CHAPTER 5

User-Defined Functions

In Chapter 3, we learned the importance of good program design. The basic technique that we employed was **top-down design**. In top-down design, the programmer starts with a statement of the problem to be solved and the required inputs and outputs. Next, he or she describes the algorithm to be implemented by the program in broad outline, and applies *decomposition* to break the algorithm down into logical subdivisions called sub-tasks. Then, the programmer breaks down each sub-task until he or she winds up with many small pieces, each of which does a simple, clearly understandable job. Finally, the individual pieces are turned 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 even 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 needs to be debugged only 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, because accidental programming mistakes within a function can affect only the variables within function in which the mistake occurred.

Once a large program has been written and released, it must be *maintained*. Program maintenance involves fixing bugs and modifying the program to handle new and unforeseen circumstances. The programmer who modifies a program during maintenance is often not the person who originally wrote it. In poorly written programs, it is common for the programmer 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 programmer 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 the 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.

5.1 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 extent "`.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 may 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.

Execution begins at the top of the function and ends when either a `return` statement, an `end` statement, or the end of the function is reached. Because execution stops at the end of a function anyway, 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 of MATLAB 7.0. In earlier versions of MATLAB, the `end` statement was only used to terminate structures such as `if, for, while`, etc. It is optional in MATLAB 7 unless a file includes nested functions, which are covered later in this chapter. In this book, we will always terminate functions with an `end` statement, and we will include a comment on each `end` statement naming the function that it is associated with.[2] MATLAB doesn't use the comment, but it is helpful to a programmer trying to read your code at a later date.

✱ Good Programming Practice

Always terminate your functions with an `end` statement, and include a comment on the statement indicating which function the `end` statement is associated with.

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 below. Function `dist2` calculates the distance between points (x_1, y_1) and (x_2, y_2) in a Cartesian coordinate system.

[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, this will produce an error on Linux, Unix, and Macintosh computers, and a warning on Windows-based computers.

[2] The `end` statements at the end of a function will cause a warning if the function is executed on versions of MATLAB that preceded MATLAB 7.0.

```
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
%     ====        ==========          =====================
%   01/12/04    S. J. Chapman       Original code

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

end % function distance
```

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

```
% Script file: test_dist2.m
%
% Purpose:
%   This program tests function dist2.
%
% Record of revisions:
%     Date        Programmer          Description of change
%     ====        ==========          =====================
%   01/12/04    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.

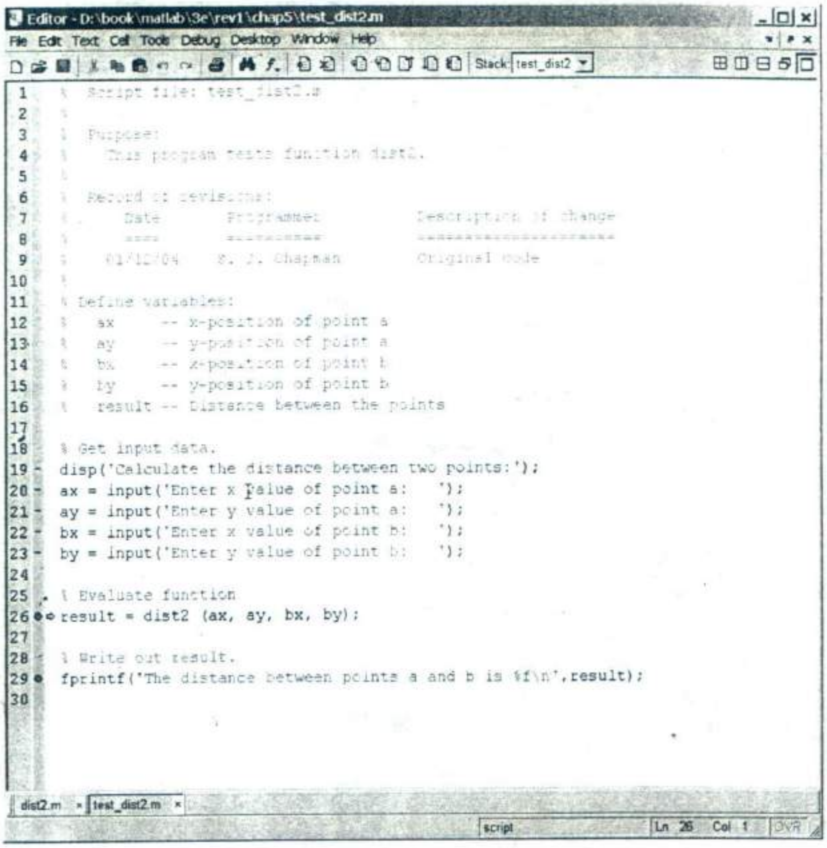
Function `dist2` also supports the MATLAB help subsystem. If we type "`help dist2`," the results are:

```
» 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
MAHAL Mahalanobis distance.
DIST Distances between vectors.
NBDIST Neighborhood matrix using vector distance.
NBGRID Neighborhood matrix using grid distance.
NBMAN Neighborhood matrix using Manhattan-distance.
```

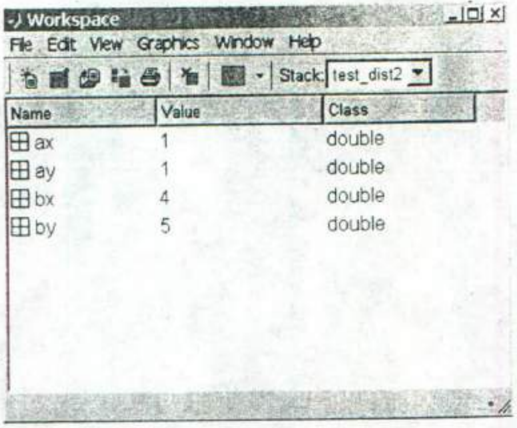
To observe the behavior of the MATLAB workspace before, during, and after the function is executed, we will load function `dist2` and the script file `test_dist2` into the MATLAB debugger, and set breakpoints before, during and after the function call (see Figure 5.1). When the program stops at the breakpoint


Editor - D:\book\matlab\3e\rev1\chap5\test_dist2.m
File Edit Text Cell Tools Debug Desktop Window Help
Stack: test_dist2
dist2.m
test_dist2.m
script
Ln 26 Col 1 OVR

```
%  Script file: test_dist2.m
%
%  Purpose:
%    This program tests function dist2.
%
%  Record of revisions:
%      Date        Programmer            Description of change
%      ====        ==========            =====================
%    01/12/04    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);
```

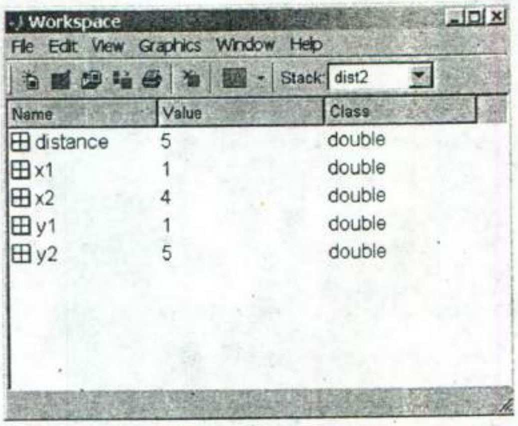
Figure 5.1 M-file `test_dist2` and function `dist2` are loaded into the debugger, with breakpoints set before, during, and after the function call.

before the function call, the workspace is as shown in Figure 5.2(a). Note that variables `ax`, `ay`, `bx`, 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 5.2(b). 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 5.2(c). 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.



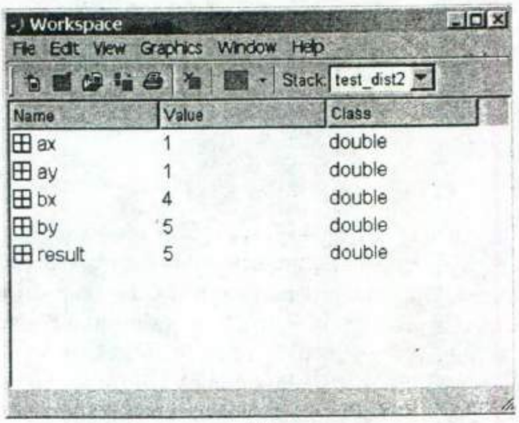
Name	Value	Class
ax	1	double
ay	1	double
bx	4	double
by	5	double

(a)



Name	Value	Class
distance	5	double
x1	1	double
x2	4	double
y1	1	double
y2	5	double

(b)



Name	Value	Class
ax	1	double
ay	1	double
bx	4	double
by	5	double
result	5	double

(c)

Figure 5.2 *(a)* The workspace before the function call. *(b)* The workspace during the function call. *(c)* The workspace after the function call.

5.2 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 highly 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 below. This function has two input arguments: a and b. During its calculations, it modifies both input arguments.

```
function out = sample(a, b, c)
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 below.

```
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 the 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.[3]

[3]The implementation of argument passing in MATLAB is actually more sophisticated than this discussion indicates. As pointed out previously, 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!

:xample 5.1—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, θ), as shown in Figure 5.3. The relationships among these two sets of coordinates are given by the following equations:

$$x = r\cos\theta \tag{5-1}$$

$$y = r\sin\theta \tag{5-2}$$

$$r = \sqrt{x^2 + y^2} \tag{5-3}$$

$$\theta = \tan^{-1}\frac{y}{x} \tag{5-4}$$

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

Solution We will apply our standard problem-solving approacn 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

$$180^\circ = \pi \text{ radians} \tag{5-5}$$

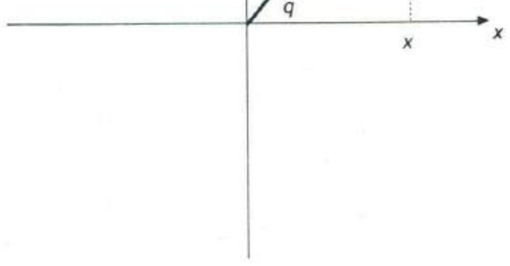


Figure 5.3 A point P in a Cartesian plane can be located by either the rectangular coordinates (x, y) or the polar coordinates (r, θ).

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 θ is expressed in degrees. Also, write a function that converts a location on a Cartesian plane expressed in polar coordinates with the angle θ is 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, θ) location of the point. The inputs to function `polar2rect` are the polar (r, θ) 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 that function up 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 below.

