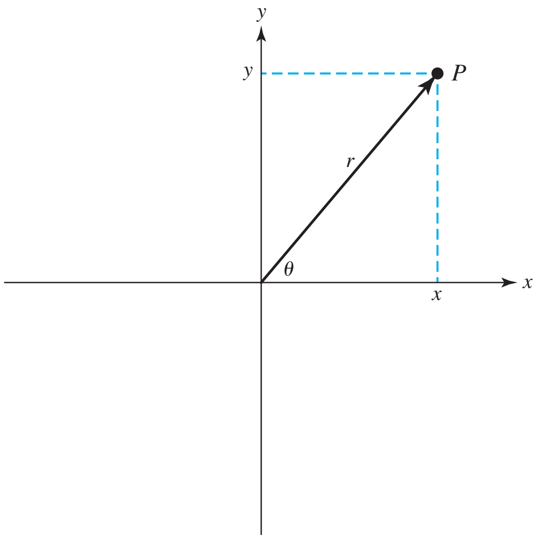


Figure 6.3 A point $P$ in a Cartesian plane can be located by either the rectangular coordinates $(x,y)$ or the polar coordinates $(r,\theta)$.

[2]The implementation of argument passing in MATLAB is actually more sophisticated than this discussion indicates. As pointed out, the copying associated with pass-by-value takes up a lot of time, but it provides protection against unintended side-effects. MATLAB actually uses the best of both approaches: it analyzes each argument of each function and determines whether or not the function modifies that argument. If the function modifies the argument, then MATLAB makes a copy of it. If it does not modify the argument, then MATLAB simply points to the existing value in the calling program. This practice increases speed while still providing protection against side effects.

Write two functions `rect2polar` and `polar2rect` that convert coordinates from rectangular to polar form, and vice versa, where the angle $\theta$ is expressed in degrees.

**Solution** We will apply our standard problem-solving approach to creating these functions. Note that MATLAB's trigonometric functions work in radians, so we must convert from degrees to radians and vice versa when solving this problem. The basic relationship between degrees and radians is

$$180^\circ = \pi \text{ radians} \tag{6.5}$$

1. **State the problem**
   A succinct statement of the problem is:

   > Write a function that converts a location on a Cartesian plane expressed in rectangular coordinates into the corresponding polar coordinates, with the angle $\theta$ expressed in degrees. Also, write a function that converts a location on a Cartesian plane expressed in polar coordinates with the angle $\theta$ expressed in degrees into the corresponding rectangular coordinates.

2. **Define the inputs and outputs**
   The inputs to function `rect2polar` are the rectangular $(x,y)$ location of a point. The outputs of the function are the polar $(r, \theta)$ location of the point. The inputs to function `polar2rect` are the polar $(r, \theta)$ location of a point. The outputs of the function are the rectangular $(x,y)$ location of the point.

3. **Describe the algorithm**
   These functions are very simple, so we can directly write the final pseudocode for them. The pseudocode for function `polar2rect` is:

```
x ← r * cos(theta * pi/180)
y ← r * sin(theta * pi/180)
```

   The pseudocode for function `rect2polar` will use the function `atan2`, because that function works over all four quadrants of the Cartesian plane. (Look up that function in the MATLAB Help Browser.)

```
r ← sqrt(x.^2 + y .^2)
theta ← 180/pi * atan2(y,x)
```

4. **Turn the algorithm into MATLAB statements**
   The MATLAB code for the selection `polar2rect` function is shown next.

```
function [x, y] = polar2rect(r,theta)
%POLAR2RECT Convert rectangular to polar coordinates
% Function POLAR2RECT accepts the polar coordinates
% (r,theta), where theta is expressed in degrees,
% and converts them into the rectangular coordinates
% (x,y).
%
% Calling sequence:
%   [x, y] = polar2rect(r,theta)
% Define variables:
%   r         -- Length of polar vector
%   theta     -- Angle of vector in degrees
%   x         -- x-position of point
%   y         -- y-position of point

%  Record of revisions:
%      Date       Programmer          Description of change
%      ====       ==========          =====================
%    02/01/18    S. J. Chapman        Original code

x = r * cos(theta * pi/180);
y = r * sin(theta * pi/180);
```

The MATLAB code for the selection `rect2polar` function is shown next.

```
function [r, theta] = rect2polar(x,y)
%RECT2POLAR Convert rectangular to polar coordinates
% Function RECT2POLAR accepts the rectangular coordinates
% (x,y) and converts them into the polar coordinates
% (r,theta), where theta is expressed in degrees.
%
% Calling sequence:
%   [r, theta] = rect2polar(x,y)

% Define variables:
%   r         -- Length of polar vector
%   theta     -- Angle of vector in degrees
%   x         -- x-position of point
%   y         -- y-position of point

%  Record of revisions:
%      Date       Programmer          Description of change
%      ====       ==========          =====================
%    02/01/18    S. J. Chapman        Original code

r = sqrt(x.^2 + y .^2);
theta = 180/pi * atan2(y,x);
```

Note that these functions both include help information, so they will work properly with MATLAB's help subsystem and with the `lookfor` command.

5. **Test the program**
   To test these functions, we will execute them directly in the MATLAB Command Window. We will test the functions using the 3-4-5 triangle, which is familiar to most people from secondary school. The smaller angle within a 3-4-5 triangle is approximately 36.87°. We will also test the function in all four quadrants of the Cartesian plane to ensure that the conversions are correct everywhere.

```
» [r, theta] = rect2polar(4,3)
r =
     5
theta =
   36.8699
» [r, theta] = rect2polar(-4,3)
r =
     5
theta =
  143.1301
» [r, theta] = rect2polar(-4,-3)
r =
     5
theta =
 -143.1301
» [r, theta] = rect2polar(4,-3)
r =
     5
theta =
  -36.8699
» [x, y] = polar2rect(5,36.8699)
x =
    4.0000
y =
    3.0000
» [x, y] = polar2rect(5,143.1301)
x =
   -4.0000
y =
    3.0000
» [x, y] = polar2rect(5,-143.1301)
x =
   -4.0000
y =
   -3.0000
» [x, y] = polar2rect(5,-36.8699)
x =
    4.0000
y =
   -3.0000
»
```

These functions appear to be working correctly in all quadrants of the Cartesian plane.

## ▶Example 6.2—Sorting Data

In many scientific and engineering applications, it is necessary to take a random input data set and to sort it so that the numbers in the data set are either all in *ascending order* (lowest-to-highest) or all in *descending order* (highest-to-lowest). For example, suppose that you were a zoologist studying a large population of animals and that you wanted to identify the largest 5 percent of the animals in the population. The most straightforward way to approach this problem would be to sort the sizes of all of the animals in the population into ascending order and take the top 5 percent of the values.

Sorting data into ascending or descending order seems to be an easy job. After all, we do it all the time. It is simple matter for us to sort the data (10, 3, 6, 4, 9) into the order (3, 4, 6, 9, 10). How do we do it? We first scan the input data list (10, 3, 6, 4, 9) to find the smallest value in the list (3), and then scan the remaining input data (10, 6, 4, 9) to find the next smallest value (4), and so forth, until the complete list is sorted.

In fact, sorting can be a very difficult job. As the number of values to be sorted increases, the time required to perform the simple sort increases rapidly, since we must scan the input data set once for each value sorted. For very large data sets, this technique just takes too long to be practical. Even worse, how would we sort the data if there were too many numbers to fit into the main memory of the computer? The development of efficient sorting techniques for large data sets is an active area of research and is the subject of whole courses.

In this example, we will confine ourselves to the simplest possible algorithm to illustrate the concept of sorting. This simplest algorithm is called the **selection sort**. It is just a computer implementation of the mental math described above. The basic algorithm for the selection sort is:

1. Scan the list of numbers to be sorted to locate the smallest value in the list. Place that value at the front of the list by swapping it with the value currently at the front of the list. If the value at the front of the list is already the smallest value, then do nothing.
2. Scan the list of numbers from position 2 to the end to locate the next smallest value in the list. Place that value in position 2 of the list by swapping it with the value currently at that position. If the value in position 2 is already the next smallest value, then do nothing.
3. Scan the list of numbers from position 3 to the end to locate the third smallest value in the list. Place that value in position 3 of the list by swapping it with the value currently at that position. If the value in position 3 is already the third smallest value, then do nothing.
4. Repeat this process until the next-to-last position in the list is reached. After the next-to-last position in the list has been processed, the sort is complete.

Note that if we are sorting N values, this sorting algorithm requires N-1 scans through the data to accomplish the sort.