```
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
%     ====          ==========          =====================
%   01/12/04      S. J. Chapman         Original code

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

end % function polar2rect
```

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

```
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
%     ====          ==========          =====================
%   01/12/04      S. J. Chapman         Original code

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

end % function polar2rect
```

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 conversion 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 5.2—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 you were a zoologist studying a large population of animals and 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 described 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 all by itself.

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.

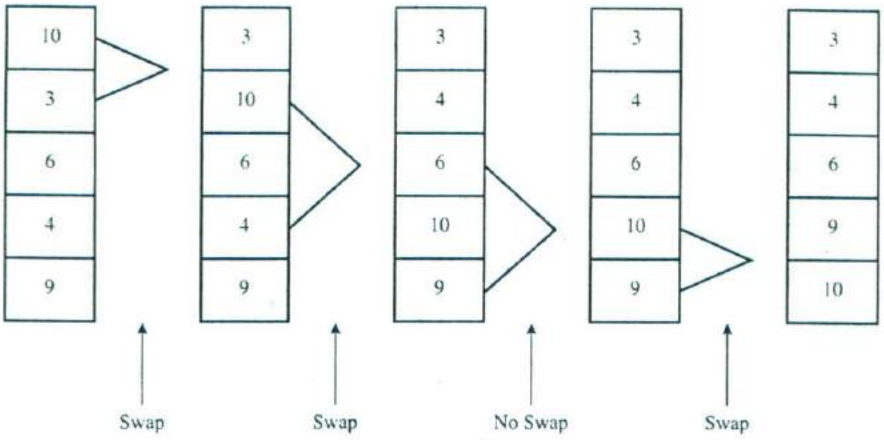


Figure 5.4 An example problem demonstrating the selection sort algorithm.

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

This process is illustrated in Figure 5.4. Since there are five values in the data set to be sorted, we will make four 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 below.

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

   Develop a program to read an arbitrary number of numeric 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 numeric 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 below:

   ```
   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 of it is not already there. The detailed pseudocode is shown below:

```
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 below.

```
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          Programme          Description of change
%     ====          ==========         =====================
%   01/12/04      S. J. Chapman      Original code

% Get the length of the array to sort
nvals = size(a,2);

% 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;

   end % function ssort
```

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

```
% 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
%      ====         ==========          =====================
%    01/12/04     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 whether 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. ◄

5.3 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 7 after we learn about the cell array data type. The functions introduced now are:

- `nargin`—This function returns the number of actual input arguments that were used to call the function.
- `nargout`—This function returns the number of actual output arguments that were used to call the function.
- `nargchk`—This function returns a standard error message if a function is called with too few or too many arguments.
- `error`—Display error message and abort the function producing the error. This function is used if the argument errors are fatal.
- `warning`—Display warning message and continue function execution. This function is used if the argument errors are not fatal, and execution can continue.
- `inputname`—This function 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 `nargchk` generates a string containing a standard error message if a function is called with too few or too many arguments. The syntax of this function is

```
message = nargchk(min_args,max_args,num_args);
```

where `min_args` is the minimum number of arguments, `max_args` is the maximum number of arguments, and `num_args` is the actual 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 an empty string is returned.

Function `error` is a standard way to display an error message and abort the user-defined function causing the error. The syntax of this function is `error('msg')`, where `msg` is a character string 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. This function works well with `nargchk`, which produces a message string when an error occurs and an empty string when there is no error.

Function `warning` is a standard way to display a warning message that includes the function and line number where the problem occurred, but lets execution continue. The syntax of this function is `warning('msg')`, where `msg` is a character string 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 5.3—Using Optional Arguments

We will 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 below.

```
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
%   msg      -- Error message
%   mag      -- Magnitude
%   x        -- Input x value
%   y        -- Input y value (optional)

% Record of revisions:
%     Date        Programmer         Description of change
%     ====        ==========         =====================
%   01/12/04    S. J. Chapman        Original code

% Check for a legal number of input arguments.
msg = nargchk(1,2,nargin);
error(msg);

% 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, calcuate
% angle in degrees.
if nargout == 2
   angle = atan2(y,x) * 180/pi;
end

end % function polar_value
```

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 any y are zero: angle is meaningless!
> In d:\book\matlab\chap5\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? A programmer can speed up the operation of a program by not bothering with useless calculations.

Quiz 5.1

This quiz provides a quick check to see if you have understood the concepts introduced in Sections 5.1 through 5.3. If you have trouble with the quiz, reread the section, ask your instructor, 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);
end % function test1
```

7. `out = test2(12);`

```
function res = test2(x,y)
error(nargchk(1,2,nargin));
```

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

5.4 Sharing Data Using Global Memory

We have seen that programs exchange data with the functions they call through a argument lists. 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 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`*, etc. 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 letter 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.[4] To avoid this error, it is customary

[4]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.

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 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. 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 actually 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 5.4—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. 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 or refinery explosions!).

Most engineering designs are tested by running *simulations* of the operation of the system before it is ever 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.[5] 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 below.[6] It relies on the unpredictability of the modulo function when applied to large numbers. Consider the following equation:

$$n_{i+1} = \mathrm{mod}\,(8121\,n_i + 28411, 134456) \tag{5-6}$$

Assume that n_i is a nonnegative integer. Then because of the modulo function, n_{i+1} will be a number between 0 and 134,455 inclusive. Next, n_{i+1} can be fed into the equation to produce a number n_{i+2} that is also between 0 and 134,455. 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, 28,411, and 134,456 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 (5-6) can serve as the basis for a simple random number generator with a uniform distribution.

We will now use Equation (5-6) to design a random number generator whose output is a real number in the range [0.0, 1.0)[7].

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

$$ran_i = \frac{n_i}{134456} \tag{5-7}$$

where n_i is a number in the range 0 to 134455 produced by Equation (5-7).

The particular sequence produced by Equations (5-6) and (5-7) 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 (5-6) and (5-7). The function should have one or two input arguments (`n` and `m`) spec-

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

[6]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.

[7]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.

ifying the size of the array to return. If there is one argument, the function should generate square array of size n × n. If there are two arguments, the function should generate an array of size n × m. 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 be generated. If only argument m is supplied, the function should generate a square array of size n × n. 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 (n, m)
Check for valid arguments
Set m <- n 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 below.

```
function ran = random0(n,m)
%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(n)     -- Generate an n x n array
% random0(n,m)   -- Generate an n x m array

% Define variables:
%   ii         -- Index variable
%   ISEED      -- Random number seed (global)
%   jj         -- Index variable
%   m          -- Number of columns
%   msg        -- Error message
%   n          -- Number of rows
%   ran        -- Output array

% Record of revisions:
%     Date          Programmer         Description of change
%     ====          ==========         =====================
%   01/12/04      S. J. Chapman      Original code
%
% Declare globl values
global ISEED        % Seed for random number generator

% Check for a legal number of input arguments.
msg = nargchk(1,2,nargin);
error(msg);

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

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

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

end % function random0
```

Function seed is shown below.

```
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
%     ====          ==========          =====================
%   01/12/04      S. J. Chapman         Original code
%
% Declare globl values
global ISEED       % Seed for random number generator

% Check for a legal number of input arguments.
msg = nargchk(1,1,nargin);
error(msg);

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

end % function seed
```

5. **Test the resulting MATLAB programs.**

If the numbers generated by these functions are truly uniformly distributed random numbers in the range $0 \leq$ `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, the if the range between 0 and 1 is divided into a number of bins of equal size, 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 `hist` will create and plot a histogram from an input data set, so we will use it to verify the distribution of random number 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 properly being reset.
6. Call `random0(2,3)` to verify that both input arguments are being used correctly.

7. Call `random0(1,20000)` 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
» arr = random0(1,20000);
» mean(arr)
ans =
    0.5020
» std(arr)
ans =
    0.2881
» hist(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 0.5020, which is quite

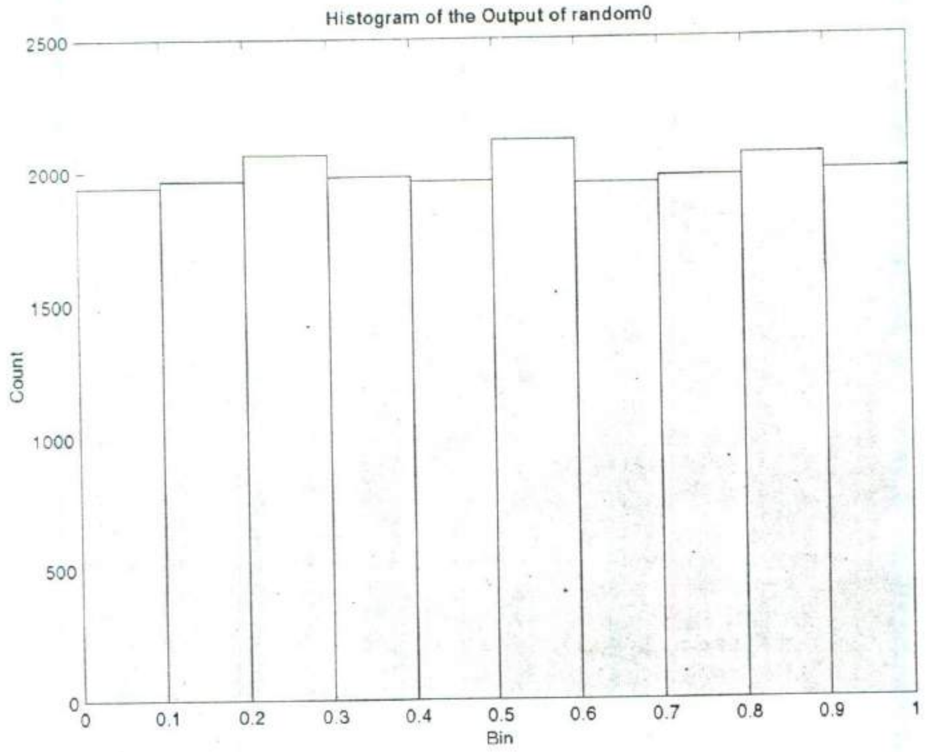


Figure 5.5 Histogram of the output of function `random0`.

close to the theoretical value of 0.5000, and the standard deviation of the data set was 0.2881, which is quite close to the theoretical value of 0.2887. The histogram is shown in Figure 5.5, and the distribution of the random values is roughly even across all of the bins.

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

- `rand`—Generates random values from a uniform distribution on 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.

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(n,m)`—Generates an $n \times m$ array of random values.

5.5 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 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 which 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 be accessed only 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,`* etc. 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 5.5—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 recalculation after each new data value is entered. A better result can be achieved by writing a special function that keeps tracks 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

$$\bar{x} = \frac{1}{N}\sum_{i=1}^{N} x_i \tag{5-8}$$

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

$$s = \sqrt{\frac{N\sum_{i=1}^{N} x_i^2 - \left(\sum_{i=1}^{N} x_i\right)^2}{N(N-1)}} \tag{5-9}$$

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 $\sum x$, and the sum of the squares of the values $\sum x^2$, then we can calculate the average and standard deviation at any time from Equations (5-8) and (5-9).

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, $\sum x$, and $\sum x^2$, 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 string `'reset'` to reset running sums to zero.
   2. The numeric values from the input data set, presented 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, as follows:

```
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 sd
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 below.

```
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 tue running sums.

% Define variables:
%   ave      -- Running average
%   msg      -- Error message
%   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
%     ====          ==========          =====================
%   01/13/04      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.
msg = nargchk(1,1,nargin);
error(msg);

% 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 sd
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

end % function runstats
```

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 below.

```
% 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
%     ====          ==========          =====================
%   01/13/04      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 five numbers, and compare the hand calculations to the results from the program. If a data set is created with the following five input values

$$3.,\quad 2.,\quad 3.,\quad 4.,\quad 2.8 \qquad (2\text{-}8)$$

then the running statistics calculated by hand would be:

Value	n	$\sum x$	$\sum x^2$	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. ◀

5.6 Function Functions

Function functions are functions whose input arguments include the names of other functions. The functions whose names are passed to the function function are normally used during the function's execution.

For example, MATLAB contains a function function called `fzero`. This function locates a zero of the function that is passed to it. For example, the statement `fzero('cos',[0 pi])` locates a zero of the function `cos` between 0 and π, and `fzero('exp(x)-2',[0 1])` locates a zero of the function `exp(x)-2` between 0 and 1. When these statements are executed, the result is:

```
» fzero('cos',[0 pi])
ans =
   1.5708
» fzero('exp(x)-2',[0 1])
ans =
   0.6931
```

The keys to the operation of function functions are two special MATLAB functions, `eval` and `feval`. Function `eval` *evaluates a character string* as though it had been typed in the Command Window, while function `feval` *evaluates a named function* at a specific input value.

Function `eval` evaluates a character string as though it has been typed in the Command Window. This function gives MATLAB functions a chance to construct executable statements during execution. The form of the `eval` function is

```
eval(string)
```

For example, the statement `x = eval('sin(pi/4)')` produces the result

```
» x = eval('sin(pi/4)')
x =
    0.7071
```

An example where a character string is constructed and evaluated using the `eval` function is shown below:

```
x = 1;
str = ['exp(' num2str(x)') -1'];
res = eval(str);
```

In this case, `str` contains the character string `'exp(1) -1,'` which `eval` evaluates to get the result 1.7183.

Function `feval` evaluates a *named function* defined by an M-file at a specified input value. The general form of the `feval` function is

```
feval(fun,value)
```

For example, the statement `x = feval('sin',pi/4)` produces the result

```
» x = feval('sin',pi/4)
x =
    0.7071
```

Some of the more common MATLAB function functions are listed in Table 5.1. Type `help fun_name` to learn how to use each of these functions.

Table 5.1 Common MATLAB Function Functions

Function Name	Description
`fminbnd`	Minimize a function of one variable.
`Fzero`	Find a zero of a function of one variable.
`Quad`	Numerically integrate a function.
`Ezplot`	Easy to use function plotter.
`fplot`	Plot a function by name.

Example 5.6—Creating a Function Function

Create a function function that will plot any MATLAB function of a single variable between specified starting and ending values.

SOLUTION This function have two input arguments, the first one containing the name of the function to plot and the second one containing a two-element vector with the range of values to plot.

1. **State the problem.**
   Create a function to plot any MATLAB function of a single variable between two user-specified limits.

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

   1. A character string containing the name of a function.
   2. A two-element vector containing the first and last values to plot.

   The output from this function is a plot of the function specified in the first input argument.

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

```
Check for a legal number of arguments
Check that the second argument has two elements
Calculate the value of the function between the
   start and stop points
Plot and label the function
```

   The detailed pseudocode for the evaluation and plotting steps is:

```
n_steps <- 100
step_size <- (xlim(2) - xlim(1)) / n_steps
x <- xlim(1):step_size:xlim(2)
y <- feval(fun,x)
plot(x,y)
title(['\bfPlot of function ' fun '(x)'])
xlabel('\bfx')
ylabel(['\bf' fun '(x)'])
```

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

```
function quickplot(fun,xlim)
%QUICKPLOT Generate quick plot of a function
% Function QUICKPLOT generates a quick plot
% of a function contained in a external M-file,
% between user-specified x limits.
% Define variables:
%   fun       -- Function to plot
%   msg       -- Error message
%   n_steps   -- Number of steps to plot
%   step_size -- Step size
%   x         -- X-values to plot
%   y         -- Y-values to plot
%   xlim      -- Plot x limits
%
% Record of revisions:
%     Date          Programmer        Description of change
%     ====          ==========        =====================
%   01/13/04       S. J. Chapman      Original code

% Check for a legal number of input arguments.
msg = nargchk(2,2,nargin);
error(msg);

% Check the second argument to see if it has two
% elements. Note that this double test allows the
% argument to be either a row or a column vector.
if ( size(xlim,1) == 1 & size(xlim,2) == 2 ) | ...
   ( size(xlim,1) == 2 & size(xlim,2) == 1 )

   % Ok--continue processing.
   n_steps = 100;
   step_size = (xlim(2) - xlim(1)) / n_steps;
   x = xlim(1):step_size:xlim(2);
   y = feval(fun,x);
   plot(x,y);
   title(['\bfPlot of function ' fun '(x)']);
   xlabel('\bfx');
   ylabel(['\bf' fun '(x)']);

else
   % Else wrong number of elements in xlim.
   error('Incorrect number of elements in xlim.');
end

end % function quickplot
```

5. **Test the program.**

   To test this function, we must call it with correct and incorrect input arguments, verifying that it handles both correct inputs and errors properly. The results are shown below:

```
» quickplot('sin')
??? Error using ==> quickplot
Not enough input arguments.
```

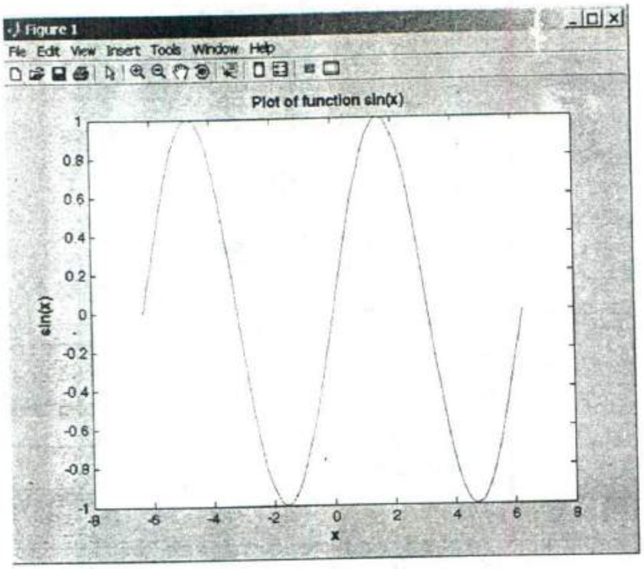


Figure 5.6 Plot of sin *x* versus *x* generated by function `quickplot`.

```
» quickplot('sin',[-2*pi 2*pi],3)
??? Error using ==> quickplot
Too many input arguments.

» quickplot('sin',-2*pi)
??? Error using ==> quickplot
Incorrect number of elements in xlim.