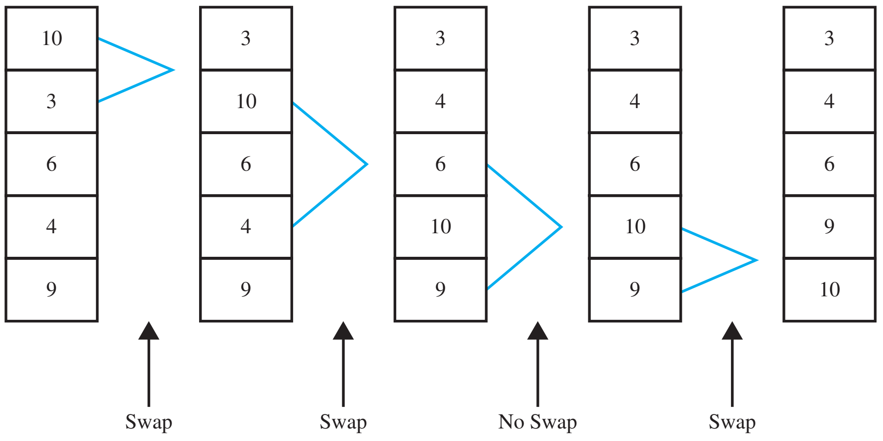


**Figure 6.4** An example problem demonstrating the selection sort algorithm.

This process is illustrated in Figure 6.4. Since there are 5 values in the data set to be sorted, we will make 4 scans through the data. During the first pass through the entire data set, the minimum value is 3, so the 3 is swapped with the 10, which was in position 1. Pass 2 searches for the minimum value in positions 2 through 5. That minimum is 4, so the 4 is swapped with the 10 in position 2. Pass 3 searches for the minimum value in positions 3 through 5. That minimum is 6, which is already in position 3, so no swapping is required. Finally, pass 4 searches for the minimum value in positions 4 through 5. That minimum is 9, so the 9 is swapped with the 10 in position 4, and the sort is completed.

## Programming Pitfalls

The selection sort algorithm is the easiest sorting algorithm to understand, but it is computationally inefficient. *It should never be applied to sort large data sets* (say, sets with more than 1000 elements). Over the years, computer scientists have developed much more efficient sorting algorithms. The `sort` and `sortrows` functions built into MATLAB are extremely efficient and should be used for all real work.

We will now develop a program to read in a data set from the Command Window, sort it into ascending order, and display the sorted data set. The sorting will be done by a separate user-defined function.

**Solution** This program must be able to ask the user for the input data, sort the data, and write out the sorted data. The design process for this problem is given next.

1. **State the problem**
   We have not yet specified the type of data to be sorted. If the data is numerical, then the problem may be stated as follows:

   Develop a program to read an arbitrary number of numerical input values from the Command Window, sort the data into ascending order using a separate sorting function, and write the sorted data to the Command Window.

2. **Define the inputs and outputs**
   The inputs to this program are the numerical values typed in the Command Window by the user. The outputs from this program are the sorted data values written to the Command Window.

3. **Describe the algorithm**
   This program can be broken down into three major steps:

```
Read the input data into an array
Sort the data in ascending order
Write the sorted data
```

   The first major step is to read in the data. We must prompt the user for the number of input data values and then read in the data. Since we will know how many input values there are to read, a `for` loop is appropriate for reading in the data. The detailed pseudocode is shown next:

```
Prompt user for the number of data values
Read the number of data values
Preallocate an input array
for ii = 1:number of values
   Prompt for next value
   Read value
end
```

   Next we have to sort the data in a separate function. We will need to make `nvals-1` passes through the data, finding the smallest remaining value each time. We will use a pointer to locate the smallest value in each pass. Once the smallest value is found, it will be swapped to the top of the list if it is not already there. The detailed pseudocode is shown next:

```
for ii = 1:nvals-1

   % Find the minimum value in a(ii) through a(nvals)
   iptr ← ii
   for jj == ii+1 to nvals
      if a(jj) < a(iptr)
         iptr ← jj
      end
   end
```

```
      % iptr now points to the min value, so swap a(iptr)
      % with a(ii) if iptr ~= ii.
      if i ~= iptr
         temp ← a(i)
         a(i) ← a(iptr)
         a(iptr) ← temp
      end
   end
```

The final step is writing out the sorted values. No refinement of the pseudocode is required for that step. The final pseudocode is the combination of the reading, sorting, and writing steps.

4. **Turn the algorithm into MATLAB statements**
   The MATLAB code for the selection sort function is shown next.

```
function out = ssort(a)
%SSORT Selection sort data in ascending order
% Function SSORT sorts a numeric data set into
% ascending order.  Note that the selection sort
% is relatively inefficient.  DO NOT USE THIS
% FUNCTION FOR LARGE DATA SETS.  Use MATLAB's
% "sort" function instead.

% Define variables:
%   a         -- Input array to sort
%   ii        -- Index variable
%   iptr      -- Pointer to min value
%   jj        -- Index variable
%   nvals     -- Number of values in "a"
%   out       -- Sorted output array
%   temp      -- Temp variable for swapping

%  Record of revisions:
%      Date       Programmer          Description of change
%      ====       ==========          =====================
%    02/02/18    S. J. Chapman        Original code

% Get the length of the array to sort
nvals = length(a);

% Sort the input array
for ii = 1:nvals-1

   % Find the minimum value in a(ii) through a(n)
   iptr = ii;
   for jj = ii+1:nvals
      if a(jj) < a(iptr)
         iptr = jj;
      end
   end

   % iptr now points to the minimum value, so swap a(iptr)
   % with a(ii) if ii ~= iptr.
   if ii ~= iptr
      temp    = a(ii);
      a(ii)   = a(iptr);
      a(iptr) = temp;
   end
end

% Pass data back to caller
out = a;
```

The program to invoke the selection sort function is shown next.

```
%  Script file: test_ssort.m
%
%  Purpose:
%    To read in an input data set, sort it into ascending
%    order using the selection sort algorithm, and to
%    write the sorted data to the Command Window.  This
%    program calls function "ssort" to do the actual
%    sorting.
%
%  Record of revisions:
%      Date       Programmer          Description of change
%      ====       ==========          =====================
%    02/02/18    S. J. Chapman        Original code
%
% Define variables:
%   array  -- Input data array
%   ii     -- Index variable
%   nvals  -- Number of input values
%   sorted -- Sorted data array

% Prompt for the number of values in the data set
nvals = input('Enter number of values to sort:  ');

% Preallocate array
array = zeros(1,nvals);
```

```
% Get input values
for ii = 1:nvals

   % Prompt for next value
   string = ['Enter value ' int2str(ii) ':  '];
   array(ii) = input(string);

end

% Now sort the data
sorted = ssort(array);

% Display the sorted result.
fprintf('\nSorted data:\n');
for ii = 1:nvals
   fprintf(' %8.4f\n',sorted(ii));
end
```

5. **Test the program**
   To test this program, we will create an input data set and run the program with it. The data set should contain a mixture of positive and negative numbers as well as at least one duplicated value to see if the program works properly under those conditions.

```
» test_ssort
Enter number of values to sort:  6
Enter value 1:  -5
Enter value 2:  4
Enter value 3:  -2
Enter value 4:  3
Enter value 5:  -2
Enter value 6:  0

Sorted data:
  -5.0000
  -2.0000
  -2.0000
   0.0000
   3.0000
   4.0000
```

   The program gives the correct answers for our test data set. Note that it works for both positive and negative numbers as well as for repeated numbers.

## 6.3 Optional Arguments

Many MATLAB functions support optional input arguments and output arguments. For example, we have seen calls to the `plot` function with as few as two or as many as seven input arguments. On the other hand, the function `max` supports either one or two output arguments. If there is only one output argument, `max` returns the maximum value of an array. If there are two output arguments, `max` returns both the maximum value and the location of the maximum value in an array. How do MATLAB functions know how many input and output arguments are present, and how do they adjust their behavior accordingly?

There are eight special functions that can be used by MATLAB functions to get information about their optional arguments and to report errors in those arguments. Six of these functions are introduced here, and the remaining two will be introduced in Chapter 10 after we learn about the cell array data type. The functions introduced now are:

- `nargin`—Returns the number of actual input arguments that were used to call the function.
- `nargout`—Returns the number of actual output arguments that were used to call the function.
- `narginchk`—Returns a standard error message if a function is called with too few or too many arguments.
- `error`—Displays an error message and aborts the function producing the error. This function is used if the argument errors are fatal.
- `warning`—Displays a warning message and continues function execution. This function is used if the argument errors are not fatal and execution can continue.
- `inputname`—Returns the actual name of the variable that corresponds to a particular argument number.

When functions `nargin` and `nargout` are called within a user-defined function, these functions return the number of actual input arguments and the number of actual output arguments that were used when the user-defined function was called.

Function `narginchk` generates an error message if a function is called with too few or too many arguments. The syntax of this function is

```
narginchk(min_args,max_args);
```

where `min_args` is the minimum number of arguments and `max_args` is the maximum number of arguments. If the number of arguments is outside the acceptable limits, a standard error message is produced. If the number of arguments is within acceptable limits, then execution continues with no error.

Function `error` is a standard way to display an error message and abort the user-defined function that is causing the error. The syntax of this function is `error('msg')`, where `msg` is a character array containing an error message. When `error` is executed, it halts the current function and returns to the keyboard, displaying the error message in the Command Window. If the message string is empty, `error` does nothing, and execution continues.

Function `warning` is a standard way to display a warning message that includes the function and line number where the problem occurred but let execution continue. The syntax of this function is `warning('msg')`, where `msg` is a character array containing a warning message. When `warning` is executed, it displays the warning message in the Command Window and lists the function name and line number where the warning came from. If the message string is empty, `warning` does nothing. In either case, execution of the function continues.

Function `inputname` returns the name of the actual argument used when a function is called. The syntax of this function is

```
name = inputname(argno);
```

where `argno` is the number of the argument. If the argument is a variable, then its name is returned. If the argument is an expression, then this function will return an empty string. For example, consider the function

```
function myfun(x,y,z)
name = inputname(2);
disp(['The second argument is named ' name]);
```

When this function is called, the results are

```
» myfun(dog,cat)
The second argument is named cat
» myfun(1,2+cat)
The second argument is named
```

Function `inputname` is useful for displaying argument names in warning and error messages.

## ▶Example 6.3—Using Optional Arguments

In this example we illustrate the use of optional arguments by creating a function that accepts an $(x,y)$ value in rectangular coordinates and produces the equivalent polar representation consisting of a magnitude and an angle in degrees. The function will be designed to support two input arguments, $x$ and $y$. However, if only one argument is supplied, the function will assume that the $y$ value is zero and proceed with the calculation. The function will normally return both the magnitude and the angle in degrees, but if only one output argument is present, it will return only the magnitude. This function is shown next:

```
function [mag, angle] = polar_value(x,y)
%POLAR_VALUE Converts (x,y) to (r,theta)
% Function POLAR_VALUE converts an input (x,y)
% value into (r,theta), with theta in degrees.
% It illustrates the use of optional arguments.

% Define variables:
%   angle    -- Angle in degrees
%   mag      -- Magnitude
%   x        -- Input x value
%   y        -- Input y value (optional)
%  Record of revisions:
%      Date       Programmer          Description of change
%      ====       ==========          =====================
%    02/03/18    S. J. Chapman        Original code

% Check for a legal number of input arguments.
narginchk(1,2);

% If the y argument is missing, set it to 0.
if nargin < 2
   y = 0;
end

% Check for (0,0) input arguments, and print out
% a warning message.
if x == 0 & y == 0
   msg = 'Both x any y are zero: angle is meaningless!';
   warning(msg);
end

% Now calculate the magnitude.
mag = sqrt(x.^2 + y.^2);

% If the second output argument is present, calculate
% angle in degrees.
if nargout == 2
   angle = atan2(y,x) * 180/pi;
end
```

We will test this function by calling it repeatedly from the Command Window. First, we will try to call the function with too few or too many arguments.

```
» [mag angle] = polar_value
Error using polar_value
Not enough input arguments.

» [mag angle] = polar_value(1,-1,1)
Error using polar_value
Too many input arguments.
```

The function provides proper error messages in both cases. Next, we will try to call the function with one or two input arguments.

```
» [mag angle] = polar_value(1)
mag =
     1
angle =
     0
```

```
» [mag angle] = polar_value(1,-1)
mag =
          1.4142
angle =
   -45
```

The function provides the correct answer in both cases. Next, we will try to call the function with one or two output arguments.

```
» mag = polar_value(1,-1)
mag =
    1.4142
» [mag angle] = polar_value(1,-1)
mag =
    1.4142
angle =
   -45
```

The function provides the correct answer in both cases. Finally, we will try to call the function with both *x* and *y* equal to zero.

```
» [mag angle] = polar_value(0,0)

Warning: Both x and y are zero: angle is meaningless!
> In d:\book\matlab\chap6\polar_value.m at line 32
mag =
     0
angle =
     0
```

In this case, the function displays the warning message, but execution continues.

Note that a MATLAB function may be declared to have more output arguments than are actually used, and this is *not* an error. The function does not actually have to check `nargout` to determine if an output argument is present. For example, consider the following function:

```
function [z1, z2] = junk(x,y)
z1 = x + y;
z2 = x - y;
end % function junk
```

This function can be called successfully with one or two output arguments.

```
» a = junk(2,1)
a =
     3
```

```
» [a b] = junk(2,1)
a =
      3
b =
      1
```

The reason for checking `nargout` in a function is to prevent useless work. If a result is going to be thrown away anyway, why bother to calculate it in the first place? You can speed up the operation of a program by not bothering with useless calculations.

## Quiz 6.1

This quiz provides a quick check to see if you have understood the concepts introduced in Sections 6.1 through 6.3. If you have trouble with the quiz, reread the sections, ask your instructor for help, or discuss the material with a fellow student. The answers to this quiz are found in the back of the book.

1. What are the differences between a script file and a function?
2. How does the `help` command work with user-defined functions?
3. What is the significance of the H1 comment line in a function?
4. What is the pass-by-value scheme? How does it contribute to good program design?
5. How can a MATLAB function be designed to have optional arguments?

For questions 6 and 7, determine whether the function calls are correct or not. If they are in error, specify what is wrong with them.

6. `out = test1(6);`

```
function res = test1(x,y)
res = sqrt(x.^2 + y.^2);
```

7. `out = test2(12);`

```
function res = test2(x,y)
narginchk(1,2));
if nargin == 2
   res = sqrt(x.^2 + y.^2);
else
   res = x;
end
```

## 6.4 Sharing Data Using Global Memory

We have seen that programs exchange data with the functions they call through an argument list. When a function is called, each actual argument is copied, and the copy is used by the function.

In addition to the argument list, MATLAB functions can exchange data with each other and with the base workspace through global memory. **Global memory** is a special type of memory that can be accessed from any workspace. If a variable is declared to be global in a function, then it will be placed in the global memory instead of the local workspace. If the same variable is declared to be global in another function, then that variable will refer to the *same memory location* as the variable in the first function. Each script file or function that declares the global variable will have access to the same data values, so *global memory provides a way to share data between functions*.

A global variable is declared with the **`global` statement**. The form of a `global` statement is

```
global var1 var2 var3 ...
```

where *`var1`*, *`var2`*, *`var3`*, and so forth are the variables to be placed in global memory. By convention, global variables are declared in all capital letters, but this is not actually a requirement.

## Good Programming Practice

Declare global variables in all capital letters to make them easy to distinguish from local variables.

Each global variable must be declared to be global before it is used for the first time in a function—it is an error to declare a variable to be global after it has already been created in the local workspace.[3] To avoid this error, it is customary to declare global variables immediately after the initial comments and before the first executable statement in a function.

## Good Programming Practice

Declare global variables immediately after the initial comments and before the first executable statement of each function that uses them.

Global variables are especially useful for sharing very large volumes of data among many functions because the entire data set does not have to be copied each time that a function is called. The downside of using global memory to exchange data among functions is that the functions will only work for that specific data set.

[3]If a variable is declared `global` after it has already been defined in a function, MATLAB will issue a warning message and then change the local value to match the global value. You should never rely on this capability, though, because future versions of MATLAB will not allow it.

A function that exchanges data through input arguments can be reused by simply calling it with different arguments, but a function that exchanges data through global memory must be modified to allow it to work with a different data set.

Global variables are also useful for sharing hidden data among a group of related functions while keeping it invisible from the invoking program unit.

## Good Programming Practice

You may use global memory to pass large amounts of data among functions within a program.

## Example 6.4—Random Number Generator

It is impossible to make perfect measurements in the real world. There will always be some *measurement noise* associated with each measurement. This fact is an important consideration in the design of systems to control the operation of such real-world devices as airplanes, refineries, and nuclear reactors. A good engineering design must take these measurement errors into account so that the noise in the measurements will not lead to unstable behavior (no plane crashes, refinery explosions, or meltdowns).

Most engineering designs are tested by running *simulations* of the operation of the system before it is built. These simulations involve creating mathematical models of the behavior of the system and feeding the models a realistic string of input data. If the models respond correctly to the simulated input data, then we can have reasonable confidence that the real-world system will respond correctly to the real-world input data.

The simulated input data supplied to the models must be corrupted by a simulated measurement noise, which is just a string of random numbers added to the ideal input data. The simulated noise is usually produced by a *random number generator*.

A random number generator is a function that will return a different and apparently random number each time it is called. Since the numbers are in fact generated by a deterministic algorithm, they only appear to be random.[4] However, if the algorithm used to generate them is complex enough, the numbers will be random enough to use in the simulation.

One simple random number generator algorithm is described next.[5] It relies on the unpredictability of the modulo function when applied to large numbers.

[4]For this reason, some people refer to these functions as *pseudorandom number generators*.

[5]This algorithm is adapted from the discussion found in Chapter 7 of *Numerical Recipes: The Art of Scientific Programming* by Press, Flannery, Teukolsky, and Vetterling, Cambridge University Press, 1986.