» quickplot('sin',[-2*pi 2*pi])
```

The last call was correct, and it produced the plot shown in Figure 5.6.

5.7 Subfunctions, Private Functions, and Nested Functions

MATLAB includes several special types of functions that behave differently than the ordinary functions we have used so far. Ordinary functions can be called by any other function, as long as they are in the same directory or in any directory on the MATLAB path.

The **scope** of a function is defined as the locations within MATLAB from which the function can be accessed. The scope of an ordinary MATLAB function is the current working directory. If the function lies in a directory on the MATLAB path, then the scope extends to all MATLAB functions in a program, because they all check the path when trying to find a function with a given name.

In contrast, the scope of the other function types that we will discuss in the rest of this chapter is more limited in one way or another.

Subfunctions

It is possible to place more than one function in a single file. If more than one function is present in a file, the top function is a normal or **primary function**, while the ones below it are **subfunctions**. The primary function should have the same name as the file it appears in. Subfunctions look just like ordinary functions, but they are only accessible to the other functions within the same file. In other words, the scope of a subfunction is the other functions within the same file (see Figure 5.7).

Subfunctions are often used to implement "utility" calculations for a main function. For example, the file `mystats.m` shown below contains a primary function `mystats` and two subfunctions `mean` and `median`. Function `mystats` is a normal MATLAB function, so it can be called by any other MATLAB function in the same directory. If this file is in a directory included in the MATLAB search path, it can be called by any other MATLAB function, even if the other function is

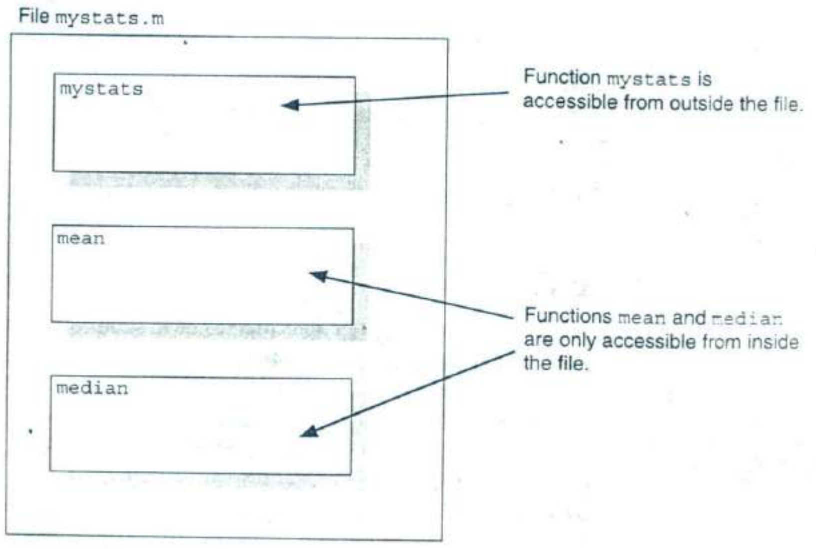

File mystats.m
mystats
mean
median
Function mystats is accessible from outside the file.
Functions mean and median are only accessible from inside the file.

Figure 5.7 The first function in a file is called the primary function. It should have the same name as the file it appears in, and it is accessible from outside the file. The remaining functions in the file are subfunctions; they are accessible only from within the file.

not in the same directory. By contrast, the scope of functions mean and median is restricted to other functions within the same file. Function mystats can call them and they can call each other, but a function outside of the file cannot. They are "utility" functions that perform a part the job of the main function mystats.

```
function [avg, med] = mystats(u)
% MYSTATS Find mean and median with internal functions.
% Function MYSTATS calculates the average and median
% of a data set using subfunctions.

n = length(u);
avg = mean(u,n);
med = median(u,n);

end % function mystats

function a = mean(v,n)
% Subfunction to calculate average.
a = sum(v)/n;

end % function mean

function m = median(v,n)
% Subfunction to calculate median.
w = sort(v);
if rem(n,2) == 1
   m = w((n + 1)/2);
else
   m = (w(n/2)+ w(n/2 + 1))/2;
end

end % function median
```

Private Functions

Private functions are functions that reside in subdirectories with the special name private. They are only visible to other functions in the private directory or to functions in the parent directory. In other words, the scope of these functions is restricted to the private directory and to the parent directory that contains it.

For example, assume the directory testing is on the MATLAB search path. A subdirectory of testing called private can contain functions that only the functions in testing can call. Because private functions are invisible outside of the parent directory, they can use the same names as functions in other directories. This is useful if you want to create your own version of a particular function while retaining the original in another directory. Because MATLAB looks for private functions before standard M-file functions, it will find a private function named test.m before a non-private function named test.m.

You can create your own private directories by simply creating a subdirectory called `private` under the directory containing your functions. Do not place these private directories on your search path.

When a function is called from within an M-file, MATLAB first checks the file to see whether the function is a subfunction defined in the same file. If not, it checks for a private function with that name. If it is not a private function, MATLAB checks the current directory for the function name. If it is not in the current directory, MATLAB checks the standard search path for the function.

If you have special-purpose MATLAB functions that should be used only by other functions and should never be called directly by the user, consider hiding them as subfunctions or private functions. Hiding the functions will prevent their accidental use and will also prevent conflicts with other public functions of the same name.

Nested Functions

Nested functions are functions that are defined *entirely within the body of another function*, called the **host function**. They are visible only to the host function in which they are embedded and to other nested functions embedded at the same level within the same host function.

A nested function has access to any variables defined with it, *plus any variables defined within the host function* (see Figure 5.8). The only exception occurs if a variable in the nested function has the same name as a variable within the host function. In that case, the variable within the host function is not accessible.

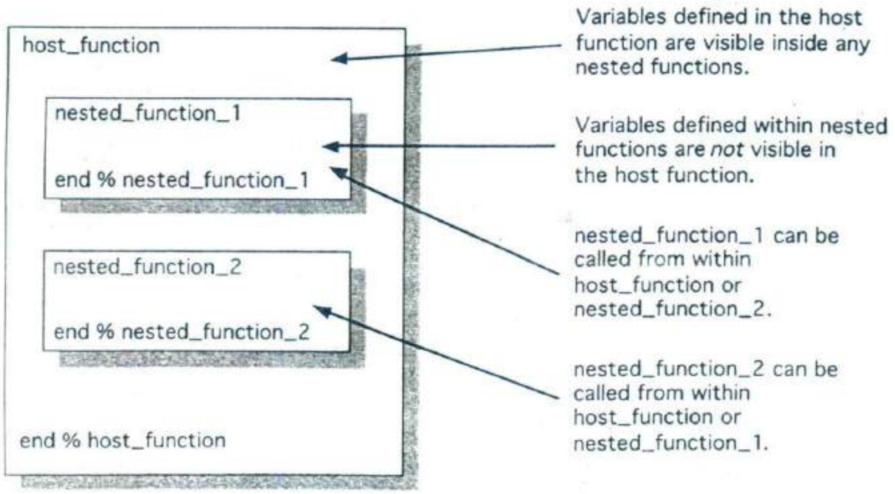

host_function
nested_function_1
end % nested_function_1
nested_function_2
end % nested_function_2
end % host_function
Variables defined in the host function are visible inside any nested functions.
Variables defined within nested functions are *not* visible in the host function.
nested_function_1 can be called from within host_function or nested_function_2.
nested_function_2 can be called from within host_function or nested_function_1.

Figure 5.8 Nested functions are defined within a host function, and they inherit variables defined within the host function.

Note that if a file contains one or more nested functions, then *every function in the file* must be terminated with an end statement. This is the only time when the end statement is required at the end of a function—at all other times it is optional.

Programming Pitfalls

If a file contains one or more nested functions, then *every function in the file* must be terminated with an end statement. It is an error to omit end statements in this case.

The following program illustrates the use of variables in nested functions. It contains a host function test_nested_1 and a nested function fun1. When the program starts, variables a, b, x, and y are initialized as shown in the host function, and their values are displayed. Then the program calls fun1. Since fun1 is nested, it inherits a, b, and x from the host function. Note that it does *not* inherit y, because fun1 defines a local variable with that name. When the values of the variables are displayed at the end of fun1, we see that a has been increased by 1 (due to the assignment statement), and that y is set to 5. When execution returns to the host function, a is still increased by 1, showing that the variable a in the host function and the variable a in the nested function are really the same. On the other hand, y is again 9, because the variable y in the host function is not the same as the variable y in the nested function.

```
function res = test_nested_1

% This is the top level function.
% Define some variables.
a = 1; b = 2; x = 0; y = 9;

% Display variables before call to fun1
fprintf('Before call to fun1:\n');
fprintf('a, b, x, y = %2d %2d %2d %2d\n', a, b, x, y);

% Call nested function fun1
x = fun1(x);

% Display variables after call to fun1
fprintf('\nAfter call to fun1:\n');
fprintf('a, b, x, y = %2d %2d %2d %2d\n', a, b, x, y);

    % Declare a nested function
    function res = fun1(y)

    % Display variables at start of call to fun1
    fprintf('\nAt start of call to fun1:\n');
    fprintf('a, b, x, y = %2d %2d %2d %2d\n', a, b, x, y);
    y = y + 5;
    a = a + 1;
    res = y;

    % Display variables at end of call to fun1
    fprintf('\nAt end of call to fun1:\n');
    fprintf('a, b, x, y = %2d %2d %2d %2d\n', a, b, x, y);

    end % function fun1

end % function test_nested_1
```

When this program is executed, the results are:

```
» test_nested_1
Before call to fun1:
a, b, x, y = 1 2 0 9

At start of call to fun1:
a, b, x, y = 1 2 0 0

At end of call to fun1:
a, b, x, y = 2 2 0 5

After call to fun1:
a, b, x, y = 2 2 5 9
```

Like subfunctions, nested functions can be used to perform special-purpose calculations within a host function.

✱ Good Programming Practice

Use subfunctions, private functions, or nested functions to hide special-purpose calculations that should not be generally accessible to other functions. Hiding the functions will prevent their accidental use, and will also prevent conflicts with other public functions of the same name.

Order of Function Evaluation

In a large program, there could possibly be multiple functions (subfunctions, private functions, nested functions, and public functions) with the same name. When a function with a given name is called, how do we know which copy of the function will be executed?

The answer to this question is that MATLAB locates functions in a specific order as follows:

1. First, MATLAB checks to see whether there is a nested function with the specified name. If so, it is executed.
2. MATLAB checks to see whether there is a subfunction with the specified name. If so, it is executed.
3. MATLAB checks for a private function with the specified name. If so, it is executed.
4. MATLAB checks for a function with the specified name in the current directory. If so, it is executed.
5. MATLAB checks for a function with the specified name on the MATLAB path. MATLAB will stop searching and execute the first function with the right name found on the path.

5.8 Summary

In Chapter 5, 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 normal function (one that is not a subfunction) 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 then number of actual input arguments used in a function call, and function `nargout` reports then 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.

Function functions are MATLAB functions whose input arguments include the names of other functions. The functions whose names are passed to the function function are normally used during that function's execution. Examples are some root-solving and plotting functions.

Subfunctions are additional functions placed within a single file. Subfunctions are accessible only from other functions within the same file. Private functions are functions placed in a special subdirectory called `private`. They are only accessible to functions in the parent directory. Nested functions are functions completed defined within the body of another function (called the host function). Nested functions have access to the variables in the host function as well as to their own local variables. Subfunctions, private functions, and nested functions can be used to restrict access to MATLAB functions.

Summary of Good Programming Practice

The following guidelines should be adhered to when working with MATLAB functions.

1. Break large program tasks into smaller, more understandable functions whenever possible.
2. Always terminate your functions with an `end` statement, and include a comment on the statement indicating which function the `end` statement is associated with.
3. Declare global variables in all capital letters to make them easy to distinguish from local variables.
4. Declare global variables immediately after the initial comments and before the first executable statement each function that uses them.
5. You may use global memory to pass large amounts of data among functions within a program.
6. Use persistent memory to preserve the values of local variables within a function between calls to the function.
7. Use subfunctions, private functions, or nested functions to hide special-purpose calculations that should not be generally accessible to other functions. Hiding the functions will prevent their accidental use, and will also prevent conflicts with other public functions of the same name.

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.
`eval`	Evaluates a character string as though it had been typed in the Command Window.
`ezplot`	Easy-to-use function plotter.

(continued)

Commands and Functions

`feval`	Calculates the value of a function $f(x)$ defined by an M-file at a specific x.
`fmin`	Minimize a function of one variable.
`fplot`	Plot a function by name.
`fzero`	Find a zero of a function of one variable.
`global`	Declares global variables.
`hist`	Calculate and plot a histogram of a data set.
`inputname`	Returns the actual name of the variable that corresponds to a particular argument number.
`nargchk`	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.
`quad`	Numerically integrate a function.
`rand`	Generates random values from a uniform distribution.
`randn`	Generates random values from a normal distribution.
`return`	Stop executing a function and return to caller.
`warning`	Displays a warning message and continues function execution. This function is used if the argument errors are not fatal, and execution can continue.

5.9 Exercises

5.1 What is the difference between a script file and a function?

5.2 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?

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

5.4 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.)

5.5 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.

5.6 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.

5.7 Write a function that uses function `random0` to generate a random value in the range $[-1.0, 1.0)$. Make `random0` a subfunction of your new function.

5.8 Write a function that uses function `random0` to generate a random value in the range `[low, high)`, where `low` and `high` are passed as calling arguments. Make `random0` a private function called by your new function.

5.9 **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.)

5.10 **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

$$P(k, t) = e^{-\lambda t}\frac{(\lambda t)^k}{k!} \quad \text{for } t \geq 0,\ \lambda > 0, \text{ and } k = 0, 1, 2, \ldots \tag{5-10}$$

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 λ. 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 λ is 1.6 per minute for that highway. Plot the Poisson distribution for $t = 1$ and $\lambda = 1.6$.

5.11 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.

5.12 Write a single MATLAB function `hyperbolic` to calculate the hyperbolic sine, cosine, and tangent functions as defined in the previous problem. The function should have two arguments. The first argument will be a string containing the function names `'sinh'`, `'cosh'`, or `'tanh'`, and the second argument will be the value of x at which to evaluate the function. The file should also contain three subfunctions—`sinh1`, `cosh1`, and `tanh1`—to perform the actual calculations, and the primary function should call the proper subfunction depending on the value in the string. [**Note:** Be sure to handle the case of an incorrect number of arguments, and also the case of an invalid string. In either case, the function should generate an error.]

5.13 **Cross Product** Write a function to calculate the cross product of two vectors $\mathbf{V}_1$ and $\mathbf{V}_2$:

$$\mathbf{V}_1 \times \mathbf{V}_2 = (V_{y1}V_{z2} - V_{y2}V_{z1})\,\mathbf{i} + (V_{z1}V_{x2} - V_{z2}V_{x1})\,\mathbf{j} + (V_{x1}V_{y2} - V_{x2}V_{y1})\,\mathbf{k}$$

where $\mathbf{V}_1 = V_{x1}\,\mathbf{i} + V_{y1}\,\mathbf{j} + V_{z1}\,\mathbf{k}$ and $\mathbf{V}_2 = V_{x2}\,\mathbf{i} + V_{y2}\,\mathbf{j} + V_{z2}\,\mathbf{k}$. Note that this function will return a real array as its result. Use the function to calculate the cross product of the two vectors $\mathbf{V}_1 = [-2, 4, 0.5]$ and $\mathbf{V}_2 = [0.5, 3, 2]$.

5.14 **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];
```

5.15 Use the Help Browser to look up information about the standard MATLAB function `sortrows`, and compare the performance of `sortrows` with the sort-with-carry function created in the previous exercise. To do this, create two copies of a 1000×2 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`?

5.16 Figure 5.9 shows two ships steaming on the ocean. Ship 1 is at position (x_1, y_1) and steaming on heading θ_1. Ship 2 is at position (x_2, y_2) and steaming on heading θ_2. Suppose that Ship 1 makes radar contact with an object at range r_1 and bearing ϕ_1. Write a MATLAB function that will calculate the range r_2 and bearing ϕ_2 at which Ship 2 should see the object.

5.17 **Minima and Maxima of a Function** Write a function that attempts to locate the maximum and minimum values of an arbitrary function $f(x)$ over a certain range. The function being evaluated should be passed to the function as a calling argument. The function should have the following input arguments:

`first_value`—The first value of x to search
`last_value`—The last value of x to search
`num_steps`—The number of steps to include in the search
`func`—The name of the function to search

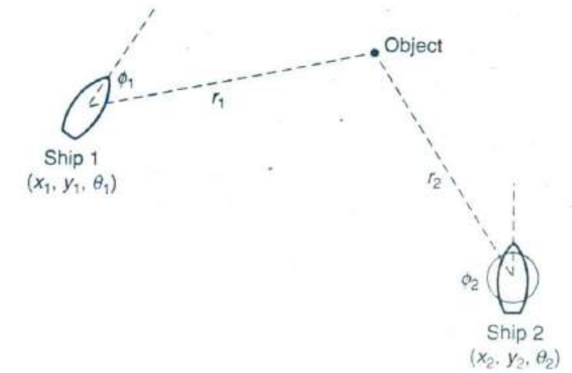


Figure 5.9 Two ships at positions (x_1, y_1) and (x_2, y_2) respectively. Ship 1 is traveling at heading θ_1, and Ship 2 is traveling at heading θ_2.

The function should have the following output arguments:

`xmin`—The value of x at which the minimum was found
`min_value`—The minimum value of $f(x)$ found
`xmax`—The value of x at which the maximum was found
`max_value`—The maximum value $f(x)$ found

Be sure to check that there are a valid number of input arguments, and that the MATLAB `help` and `lookfor` commands are properly supported.

5.18 Write a test program for the function generated in the previous exercise. The test program should pass to the function function the user-defined function $f(x) = x^3 - 5x^2 + 5x + 2$, and search for the minimum and maximum in 200 steps over the range $-1 \leq x \leq 3$. It should print out the resulting minimum and maximum values.

5.19 **Derivative of a Function** The *derivative* of a continuous function $f(x)$ is defined by the equation

$$\frac{d}{dx} f(x) = \lim_{\Delta x \to 0} \frac{f(x + \Delta x) - f(x)}{\Delta x} \tag{5-11}$$

In a sampled function, this definition becomes

$$f'(x_i) = \frac{f(x_{i+1}) - f(x_i)}{\Delta x} \tag{5-12}$$

where $\Delta x = x_{i+1} - x_i$. Assume that a vector `vect` contains `nsamp` samples of a function taken at a spacing of `dx` per sample. Write a function that will calculate the derivative of this vector from Equation (5-12). The function should check to make sure that `dx` is greater than zero to prevent divide-by-zero errors in the function.

To check your function, you should generate a data set whose derivative is known, and compare the result of the function with the known correct answer. A good choice for a test function is sin x. From elementary calculus, we know that $\frac{d}{dx}(\sin x) = \cos x$. Generate an input vector containing 100 values of the function sin x starting at $x = 0$ and using a step size Δx of 0.05. Take the derivative of the vector with your function, and then compare the resulting answers to the known correct answer. How close did your function come to calculating the correct value for the derivative?

5.20 **Derivative in the Presence of Noise** We will now explore the effects of input noise on the quality of a numerical derivative. First, generate an input vector containing 100 values of the function sin x starting at $x = 0$ and using a step size Δx of 0.05, just as you did in the previous problem. Next, use function `random0` to generate a small amount of random noise with a maximum amplitude of ± 0.02 and add that random noise to the samples in your input vector. Note that the peak amplitude of the noise is only 2% of the peak amplitude of your signal, since the maximum value of sin x is 1. Now take the derivative of the function using the derivative function that you developed in Problem 5.19. How close to the theoretical value of the derivative did you come?