Recall from Chapter 2 that the modulus function `mod` returns the remainder after the division of two numbers. Consider the following equation:

$$n_{i+1} = \text{mod}(8121 n_i + 28411,\ 134456) \tag{6.6}$$

Assume that $n_i$ is a nonnegative integer. Then because of the modulo function, $n_{i+1}$ will be a number between 0 and 134455 inclusive. Next, $n_{i+1}$ can be fed into the equation to produce a number $n_{i+2}$ that is also between 0 and 134455. This process can be repeated forever to produce a series of numbers in the range [0, 134455]. If we didn't know the numbers 8121, 28411, and 134456 in advance, it would be impossible to guess the order in which the values of $n$ would be produced. Furthermore, it turns out that there is an equal (or uniform) probability that any given number will appear in the sequence. Because of these properties, Equation (6.6) can serve as the basis for a simple random number generator with a uniform distribution.

We will now use Equation (6.6) to design a random number generator whose output is a real number in the range [0.0, 1.0).[6]

**Solution** We will write a function that generates one random number in the range $0 \le \text{ran} \le 1.0$ each time that it is called. The random number will be based on the equation

$$\text{ran}_i = \frac{n_i}{134456} \tag{6.7}$$

where $n_i$ is a number in the range 0 to 134455 produced by Equation (6.7).

The particular sequence produced by Equations (6.6) and (6.7) will depend on the initial value of $n_0$ (called the *seed*) of the sequence. We must provide a way for the user to specify $n_0$ so that the sequence may be varied from run to run.

1. **State the problem**
   Write a function `random0` that will generate and return an array `ran` containing one or more numbers with a uniform probability distribution in the range $0 \le$ `ran` $< 1.0$, based on the sequence specified by Equations (6.6) and (6.7). The function should have one or two input arguments (`m` and `n`) specifying the size of the array to return. If there is one argument, the function should generate square array of size `m` × `m`. If there are two arguments, the function should generate an array of size `m` × `n`. The initial value of the seed $n_0$ will be specified by a call to a function called `seed`.

2. **Define the inputs and outputs**
   There are two functions in this problem: `seed` and `random0`. The input to function `seed` is an integer to serve as the starting point of the sequence. There is no output from this function. The input to function `random0` is one or two integers specifying the size of the array of random numbers to

[6]The notation [0.0, 1.0) implies that the range of the random numbers is between 0.0 and 1.0, including the number 0.0 but excluding the number 1.0.

be generated. If only argument m is supplied, the function should generate a square array of size m × m. If both arguments m and n are supplied, the function should generate an array of size n × m. The output from the function is the array of random values in the range [0.0, 1.0).

3. **Describe the algorithm**
   The pseudocode for function random0 is:

```
function ran = random0 ( m, n )
Check for valid arguments
Set n ← m if not supplied
Create output array with "zeros" function
for ii = 1:number of rows
   for jj = 1:number of columns
      ISEED ← mod (8121 * ISEED + 28411, 134456 )
      ran(ii,jj) ← iseed / 134456
   end
end
```

   where the value of ISEED is placed in global memory so that it is saved between calls to the function. The pseudocode for function seed is trivial:

```
function seed (new_seed)
new_seed ← round(new_seed)
ISEED ← abs(new_seed)
```

   The round function is used in case the user fails to supply an integer, and the absolute value function is used in case the user supplies a negative seed. The user will not have to know in advance that only positive integers are legal seeds.

   The variable ISEED will be placed in global memory so that it may be accessed by both functions.

4. **Turn the algorithm into MATLAB statements**
   Function random0 is shown next.

```
function ran = random0(m,n)
%RANDOM0 Generate uniform random numbers in [0,1)
% Function RANDOM0 generates an array of uniform
% random numbers in the range [0,1).  The usage
% is:
%
% random0(m)     -- Generate an m x m array
% random0(m,n)   -- Generate an m x n array

% Define variables:
%   ii       -- Index variable
%   ISEED    -- Random number seed (global)
%   jj       -- Index variable
%   m        -- Number of columns
%   n        -- Number of rows
%   ran      -- Output array
%
%  Record of revisions:
%      Date       Programmer          Description of change
%      ====       ==========          =====================
%    02/04/18    S. J. Chapman        Original code

% Declare global values
global ISEED            % Seed for random number generator

% Check for a legal number of input arguments.
narginchk(1,2);

% If the n argument is missing, set it to m.
if nargin < 2
   n = m;
end

% Initialize the output array
ran = zeros(m,n);

% Now calculate random values
for ii = 1:m
   for jj = 1:n
      ISEED = mod(8121*ISEED + 28411, 134456 );
      ran(ii,jj) = ISEED / 134456;
   end
end
```

Function seed is as follows:

```
function seed(new_seed)
%SEED Set new seed for function RANDOM0
% Function SEED sets a new seed for function
% RANDOM0.  The new seed should be a positive
% integer.

% Define variables:
%   ISEED    -- Random number seed (global)
%   new_seed -- New seed
%  Record of revisions:
%      Date       Programmer          Description of change
%      ====       ==========          =====================
%    02/04/18    S. J. Chapman        Original code
%
% Declare globl values
global ISEED            % Seed for random number generator

% Check for a legal number of input arguments.
narginchk(1,1);

% Save seed
new_seed = round(new_seed);
ISEED = abs(new_seed);
```

5. **Test the resulting MATLAB programs**
If the numbers generated by these functions are truly uniformly distributed random numbers in the range $0 \le$ `ran` $< 1.0$, then the average of many numbers should be close to 0.5 and the standard deviation of the numbers should be close to $\frac{1}{\sqrt{12}}$.

Furthermore, if the range between 0 and 1 is divided into a number of equal-size bins, the number of random values falling in each bin should be about the same. A **histogram** is a plot of the number of values falling in each bin. MATLAB function `histogram` will create and plot a histogram from an input data set, so we will use it to verify the distribution of random numbers generated by `random0`.

To test the results of these functions, we will perform the following tests:

1. Call `seed` with `new_seed` set to 1024.
2. Call `random0(4)` to see that the results appear random.
3. Call `random0(4)` to verify that the results differ from call to call.
4. Call `seed` again with `new_seed` set to 1024.
5. Call `random0(4)` to see that the results are the same as in (2) above. This verifies that the seed is being reset properly.
6. Call `random0(2,3)` to verify that both input arguments are being used correctly.
7. Call `random0(1,100000)` and calculate the average and standard deviation of the resulting data set using MATLAB functions `mean` and `std`. Compare the results to 0.5 and $\frac{1}{\sqrt{12}}$.
8. Create a histogram of the data from (7) to see if approximately equal numbers of values fall in each bin.

We will perform these tests interactively, checking the results as we go.

```
» seed(1024)
» random0(4)
ans =
    0.0598    1.0000    0.0905    0.2060
    0.2620    0.6432    0.6325    0.8392
    0.6278    0.5463    0.7551    0.4554
    0.3177    0.9105    0.1289    0.6230
» random0(4)
ans =
    0.2266    0.3858    0.5876    0.7880
    0.8415    0.9287    0.9855    0.1314
    0.0982    0.6585    0.0543    0.4256
    0.2387    0.7153    0.2606    0.8922
» seed(1024)
» random0(4)
ans =
    0.0598    1.0000    0.0905    0.2060
    0.2620    0.6432    0.6325    0.8392
    0.6278    0.5463    0.7551    0.4554
    0.3177    0.9105    0.1289    0.6230
» random0(2,3)
ans =
    0.2266    0.3858    0.5876
    0.7880    0.8415    0.9287
» edit random
» mean(arr)
ans =
    0.5001
» std(arr)
ans =
    0.2887
» histogram(arr,10)
» title('\bfHistogram of the Output of random0');
» xlabel('Bin');
» ylabel('Count');
```

The results of these tests look reasonable, so the function appears to be working. The average of the data set was 0.5001, which is close to the theoretical value of 0.5000, and the standard deviation of the data set was 0.2887, which is equal to the theoretical value of 0.2887 to the accuracy displayed. The histogram is shown in Figure 6.5, and the distribution of the random values is roughly even across all of the bins.

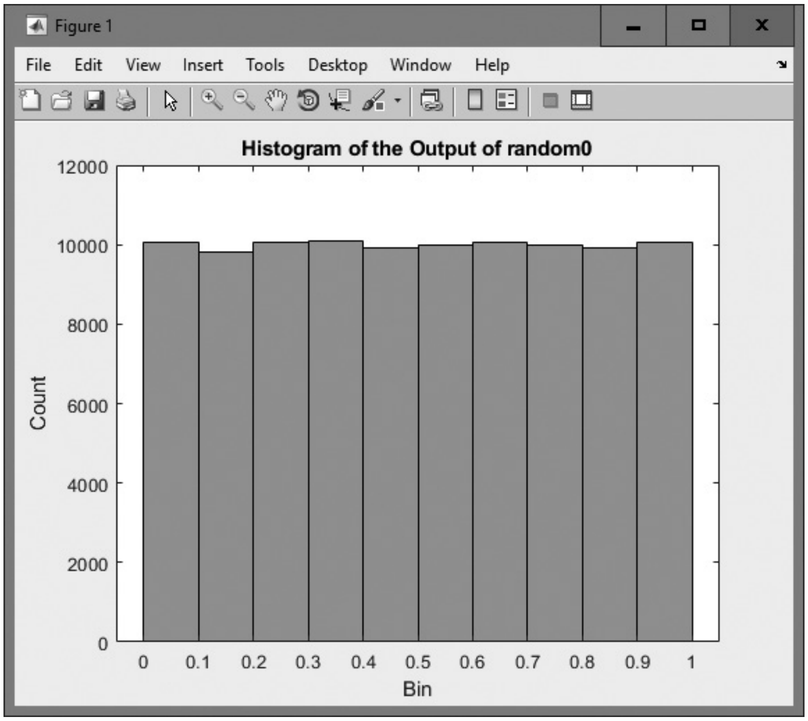


**Figure 6.5** Histogram of the output of function `random0`.

## 6.5 Preserving Data between Calls to a Function

When a function finishes executing, the special workspace created for that function is destroyed, so the contents of all local variables within the function will disappear. The next time that the function is called, a new workspace will be created, and all of the local variables will be returned to their default values. This behavior is usually desirable since it ensures that MATLAB functions behave in a repeatable fashion every time they are called.

However, it is sometimes useful to preserve some local information within a function between calls to the function. For example, we might want to create a counter to count the number of times that the function has been called. If such a counter were destroyed every time the function exited, the count would never exceed 1!

MATLAB includes a special mechanism to allow local variables to be preserved between calls to a function. **Persistent memory** is a special type of memory that can only be accessed from within the function but is preserved unchanged between calls to the function.

A persistent variable is declared with the **`persistent` statement**. The form of a global statement is

```
persistent var1 var2 var3 ...
```

where *`var1`*, *`var2`*, *`var3`*, and so forth are the variables to be placed in persistent memory.

## Good Programming Practice

Use persistent memory to preserve the values of local variables within a function between calls to the function.

## ►Example 6.5—Running Averages

It is sometimes desirable to calculate running statistics on a data set on-the-fly as the values are being entered. The built-in MATLAB functions `mean` and `std` could perform this function, but we would have to pass the entire data set to them for re-calculation after each new data value is entered. A better result can be achieved by writing a special function that keeps track of the appropriate running sums between calls and only needs the latest value to calculate the current average and standard deviation.

The average or arithmetic mean of a set of numbers is defined as

$$\bar{x} = \frac{1}{N}\sum_{i=1}^{N} x_i \tag{6.8}$$

where $x_i$ is sample $i$ out of $N$ samples. The standard deviation of a set of numbers is defined as

$$s = \sqrt{\frac{N\sum_{i=1}^{N} x_i^2 - \left(\sum_{i=1}^{N} x_i\right)^2}{N(N-1)}} \tag{6.9}$$

Standard deviation is a measure of the amount of scatter on the measurements; the greater the standard deviation, the more scattered the points in the data set are. If we can keep track of the number of values $N$, the sum of the values $\Sigma x$, and the sum of the squares of the values $\Sigma x^2$, then we can calculate the average and standard deviation at any time from Equations (6.8) and (6.9).

Write a function to calculate the running average and standard deviation of a data set as it is being entered.

**Solution** This function must be able to accept input values one at a time and keep running sums of $N$, $\Sigma x$, and $\Sigma x^2$, which will be used to calculate the current average and standard deviation. It must store the running sums in global memory so that they are preserved between calls. Finally, there must be a mechanism to reset the running sums.

1. **State the problem**
   Create a function to calculate the running average and standard deviation of a data set as new values are entered. The function must also include a feature to reset the running sums when desired.

2. **Define the inputs and outputs**
   There are two types of inputs required by this function:

   1. The character array `'reset'` to reset running sums to zero.
   2. The numerical values from the input data set, present one value per function call.

   The outputs from this function are the mean and standard deviation of the data supplied to the function so far.

3. **Design the algorithm**
   This function can be broken down into four major steps:

```
Check for a legal number of arguments
Check for a 'reset', and reset sums if present
Otherwise, add current value to running sums
Calculate and return running average and std dev
   if enough data is available. Return zeros if
   not enough data is available.
```

   The detailed pseudocode for these steps is:

```
Check for a legal number of arguments
if x == 'reset'
   n ← 0
   sum_x ← 0
   sum_x2 ← 0
else
   n ← n + 1
   sum_x ← sum_x + x
   sum_x2 ← sum_x2 + x^2
end

% Calculate ave and std
if n == 0
   ave ← 0
   std ← 0
elseif n == 1
   ave ← sum_x
   std ← 0
else
   ave ← sum_x / n
   std ← sqrt((n*sum_x2 - sum_x^2) / (n*(n-1)))
end
```

4. **Turn the algorithm into MATLAB statements**
   The final MATLAB function is shown next.

```
function [ave, std] = runstats(x)
%RUNSTATS Generate running ave / std deviation
% Function RUNSTATS generates a running average
% and standard deviation of a data set. The
% values x must be passed to this function one
% at a time. A call to RUNSTATS with the argument
% 'reset' will reset the running sums.

% Define variables:
%   ave      -- Running average
%   n        -- Number of data values
%   std      -- Running standard deviation
%   sum_x    -- Running sum of data values
%   sum_x2   -- Running sum of data values squared
%   x        -- Input value
%
%  Record of revisions:
%      Date       Programmer           Description of change
%      ====       ==========           =====================
%    02/05/18    S. J. Chapman         Original code

% Declare persistent values
persistent n            % Number of input values
persistent sum_x        % Running sum of values
persistent sum_x2       % Running sum of values squared

% Check for a legal number of input arguments.
narginchk(1,1);

% If the argument is 'reset', reset the running sums.
if x == 'reset'
   n = 0;
   sum_x = 0;
   sum_x2 = 0;
else
   n = n + 1;
   sum_x = sum_x + x;
   sum_x2 = sum_x2 + x^2;
end

% Calculate ave and std
if n == 0
   ave = 0;
   std = 0;
```

```
elseif n == 1
   ave = sum_x;
   std = 0;
else
   ave = sum_x / n;
   std = sqrt((n*sum_x2 - sum_x^2) / (n*(n-1)));
end
```

5. **Test the program**
   To test this function, we must create a script file that resets `runstats`, reads input values, calls `runstats`, and displays the running statistics. An appropriate script file is shown next:

```
%  Script file: test_runstats.m
%
%  Purpose:
%    To read in an input data set and calculate the
%    running statistics on the data set as the values
%    are read in. The running stats will be written
%    to the Command Window.
%
%  Record of revisions:
%      Date       Programmer          Description of change
%      ====       ==========          =====================
%    02/05/18    S. J. Chapman        Original code
%
% Define variables:
%   array  -- Input data array
%   ave    -- Running average
%   std    -- Running standard deviation
%   ii     -- Index variable
%   nvals  -- Number of input values
%   std    -- Running standard deviation

% First reset running sums
[ave std] = runstats('reset');

% Prompt for the number of values in the data set
nvals = input('Enter number of values in data set:  ');

% Get input values
for ii = 1:nvals

   % Prompt for next value
   string = ['Enter value ' int2str(ii) ':  '];
   x = input(string);
```

```
      % Get running statistics
      [ave std] = runstats(x);

      % Display running statistics
      fprintf('Average = %8.4f; Std dev = %8.4f\n',ave, std);

   end
```

To test this function, we will calculate running statistics by hand for a set of 5 numbers and compare the hand calculations to the results from the program. If a data set is created with the following 5 input values

3., 2., 3., 4., 2.8

then the running statistics calculated by hand would be:

| Value | $n$ | $\Sigma x$ | $\Sigma x^2$ | Average | Std_dev |
|---|---|---|---|---|---|
| 3.0 | 1 | 3.0 | 9.0 | 3.00 | 0.000 |
| 2.0 | 2 | 5.0 | 13.0 | 2.50 | 0.707 |
| 3.0 | 3 | 8.0 | 22.0 | 2.67 | 0.577 |
| 4.0 | 4 | 12.0 | 38.0 | 3.00 | 0.816 |
| 2.8 | 5 | 14.8 | 45.84 | 2.96 | 0.713 |

The output of the test program for the same data set is:

```
» test_runstats
Enter number of values in data set:  5
Enter value 1:  3
Average =   3.0000; Std dev =   0.0000
Enter value 2:  2
Average =   2.5000; Std dev =   0.7071
Enter value 3:  3
Average =   2.6667; Std dev =   0.5774
Enter value 4:  4
Average =   3.0000; Std dev =   0.8165
Enter value 5:  2.8
Average =   2.9600; Std dev =   0.7127
```

so the results check to the accuracy shown in the hand calculations.