5.21 **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 [illegible] y. (The equations describing the slope and intercept of the least-squares line given in Example 4.6 in the previous chapter.) 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	−5.62	15	1.01	5.75
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	5.13	7.31
9	0.28	−1.16	19	4.43	9.05
10	1.08	0.52	20	4.12	10.95

5.22 **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 4.6, and the equation for the correlation coefficient is

$$r = \frac{n\left(\sum xy\right) - \left(\sum x\right)\left(\sum y\right)}{\sqrt{\left[\left(n\sum x^2\right) - \left(\sum x\right)^2\right]\left[\left(n\sum y^2\right) - \left(\sum y\right)^2\right]}} \tag{5-13}$$

where

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

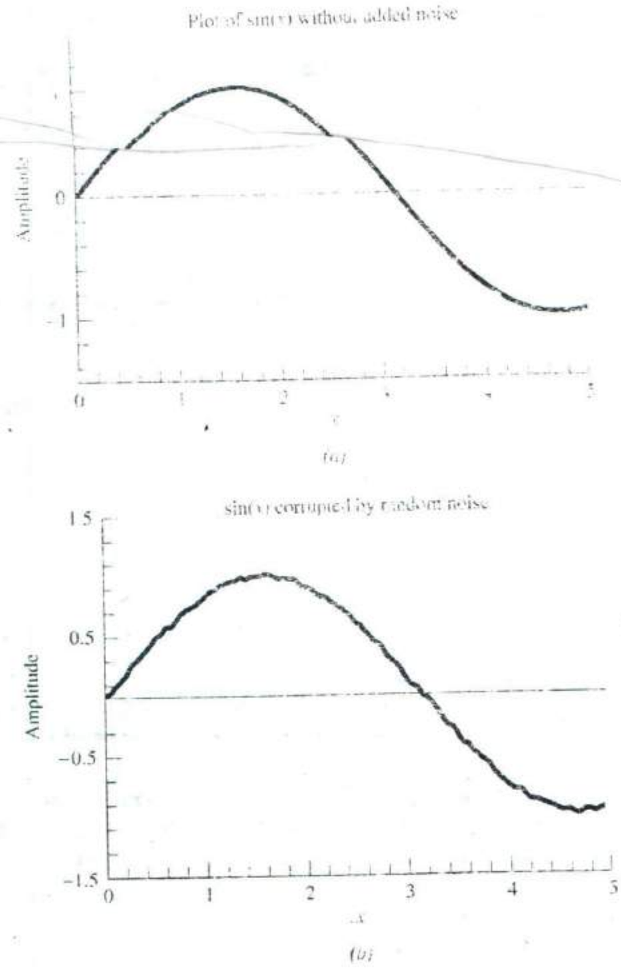


Figure 5.10 *(a)* A plot of sin x as a function of x with no noise added to the data. *(b)* A plot of sin x as a function of x with a 2% peak amplitude uniform random noise added to the data.

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

5.23 **Recursion** A function is said to be *recursive* if the function calls itself. MATLAB functions are designed to allow recursive operation. To test this feature, write a MATLAB function to evaluate the factorial function, which is defined as follows:

$$N! = \begin{cases} N(N-1)! & N \geq 1 \\ 1 & N = 0 \end{cases} \qquad (5\text{-}14)$$

where N is a positive integer. The function should check to make sure that there is a single argument N, and that N is a nonnegative integer. If it is not, generate an error using the `error` function. If the input argument is a nonegative integer, the function should evaluate $N!$ using Equation (5-14).

5.24 **The Birthday Problem** The Birthday Problem is as follows: if there are a group of n people in a room, what is the probability that two or more of them have the same birthday? It is possible to determine the answer to this question by simulation. Write a function that calculates the probability that two or more of n people will have the same birthday, where n is a calling argument. (*Hint*: To do this, the function should create an array of size n and generate n birthdays in the range 1 to 365 randomly. It should then check to see if any of the n birthdays are identical. The function should perform this experiment at least 5000 times and calculate the fraction of those times in which two or more people had the same birthday.) Write a test program that calculates and prints out the probability that 2 or more of n people will have the same birthday for $n = 2, 3, \ldots, 40$.

5.25 Use function `random0` to generate a set of three arrays of random numbers. The three arrays should be 100, 1000, and 2000 elements long. Then, use functions `tic` and `toc` to determine the time that it takes function `ssort` to sort each array. How does the elapsed time to sort increase as a function of the number of elements being sorted? (*Hint*: On a fast computer, you will need to sort each array many times and calculate the average sorting time in order to overcome the quantization error of the system clock.)

5.26 **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 5.11. 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

$$p(x) = \frac{1}{\sqrt{2\pi}} e^{-x^2/2} \tag{5-15}$$

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 equations below will be a normally distributed random variable.

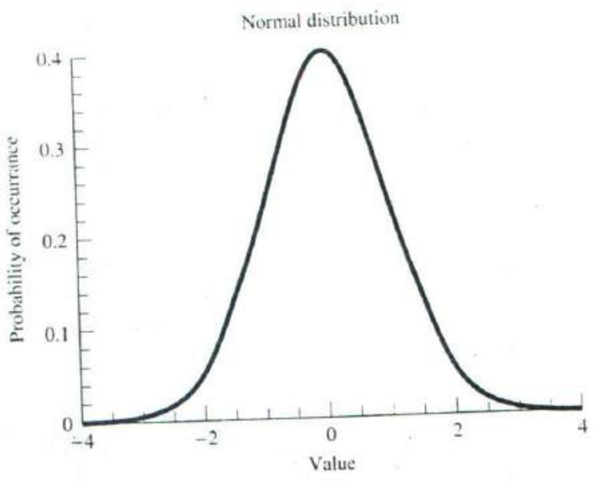


Figure 5.11 A Normal probability distribution.

$$y_1 = \sqrt{\frac{-2\ln r}{r}}\,x_1 \tag{5-16}$$

$$y_2 = \sqrt{\frac{-2\ln r}{r}}\,x_2 \tag{5-17}$$

where

$$r = x_1^2 + x_2^2 \tag{5-18}$$

Write a function that returns a normally distributed random value each time that 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?

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

$$F = \frac{Gm_1m_2}{r^2} \tag{5-19}$$

where G is the gravitation constant (6.672×10^{-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 you function by determining the force on an 800 kg satellite in orbit 38,000 km above the Earth. (The mass of the Earth is 5.98×10^{24} kg.)

5.28 **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:

$$r = \sqrt{n_1^2 + n_2^2} \tag{5-20}$$

(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.

5.29 Constant False Alarm Rate (CFAR) A simplified radar receiver chain is shown in Figure 5.12*a*. When a signal is received in this receiver, it contains both the desired information (returns from targets) and thermal noise. After the detection step in the receiver, we would like to be able to pick out received target returns from the thermal noise background. We can do this be setting a threshold level, and then declaring that we see a target whenever the signal crosses that threshold. Unfortunately, it is occasionally possible for the receiver noise to cross the detection threshold even if no target is present. If that happens, we will declare the noise spike to be a target, creating a *false alarm*. The detection threshold needs to be set as low as possible so that we can detect weak targets, but it must not be set too low, or we get many false alarms.

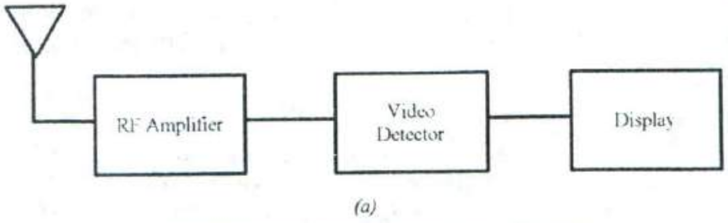
After video detection, the thermal noise in the receiver has a Rayleigh distribution. Figure 5.12*b* shows 100 samples of a Rayleigh-distributed noise with a mean amplitude of 10 volts. Note that there would be one false alarm even if the detection threshold were as high as 26! The probability distribution of these noise samples is shown in Figure 5.12*c*.

Detection thresholds are usually calculated as a multiple of the mean noise level, so that if the noise level changes, the detection threshold will change with it to keep false alarms under control. This is known as *constant false alarm rate* (CFAR) detection. A detection threshold is typical quoted in decibels. The relationship between the threshold in dB and the threshold in volts is

$$\text{Threshold (volts)} = \text{Mean Noise Level (volts)} \times 10^{\frac{\text{dB}}{20}} \tag{5-21}$$

or

$$\text{dB} = 20 \log_{10}\left(\frac{\text{Threshold (volts)}}{\text{Mean Noise Level (volts)}}\right) \tag{5-22}$$



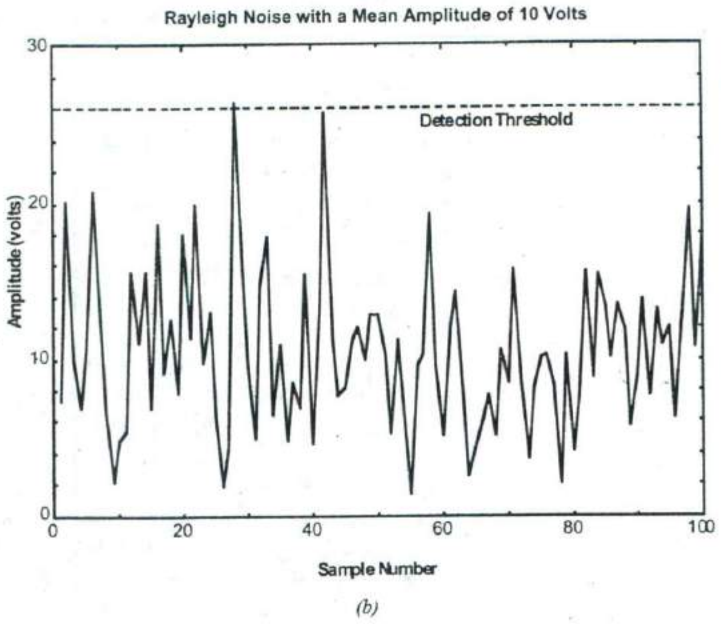


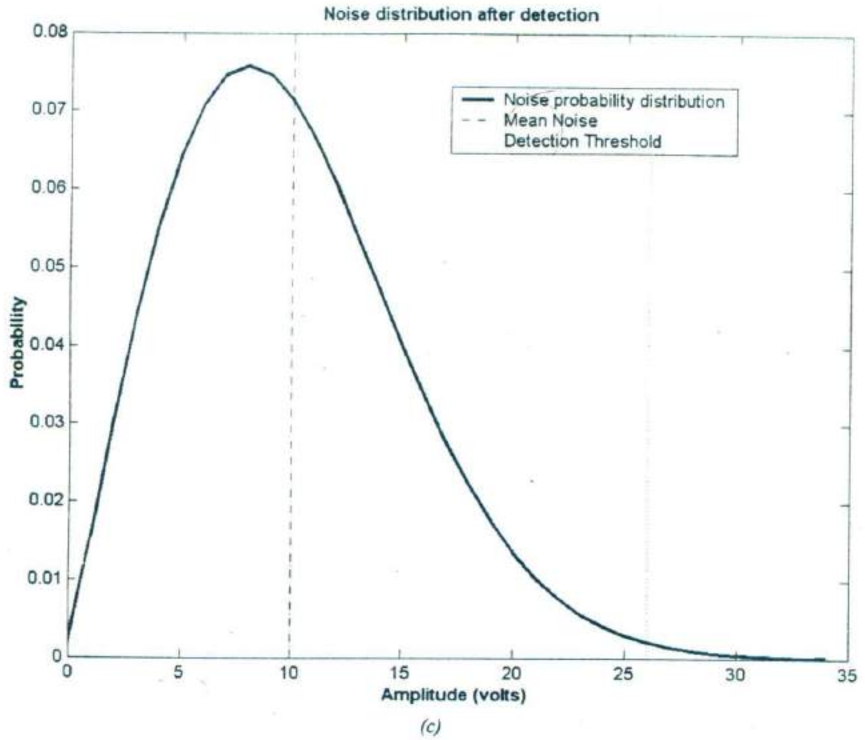
Figure 5.12 *(a)* A typical radar receiver. *(b)* Thermal noise with a mean of 10 volts output from the detector. The noise sometimes crosses the detection threshold. *(c)* Probability distribution of the noise out of the detector.