## 6.6 Built-In MATLAB Functions: Sorting Functions

MATLAB includes two built-in sorting functions that are extremely efficient and should be used instead of the simple sort function we created in Example 6.2. These functions are enormously faster than the sort we created in Example 6.2, and the speed difference increases rapidly as the size of the data set to sort increases.

Function `sort` sorts a data set into ascending or descending order. If the data is a column or row vector, the entire data set is sorted. If the data is a two-dimensional matrix, the columns of the matrix are sorted separately.

The most common forms of the `sort` function are

```
res = sort(a);              % Sort in ascending order
res = sort(a,'ascend');     % Sort in ascending order
res = sort(a,'descend');    % Sort in descending order
```

If `a` is a vector, the data set is sorted in the specified order. For example,

```
» a = [1 4 5 2 8];
» sort(a)
ans =
     1     2     4     5     8
» sort(a,'ascend')
ans =
     1     2     4     5     8
» sort(a,'descend')
ans =
     8     5     4     2     1
```

If `b` is a matrix, the data set is sorted independently by column. For example,

```
» b = [1 5 2; 9 7 3; 8 4 6]
b =
     1     5     2
     9     7     3
     8     4     6
» sort(b)
ans =
     1     4     2
     8     5     3
     9     7     6
```

Function `sortrows` sorts a matrix of data into ascending or descending order *according to one or more specified columns*.

The most common forms of the `sortrows` function are

```
res = sortrows(a);        % Ascending sort of col 1
res = sortrows(a,n);      % Ascending sort of col n
res = sortrows(a,-n);     % Descending order of col n
```

It is also possible to sort by more than one column. For example, the statement

```
res = sortrows(a,[m n]);
```

would sort the rows by column `m`, and if two or more rows have the same value in column `m`, it would further sort those rows by column `n`.

For example, suppose `b` is a matrix as defined in the following code fragment. Then `sortrows(b)` will sort the rows in ascending order of column 1, and `sortrows(b,[2 3])` will sort the rows in ascending order of columns 2 and 3.

```
» b = [1 7 2; 9 7 3; 8 4 6]
b =
     1     7     2
     9     7     3
     8     4     6
» sortrows(b)
ans =
     1     7     2
     8     4     6
     9     7     3
» sortrows(b,[2 3])
ans =
     8     4     6
     1     7     2
     9     7     3
```

## 6.7 Built-In MATLAB Functions: Random Number Functions

MATLAB includes two standard functions that generate random values from different distributions. They are

- `rand`—Generates random values from a uniform distribution in the range [0, 1)
- `randn`—Generates random values from a normal distribution

Both of them are much faster and much more "random" than the simple function that we have created. If you really need random numbers in your programs, use one of these functions.

In a uniform distribution, every number in the range [0, 1) has an equal probability of appearing. In contrast, the normal distribution is a classic "bell-shaped curve" with the most likely number being 0.0 and a standard deviation of 1.0.

Functions `rand` and `randn` have the following calling sequences:

- `rand()`—Generates a single random value
- `rand(n)`—Generates an $n \times n$ array of random values
- `rand(m,n)`—Generates an $m \times n$ array of random values

## 6.8 Summary

In Chapter 6, we presented an introduction to user-defined functions. Functions are special types of M-files that receive data through input arguments and return results through output arguments. Each function has its own independent workspace. Each function should appear in a separate file with the same name as the function, *including capitalization*.

Functions are called by naming them in the Command Window or another M-file. The names used should match the function name exactly, including capitalization. Arguments are passed to functions using a pass-by-value scheme, meaning that MATLAB copies each argument and passes the copy to the function. This copying is important because the function can freely modify its input arguments without affecting the actual arguments in the calling program.

MATLAB functions can support varying numbers of input and output arguments. Function `nargin` reports the number of actual input arguments used in a function call, and function `nargout` reports the number of actual output arguments used in a function call.

Data can also be shared between MATLAB functions by placing the data in global memory. Global variables are declared using the `global` statement. Global variables may be shared by all functions that declare them. By convention, global variable names are written in all capital letters.

Internal data within a function can be preserved between calls to that function by placing the data in persistent memory. Persistent variables are declared using the `persistent` statement.

## 6.8.1 Summary of Good Programming Practice

Adhere to the following guidelines when working with MATLAB functions.

1. Break large program tasks into smaller, more understandable functions whenever possible.
2. Declare global variables in all capital letters to make them easy to distinguish from local variables.
3. Declare global variables immediately after the initial comments and before the first executable statement of each function that uses them.
4. You may use global memory to pass large amounts of data among functions within a program.
5. Use persistent memory to preserve the values of local variables within a function between calls to the function.

## 6.8.2 MATLAB Summary

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

### Commands and Functions

| | |
|---|---|
| `error` | Displays error message and aborts the function producing the error. This function is used if the argument errors are fatal. |
| `global` | Declares global variables. |
| `narginchk` | Returns a standard error message if a function is called with too few or too many arguments. |
| `nargin` | Returns the number of actual input arguments that were used to call the function. |
| `nargout` | Returns the number of actual output arguments that were used to call the function. |
| `persistent` | Declares persistent variables. |

(*continued*)

## Commands and Functions (*Continued*)

| | |
|---|---|
| `rand` | Generates random values from a uniform distribution. |
| `randn` | Generates random values from a normal distribution. |
| `return` | Stops executing a function and returns to caller. |
| `sort` | Sorts data in ascending or descending order. |
| `sortrows` | Sorts rows of a matrix in ascending or descending order based on a specified column. |
| `warning` | Displays a warning message and continues function execution. This function is used if the argument errors are not fatal and execution can continue. |

## 6.9 Exercises

**6.1** What is the difference between a script file and a function?

**6.2** When a function is called, how is data passed from the caller to the function, and how are the results of the function returned to the caller?

**6.3** What are the advantages and disadvantages of the pass-by-value scheme used in MATLAB?

**6.4** Modify the selection sort function developed in this chapter so that it accepts a second optional argument, which may be either `'up'` or `'down'`. If the argument is `'up'`, sort the data in ascending order. If the argument is `'down'`, sort the data in descending order. If the argument is missing, the default case is to sort the data in ascending order. (Be sure to handle the case of invalid arguments, and be sure to include the proper help information in your function.)

**6.5** The inputs to MATLAB functions `sin`, `cos`, and `tan` are in radians, and the output of functions `asin`, `acos`, `atan`, and `atan2` are in radians. Create a new set of functions `sin_d`, `cos_d`, and so forth whose inputs and outputs are in degrees. Be sure to test your functions. (*Note*: Recent versions of MATLAB have built-in functions `sind`, `cosd`, and so forth, which work with inputs in degrees instead of radians. You can evaluate your functions and the corresponding built-in functions with the same input values to verify the proper operation of your functions.)

**6.6** Write a function `f_to_c` that accepts a temperature in degrees Fahrenheit and returns the temperature in degrees Celsius. The equation is

$$T\,(\text{in }^\circ\text{C}) = \frac{5}{9}\left[T(\text{in }^\circ\text{F}) - 32.0\right] \tag{6.10}$$

**6.7** Write a function `c_to_f` that accepts a temperature in degrees Celsius and returns the temperature in degrees Fahrenheit. The equation is

$$T\,(\text{in }^\circ\text{F}) = \frac{9}{5}T\,(\text{in }^\circ\text{C}) + 32 \tag{6.11}$$

Demonstrate that this function is the inverse of the one in Exercise 6.6. In other words, demonstrate that the expression `c_to_f(f_to_c(temp))` is just the original temperature `temp`.

**6.8 Factorial Function** The factorial function is calculated from the equation

$$n! = \begin{cases} 1 & n = 0 \\ n \times (n-1) \times (n-2) \times \ldots \times 2 \times 1 & n > 0 \end{cases} \tag{6.12}$$

where $n$ is 0 or a positive integer. Write a function `factorial` that calculates the factorial function from this equation. The function should check for the proper number of input arguments and should throw an error if there are too many or too few arguments. It should also check to ensure that the input is a nonnegative integer (*Hint*: Check out the function `isinteger`) and create an error if the value is not correct.

**6.9** The area of a triangle whose three vertices are points $(x_1, y_1)$, $(x_2, y_2)$, and $(x_3, y_3)$ (see Figure 6.6) can be found from the equation

$$A = \frac{1}{2}\begin{vmatrix} x_1 & x_2 & x_3 \\ y_1 & y_2 & y_3 \\ 1 & 1 & 1 \end{vmatrix} \tag{6.13}$$

where $|\,|$ is the determinant operation. The area returned will be positive if the points are taken in counterclockwise order, and negative if the points are taken in clockwise order. This determinant can be evaluated by hand to produce the following equation:

$$A = \frac{1}{2}\left[x_1(y_2 - y_3) - x_2(y_1 - y_3) + x_3(y_1 - y_2)\right] \tag{6.14}$$

Write a function `area2d` that calculates the area of a triangle given the three bounding points $(x_1, y_1)$, $(x_2, y_2)$, and $(x_3, y_3)$ using Equation (6.14). Then test your function by calculating the area of a triangle bounded by the points (0, 0), (5, 0), and (15, 10).

**6.10** Write a new function `area2d_1` that calculates the area of a triangle directly from Equation (6.13). Create the array

$$\text{arr} = \begin{bmatrix} x_1 & x_2 & x_3 \\ y_1 & y_2 & y_3 \\ 1 & 1 & 1 \end{bmatrix} \tag{6.15}$$

and then use the function `det()` to calculate the determinant of the array `arr`. Prove that the new function produces the same result as the function created in Exercise 6.8.

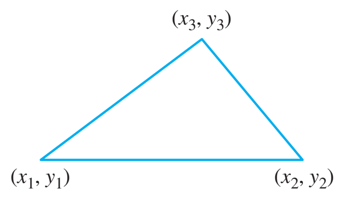


**Figure 6.6** A triangle bounded by points $(x_1,y_1)$, $(x_2,y_2)$, and $(x_3,y_3)$.

**6.11** The area inside any polygon can be broken down into a series of triangles, as shown in Figure 6.7. If there are $n$ sides to the polygon, then it can be divided into $n - 2$ triangles. Create a function that calculates the perimeter of the polygon and the area enclosed by the polygon. Use function `area2d` from the previous exercise to calculate the area of the polygon. Write a program that accepts an ordered list of points bounding a polygon and calls your function to return the perimeter and area of the polygon. Then test your function by calculating the perimeter and area of a polygon bounded by the points (0, 0), (9, 0), (8, 9), (2, 10), and (−4, 5).

**6.12** **Inductance of a Transmission Line** The inductance per meter of a single-phase, two-wire transmission line is given by the equation

$$L = \frac{\mu_0}{\pi}\left[\frac{1}{4} + \ln\left(\frac{D}{r}\right)\right] \tag{6.16}$$

where $L$ is the inductance in henrys per meter of line, $\mu_0 = 4\pi \times 10^{-7}$ H/m is the permeability of free space, $D$ is the distance between the two conductors, and $r$ is the radius of each conductor. Write a function that calculates the total inductance of a transmission line as a function of its length in kilometers, the spacing between the two conductors, and the diameter of each conductor. Use this function to calculate the inductance of a 120-km transmission line with conductors of radius $r = 2.5$ cm and distance $D = 2.0$ m.

**6.13** Based on Equation (6.16), would the inductance of a transmission line increase or decrease if the diameter of its conductors increases? How much would the inductance of the line change if the diameter of each conductor is doubled?

**6.14** **Capacitance of a Transmission Line** The capacitance per meter of a single-phase, two-wire transmission line is given by the equation

$$C = \frac{\pi\varepsilon}{\ln\left(\frac{D-r}{r}\right)} \tag{6.17}$$

where $C$ is the capacitance in farads per meter of line, $\varepsilon_0 = 4\pi \times 10^{-7}$F/m is the permittivity of free space, $D$ is the distance between the two conductors, and $r$ is the radius of each conductor. Write a function that calculates the total capacitance of a transmission line as a function of its length in kilometers, the spacing between the two conductors, and the diameter of each conductor. Use

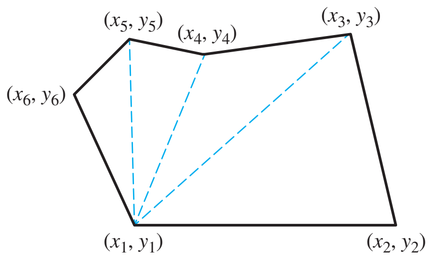


**Figure 6.7** An arbitrary polygon can be divided into a series of triangles. If there are $n$ sides to the polygon, then it can be divided into $n - 2$ triangles.

this function to calculate the capacitance of a 120-km transmission line with conductors of radius $r = 2.5$ cm and distance $D = 2.0$ m.

**6.15** What happens to the inductance and capacitance of a transmission line as the distance between the two conductors increases?

**6.16** Use function `random0` to generate a set of 100,000 random values. Sort this data set twice, once with the `ssort` function of Example 6.2, and once with MATLAB's built-in `sort` function. Use `tic` and `toc` to time the two sort functions. How do the sort times compare? (*Note*: Be sure to copy the original array and present the same data to each sort function. To have a fair comparison, both functions must get the same input data set.)

**6.17** Try the sort functions in Exercise 6.16 for array sizes of 10,000, 100,000, and 200,000. How does the sorting time increase with data set size for the sort function of Example 6.2? How does the sorting time increase with data set size for the built-in `sort` function? Which function is more efficient?

**6.18** Modify function `random0` so that it can accept 0, 1, or 2 calling arguments. If it has no calling arguments, it should return a single random value. If it has 1 or 2 calling arguments, it should behave as it currently does.

**6.19** As function `random0` is currently written, it will fail if function `seed` is not called first. Modify function `random0` so that it will function properly with some default seed even if function `seed` is never called.

**6.20** **Dice Simulation** It is often useful to be able to simulate the throw of a fair die. Write a MATLAB function `dice` that simulates the throw of a fair die by returning some random integer between 1 and 6 every time that it is called. (*Hint*: Call `random0` to generate a random number. Divide the possible values out of `random0` into six equal intervals, and return the number of the interval that a given random value falls into.)

**6.21** **Road Traffic Density** Function `random0` produces a number with a *uniform* probability distribution in the range [0.0, 1.0). This function is suitable for simulating random events if each outcome has an equal probability of occurring. However, in many events, the probability of occurrence is *not* equal for every event, and a uniform probability distribution is not suitable for simulating such events.

For example, when traffic engineers studied the number of cars passing a given location in a time interval of length $t$, they discovered that the probability of $k$ cars passing during the interval is given by the equation

$$P(k, t) = e^{-\lambda t}\frac{(\lambda t)^k}{k!} \text{ for } t \geq 0,\ \lambda > 0, \text{ and } k = 0, 1, 2, \ldots \qquad (6.18)$$

This probability distribution is known as the *Poisson distribution*; it occurs in many applications in science and engineering. For example, the number of calls $k$ to a telephone switchboard in time interval $t$, the number of bacteria $k$ in a specified volume $t$ of liquid, and the number of failures $k$ of a complicated system in time interval $t$ all have Poisson distributions.

Write a function to evaluate the Poisson distribution for any $k$, $t$, and $\lambda$. Test your function by calculating the probability of 0, 1, 2, … , 5 cars passing a particular point on a highway in 1 minute, given that $\lambda$ is 1.5 per minute for that highway. Plot the Poisson distribution for $t = 1$ and $\lambda = 1.5$.

**6.22** Write three MATLAB functions to calculate the hyperbolic sine, cosine, and tangent functions:

$$\sinh(x) = \frac{e^x - e^{-x}}{2} \qquad \cosh(x) = \frac{e^x + e^{-x}}{2} \qquad \tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

Use your functions to plot the shapes of the hyperbolic sine, cosine, and tangent functions.

**6.23** Compare the results of the functions created in Exercise 6.22 with the built-in functions `sinh`, `cosh`, and `tanh`.

$$\sinh(x) = \frac{e^x - e^{-x}}{2} \qquad \cosh(x) = \frac{e^x + e^{-x}}{2} \qquad \tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

Use your functions to plot the shapes of the hyperbolic sine, cosine, and tangent functions.

**6.24** Write a MATLAB function to perform a running average filter on a data set, as described in Exercise 5.19. Test your function using the same data set used in Exercise 5.19.

**6.25** Write a MATLAB function to perform a median filter on a data set, as described in Exercise 5.20. Test your function using the same data set used in Exercise 5.20.

**6.26** **Sort with Carry** It is often useful to sort an array `arr1` into ascending order while simultaneously carrying along a second array `arr2`. In such a sort, each time an element of array `arr1` is exchanged with another element of `arr1`, the corresponding elements of array `arr2` are also swapped. When the sort is over, the elements of array `arr1` are in ascending order, while the elements of array `arr2` that were associated with particular elements of array `arr1` are still associated with them. For example, suppose we have the following two arrays:

| Element | arr1 | arr2 |
|---|---|---|
| 1. | 6. | 1. |
| 2. | 1. | 0. |
| 3. | 2. | 10. |

After sorting array `arr1` while carrying along array `arr2`, the contents of the two arrays will be:

| Element | arr1 | arr2 |
|---|---|---|
| 1. | 1. | 0. |
| 2. | 2. | 10. |
| 3. | 6. | 1. |

Write a function to sort one real array into ascending order while carrying along a second one. Test the function with the following two 9-element arrays:

```
a = [ 1, 11, -6, 17, -23, 0, 5,  1, -1];
b = [ 31, 101, 36, -17,  0, 10, -8, -1, -1];
```

**6.27** The sort-with-carry function of Exercise 6.26 is a special case of the built-in function `sortrows`, where the number of columns is two. Create a single matrix `c` with two columns consisting of the data in vectors `a` and `b` in the previous exercise, and sort the data using `sortrows`. How does the sorted data compare to the results of Exercise 6.26?