The false alarm rate for a given detection threshold is calculated as:

$$P_{fa} = \frac{\text{Number of False Alarms}}{\text{Total Number of Samples}} \tag{5-23}$$

Write a program that generates 1,000,000 random noise samples with a mean amplitude of 10 volts and a Rayleigh noise distribution. Determine the false alarm rates when the detection threshold is set to 5, 6, 7, 8, 9, 10, 11, 12, and 13 dB above the mean noise level. At what level should the threshold be set to achieve a false alarm rate of 10^{-4}?

5.30 **Probability of Detection (P_d) versus Probability of False Alarm (P_{fa})** The signal strength returned by a radar target usually fluctuates over time. The target will be detected if its signal strength exceeds the detection



(c)

Figure 5.12 *(continued)*

threshold for any given look. The probability that the target will be detected can be calculated as:

$$P_d = \frac{\text{Number of Target Detections}}{\text{Total Number of Looks}} \tag{5-24}$$

Suppose that a specific radar looks repeatedly in a given direction. On each look, the range between 10 km and 20 km is divided into 100 independent range samples (called *range gates*). One of these range gates contains a target whose amplitude has a normal distribution with a mean amplitude of 7 volts and a standard deviation of 1 volt. All 100 of the range gates contain system noise with a mean amplitude of 2 volts and a Rayleigh distribution. Determine both the probability of target detection P_d and the probability of a false alarm P_{fa} on any given look for detection thresholds of 8.0, 8.5, 9.0, 9.5, 10.0, 10.5, 11.0, 11.5, and 12.0 dB. What threshold would you use for detection in this radar? (*Hint*: Perform the experiment many times for each threshold, and average the results to determine valid probabilities.)

CHAPTER 6

Additional Data Types and Plot Types

In earlier chapters, we were introduced to three fundamental MATLAB data types: `double`, `logical`, and `char`. In this chapter, we will learn additional details about these data types, and then we will study some additional MATLAB data types.

First, we will learn how to create, manipulate, and plot complex values in the `double` data type. Then, we will learn more about using the `char` data type, and how to extend MATLAB arrays of any type to more than two dimensions.

Finally, we will learn about some additional data types. The MATLAB data types are shown in Figure 6.1. We will learn about the `single` and integer data types in this chapter, and discuss the remaining ones on the figure later in this book.

The chapter concludes with a discussion of additional types of plots available in MATLAB.

6.1 Complex Data

Complex numbers are numbers with both a real and an imaginary component. Complex numbers occur in many problems in science and engineering. For example, complex numbers are used in electrical engineering to represent alternating current voltages, currents, and impedances. The differential equations that describe the behavior of most electrical and mechanical systems also give rise to complex numbers. Because they are so ubiquitous, it is impossible to work as an engineer without a good understanding of the use and manipulation of complex numbers.

A complex number has the general form

$$c = a + bi \tag{6-1}$$

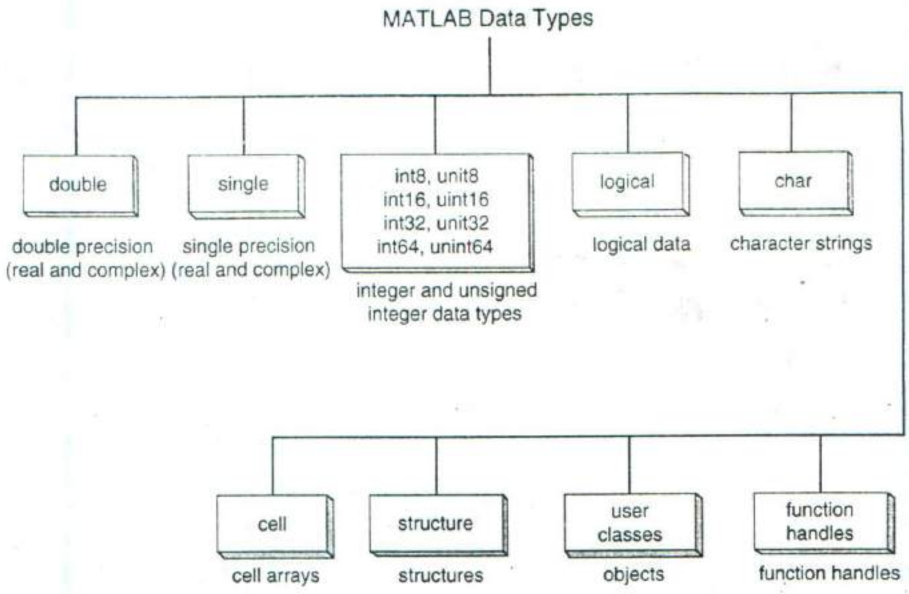


Figure 6.1 MATLAB data types.

where c is a complex number, a and b are both real numbers, and i is $\sqrt{-1}$. The number a is called the *real part* and b is called the *imaginary part* of the complex number c. Since a complex number has two components, it can be plotted as a point on a plane (see Figure 6.2). The horizontal axis of the plane is the real axis, and the vertical axis of the plane is the imaginary axis, so that any complex number $a + bi$ can be represented as a single point a units along the real axis and b units along the imaginary axis. A complex number represented this way is said to be in *rectangular coordinates*, since the real and imaginary axes define the sides of a rectangle.

A complex number can also be represented as a vector of length z and angle θ pointing from the origin of the plane to the point P (see Figure 6.3). A complex number represented this way is said to be in *polar coordinates*.

$$c = a + bi = z\angle\theta$$

The relationships among the rectangular and polar coordinate terms a, b, z, and θ are:

$$a = z\cos\theta \tag{6-2}$$

$$b = z\sin\theta \tag{6-3}$$

$$z = \sqrt{a^2 + b^2} \tag{6-4}$$

$$\theta = \tan^{-1}\frac{b}{a} \tag{6-5}$$

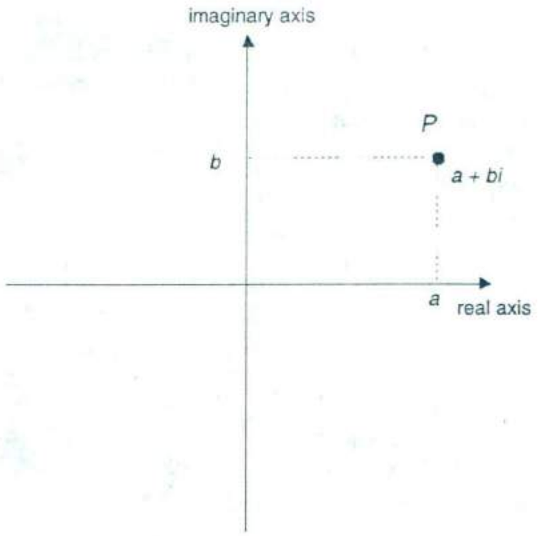


Figure 6.2 Representing a complex number in Rectangular Coordinates.

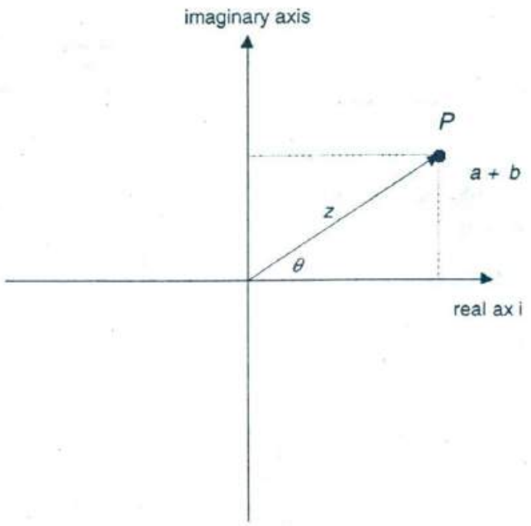


Figure 6.3 Representing a complex number in Polar Coordinates.

MATLAB uses rectangular coordinates to represent complex numbers. Each complex number consists of a pair of real numbers (a, b). The first number (a) is the real part of the complex number, and the second number (b) is the imaginary part of the complex number.

If complex numbers c_1 and c_2 are defined as $c_1 = a_1 + b_1 i$ and $c_2 = a_2 + b_2 i$, then the addition, subtraction, multiplication, and division of c_1 and c_2 are defined as:

$$c_1 + c_2 = (a_1 + a_2) + (b_1 + b_2)i \tag{6-6}$$

$$c_1 - c_2 = (a_1 - a_2) + (b_1 - b_2)i \tag{6-7}$$

$$c_1 \times c_2 = (a_1 a_2 - b_1 b_2) + (a_1 b_2 + b_1 a_2)i \tag{6-8}$$

$$\frac{c_1}{c_2} = \frac{a_1 a_2 + b_1 b_2}{a_2^2 + b_2^2} + \frac{b_1 a_2 - a_1 b_2}{a_2^2 + b_2^2}\, i \tag{6-9}$$

When two complex numbers appear in a binary operation, MATLAB performs the required additions, subtractions, multiplications, or divisions between the two complex numbers using versions of the preceding formulas.