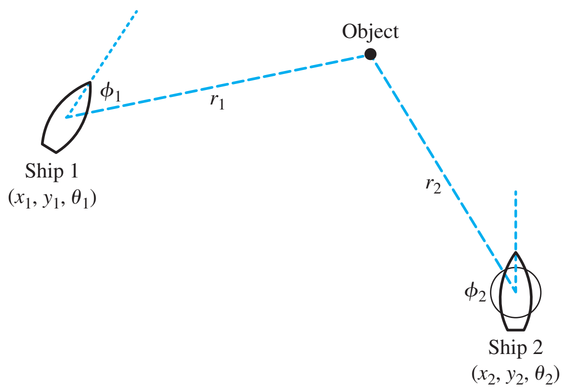


**Figure 6.8** Two ships at positions $(x_1, y_1)$ and $(x_2, y_2)$, respectively. Ship 1 is traveling at heading $\theta_1$, and Ship 2 is traveling at heading $\theta_2$.

**6.28** Compare the performance of `sortrows` with the sort-with-carry function created in Exercise 6.26. To do this, create two copies of a 10,000 × 2 element array containing random values, and sort column 1 of each array while carrying along column 2 using both functions. Determine the execution times of each sort function using `tic` and `toc`. How does the speed of your function compare with the speed of the standard function `sortrows`?

**6.29** Figure 6.8 shows two ships steaming on the ocean. Ship 1 is at position $(x_1, y_1)$ and steaming on heading $\theta_1$. Ship 2 is at position $(x_2, y_2)$ and steaming on heading $\theta_2$. Suppose that Ship 1 makes radar contact with an object at range $r_1$ and bearing $\phi_1$. Write a MATLAB function that will calculate the range $r_2$ and bearing $\phi_2$ at which Ship 2 should see the object.

**6.30** **Linear Least-Squares Fit** Develop a function that will calculate slope $m$ and intercept $b$ of the least-squares line that best fits an input data set. The input data points $(x,y)$ will be passed to the function in two input arrays, `x` and `y`. (The equations describing the slope and intercept of the least-squares line are given in Example 5.6 of Chapter 5.) Test your function using a test program and the following 20-point input data set:

**Sample Data to Test Least-Squares Fit Routine**

| No. | x | y | No. | x | y |
|---|---|---|---|---|---|
| 1 | −4.91 | −8.18 | 11 | −0.94 | 0.21 |
| 2 | −3.84 | −7.49 | 12 | 0.59 | 1.73 |
| 3 | −2.41 | −7.11 | 13 | 0.69 | 3.96 |
| 4 | −2.62 | −6.15 | 14 | 3.04 | 4.26 |
| 5 | −3.78 | −6.62 | 15 | 1.01 | 6.75 |
| 6 | −0.52 | −3.30 | 16 | 3.60 | 6.67 |
| 7 | −1.83 | −2.05 | 17 | 4.53 | 7.70 |
| 8 | −2.01 | −2.83 | 18 | 6.13 | 7.31 |
| 9 | 0.28 | −1.16 | 19 | 4.43 | 9.05 |
| 10 | 1.08 | 0.52 | 20 | 4.12 | 10.95 |

**6.31 Correlation Coefficient of Least-Squares Fit** Develop a function that will calculate both the slope *m* and intercept *b* of the least-squares line that best fits an input data set, and also the correlation coefficient of the fit. The input data points (*x*,*y*) will be passed to the function in two input arrays, `x` and `y`. The equations describing the slope and intercept of the least-squares line are given in Example 5.1, and the equation for the correlation coefficient is

$$r = \frac{n\left(\sum xy\right) - \left(\sum x\right)\left(\sum y\right)}{\sqrt{\left[\left(n\sum x^2\right) - \left(\sum x\right)^2\right]\left[\left(n\sum y^2\right) - \left(\sum y\right)^2\right]}} \tag{6.19}$$

where
$\Sigma x$ is the sum of the $x$ values
$\Sigma y$ is the sum of the $y$ values
$\Sigma x^2$ is the sum of the squares of the $x$ values
$\Sigma y^2$ is the sum of the squares of the $y$ values
$\Sigma xy$ is the sum of the products of the corresponding $x$ and $y$ values
$n$ is the number of points included in the fit

Test your function using a test driver program and the 20-point input data set given in the previous problem.

**6.32** Create a function `random1` that uses function `random0` to generate uniform random values in the range $[-1, 1)$. Test your function by calculating and displaying 20 random samples.

**6.33 Gaussian (Normal) Distribution** Function `random0` returns a uniformly distributed random variable in the range [0, 1), which means that there is an equal probability of any given number in the range occurring on a given call to the function. Another type of random distribution is the Gaussian distribution, in which the random value takes on the classic bell-shaped curve shown in Figure 6.9. A Gaussian distribution with an average of 0.0 and a standard deviation of 1.0 is called a *standardized normal distribution*, and the probability of any given value occurring in the standardized normal distribution is given by the equation

$$p(x) = \frac{1}{\sqrt{2\pi}} e^{-x^2/2} \tag{6.20}$$

It is possible to generate a random variable with a standardized normal distribution starting from a random variable with a uniform distribution in the range $[-1, 1)$ as follows:

1. Select two uniform random variables $x_1$ and $x_2$ from the range $[-1, 1)$ such that $x_1^2 + x_2^2 < 1$. To do this, generate two uniform random variables in the range $[-1, 1)$, and see if the sum of their squares happens to be less than 1. If so, use them. If not, try again.
2. Then, each of the values $y_1$ and $y_2$ in the following equations will be a normally distributed random variable.

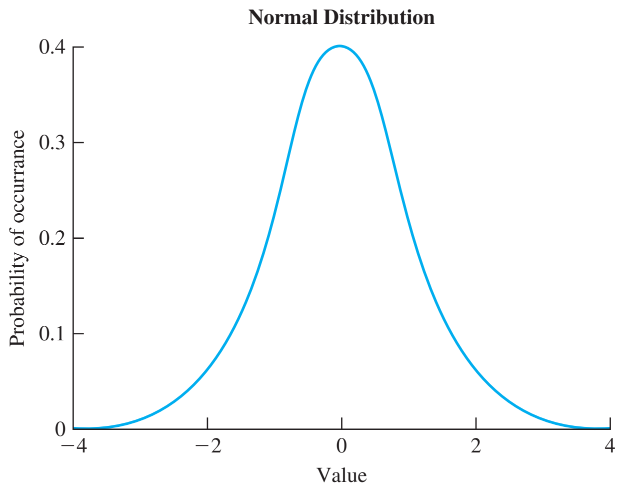


**Figure 6.9** A normal probability distribution.

$$y_1 = \sqrt{\frac{-2\ln r}{r}}\, x_1 \tag{6.21}$$

$$y_2 = \sqrt{\frac{-2\ln r}{r}}\, x_2 \tag{6.22}$$

where $$r = x_1^2 + x_2^2 \tag{6.23}$$

Write a function that returns a normally distributed random value each time it is called. Test your function by getting 1000 random values, calculating the standard deviation, and plotting a histogram of the distribution. How close to 1.0 was the standard deviation?

**6.34** Compare the Gaussian distribution function generated by the function created in Exercise 6.33 with the built-in MATLAB function `randn`. Create a 100,000-element array with each function, and create a histogram of each distribution with 21 bins. How do the two distributions compare?

**6.35** **Gravitational Force** The gravitational force $F$ between two bodies of masses $m_1$ and $m_2$ is given by the equation

$$F = \frac{Gm_1m_2}{r^2} \tag{6.24}$$

where $G$ is the gravitation constant ($6.672 \times 10^{-11}$ N-m$^2$/kg$^2$), $m_1$ and $m_2$ are the masses of the bodies in kilograms, and $r$ is the distance between the two bodies. Write a function to calculate the gravitational force between two bodies given their masses and the distance between them. Test your function by determining the force on an 800 kg satellite in orbit 38,000 km above the Earth. (The mass of the Earth is $6.98 \times 10^{24}$ kg.)

**6.36 Rayleigh Distribution** The Rayleigh distribution is another random number distribution that appears in many practical problems. A Rayleigh-distributed random value can be created by taking the square root of the sum of the squares of two normally distributed random values. In other words, to generate a Rayleigh-distributed random value *r*, get two normally distributed random values ($n_1$ and $n_2$), and perform the following calculation:

$$r = \sqrt{n_1^{\,2} + n_2^{\,2}} \tag{6.25}$$

(a) Create a function `rayleigh(n,m)` that returns an `n × m` array of Rayleigh-distributed random numbers. If only one argument is supplied [`rayleigh(n)`], the function should return an `n × n` array of Rayleigh-distributed random numbers. Be sure to design your function with input argument checking and with proper documentation for the MATLAB help system.

(b) Test your function by creating an array of 20,000 Rayleigh-distributed random values and plotting a histogram of the distribution. What does the distribution look like?

(c) Determine the mean and standard deviation of the Rayleigh distribution.