Complex Variables

A complex variable is created automatically when a complex value is assigned to a variable name. This easiest way to create a complex value is to use the intrinsic values `i` or `j`, both of which are predefined to be $\sqrt{-1}$. For example, the following statement stores the complex value $4 + i3$ into variable `c1`.

```
» c1 = 4 + i*3
c1 =
     4.0000 + 3.0000i
```

Alternately, the imaginary part can be specified by simply appending an `i` or `j` to the end of a number:

```
» c1 = 4 + 3i
c1 =
     4.0000 + 3.0000i
```

The function `isreal` can be used to determine whether a given array is real or complex. If any element of an array has an imaginary component, then the array is complex and `isreal(array)` returns a 0.

Using Complex Numbers with Relational Operators

It is possible to compare two complex numbers with the == relational operator to see if they are equal to each other, and to compare them with the ~= operator to see if they are not equal to each other. Both of these operators produce the expected results. For example, if $c_1 = 4 + i3$ and $c_2 = 4 - i3$, then the relational operation $c_1 == c_2$ produces a 0 and the relational operation $c_1 \sim= c_2$ produces a 1.

However, *comparisons with the* $>$, $<$, $>=$, *or* $<=$ *operators do not produce the expected results.* When complex numbers are compared with these relational operators, only the *real parts* of the numbers are compared. For example, if $c_1 = 4 + i3$ and $c_2 = 3 + i8$, then the relational operation $c_1 > c_2$ produces a true (1) even though the magnitude of c_1 is really smaller than the magnitude of c_2.

If you ever need to compare two complex numbers with these operators, you will probably be more interested in the total magnitude of the number than in the magnitude of only its real part. The magnitude of a complex number can be calculated with the `abs` intrinsic function (see below), or directly from Equation (6-4).

$$|c| = \sqrt{a^2 + b^2} \tag{6-4}$$

If we compare the *magnitudes* of c_1 and c_2 above, the results are more reasonable: `abs`(c_1) $>$ `abs`(c_2) produces a 0, since the magnitude of c_2 is greater than the magnitude of c_1.

Programming Pitfalls

Be careful when using the relational operators with complex numbers. The relational operators $>$, $>=$, $<$, and $<=$ compare only the *real parts* of complex numbers, not their magnitudes. If you need these relational operators with complex number, it will probably be more sensible to compare the total magnitudes rather than only the real components.

Complex Functions

MATLAB includes many functions that support complex calculations. These functions fall into three general categories:

1. **Type conversion functions.** These functions convert data from the complex data type to the real (`double`) data type. Function `real` converts the *real part* of a complex number into the double data type and throws away the imaginary part of the complex number. Function `imag` converts the *imaginary part* of a complex number into a real number.
2. **Absolute value and angle functions.** These functions convert a complex number to its polar representation. Function `abs(c)` calculates the absolute value of a complex number using the equation

   $$abs(c) = \sqrt{a^2 + b^2}$$

   where $c = a + bi$. Function `angle(c)` calculates the angle of a complex number using the equation

   ```
   angle(c) = atan2(imag(c),real(c))
   ```

   producing an answer in the range $-\pi \leq \theta \leq \pi$.

Table 6.1 Some Functions that Support Complex Numbers

Function	Description
`conj(c)`	Computes the complex conjugate of a number `c`. If $c = a + bi$, then `conj(c)` $= a - bi$
`real(c)`	Returns the real portion of the complex number `c`.
`imag(c)`	Returns the imaginary portion of the complex number `c`.
`isreal(c)`	Returns true (1) if no element of array `c` has an imaginary component. Therefore, `~isreal(c)` returns true (1) if an array is complex.
`abs(c)`	Returns the magnitude of the complex number `c`.
`angle(c)`	Returns the angle of the complex number `c`, computed from the expression `atan2(imag(c), real(c))`.

3. **Mathematical functions.** Most elementary mathematical functions are defined for complex values. These functions include exponential functions, logarithms, trigonometric functions, and square roots. The functions `sin`, `cos`, `log`, `sqrt`, etc. will work as well with complex data as they will with real data.

Some of the intrinsic functions that support complex numbers are listed in Table 6.1.

Example 6.1—The Quadratic Equation (Revisited)

The availability of complex numbers often simplifies the calculations required to solve problems. For example, when we solved the quadratic equation in Example 3.2, it was necessary to take three separate branches through the program depending on the sign of the discriminant. With complex numbers available, the square root of a negative number presents no difficulties, so we can greatly simplify these calculations.

Write a general program to solve for the roots of a quadratic equation, regardless of type. Use complex variables so that no branches will be required based on the value of the discriminant.

SOLUTION

1. **State the problem.**
   Write a program that will solve for the roots of a quadratic equation, whether they are distinct real roots, repeated real roots, or complex roots, without requiring tests on the value of the discriminant.

2. **Define the inputs and outputs.**
   The inputs required by this program are the coefficients a, b, and c of the quadratic equation

$$ax^2 + bx + c = 0$$

   The output from the program will be the roots of the quadratic equation, whether they are real, repeated, or complex.

3. **Describe the algorithm.**
   This task can be broken down into three major sections, whose functions are input, processing, and output:

```
Read the input data
Calculate the roots
Write out the roots
```

   We will now break each of the above major sections into smaller, more detailed pieces. In this algorithm, the value of the discriminant is unimportant in determining how to proceed. The resulting pseudocode is:

```
Prompt the user for the coefficients a, b, and c.
Read a, b, and c
discriminant <- b^2 - 4 * a * c
x1 <- ( -b + sqrt(discriminant) ) / ( 2 * a )
x2 <- ( -b - sqrt(discriminant) ) / ( 2 * a )
Print 'The roots of this equation are: '
Print 'x1 = ', real(x1), ' +i ', imag(x1)
Print 'x2 = ', real(x2), ' +i ', imag(x2)
```

4. **Turn the algorithm into MATLAB statements.**
   The final MATLAB code is shown below.

```
% Script file: calc_roots2.m
%.
% Purpose:
%   This program solves for the roots of a quadratic equation
%   of the form a*x**2 + b*x + c = 0. It calculates the answers
%   regardless of the type of roots that the equation possesses.
%
% Record of revisions:
%     Date           Programmer          Description of change
%     ====           ==========          =====================
%   01/15/04       S. J. Chapman         Original code
%
% Define variables:
%   a               -- Coefficient of x^2 term of equation
%   b               -- Coefficient of x term of equation
%   c               -- Constant term of equation
%   discriminant -- Discriminant of the equation
%   x1           -- First solution of equation
%   x2           -- Second solution of equation

% Prompt the user for the coefficients of the equation
disp ('This program solves for the roots of a quadratic ');
disp ('equation of the form A*X^2 + B*X + C = 0. ');
a = input ('Enter the coefficient A: ');
b = input ('Enter the coefficient B: ');
c = input ('Enter the coefficient C: ');

% Calculate discriminant
discriminant = b^2 - 4 * a * c;

% Solve for the roots
x1 = ( -b + sqrt(discriminant) ) / ( 2 * a );
x2 = ( -b - sqrt(discriminant) ) / ( 2 * a );

% Display results
disp ('The roots of this equation are:');
fprintf ('x1 = (%f) +i (%f)/ n', real(x1), imag(x1));
fprintf ('x2 = (%f) +i (%f) / n', real(x2), imag(x2));
```

5. **Test the program.**
   Next, we must test the program using real input data. We will test cases in which the discriminant is greater than, less than, and equal to 0 to be certain that the program is working properly under all circumstances. From Equation (3-1), it is possible to verify the solutions to the equations given below:

$x^2 + 5x + 6 = 0$	$x = -2$, and $x = -3$
$x^2 + 4x + 4 = 0$	$x = -2$
$x^2 + 2x + 5 = 0$	$x = -1 \pm 2i$

   When the above coefficients are fed into the program, the results are

```
» calc_roots2
This program solves for the roots of a quadratic
equation of the form A*X^2 + B*X + C = 0.
Enter the coefficient A: 1
Enter the coefficient B: 5
Enter the coefficient C: 6
The roots of this equation are:
x1 = (-2.000000) +i (0.000000)
x2 = (-3.000000) +i (0.000000)
» calc_roots2
This program solves for the roots of a quadratic
equation of the form A*X^2 + B*X + C = 0.
Enter the coefficient A: 1
Enter the coefficient B: 4
Enter the coefficient C: 4
The roots of this equation are:
x1 = (-2.000000) +i (0.000000)
x2 = (-2.000000) +i (0.000000)
» calc_roots2
This program solves for the roots of a quadratic
equation of the form A*X^2 + B*X + C = 0.
Enter the coefficient A: 1
Enter the coefficient B: 2
Enter the coefficient C: 5
The roots of this equation are:
x1 = (-1.000000) +i (2.000000)
x2 = (-1.000000) +i (-2.000000)
```

The program gives the correct answers for our test data in all three possible cases. Note how much simpler this program is compared with the quadratic root solver found in Example 3.1. The complex data type has greatly simplified our program.

◄

Plotting Complex Data

Complex data has both real and imaginary components, and plotting complex data with MATLAB is a bit different from plotting real data. For example, consider the function

$$y(t) = e^{-0.2t}(\cos t + i \sin t) \tag{6-10}$$

If this function is plotted with the conventional `plot` command, only the real data will be plotted—the imaginary part will be ignored. The following statements produce the plot shown in Figure 6.4, together with a warning message that the imaginary part of the data is being ignored.

```
t = 0:pi/20:4*pi;
y = exp(-0.2*t).*(cos(t)+i*sin(t));
plot(t,y,'LineWidth',2);
title('\bfPlot of Complex Function vs Time');
xlabel('\bf\itt');
ylabel('\bf\ity(t)');
```

If both the real and imaginary parts of the function are of interest, then the user has several choices. Both parts can be plotted as a function of time on the same axes using the statements shown below (see Figure 6.5).

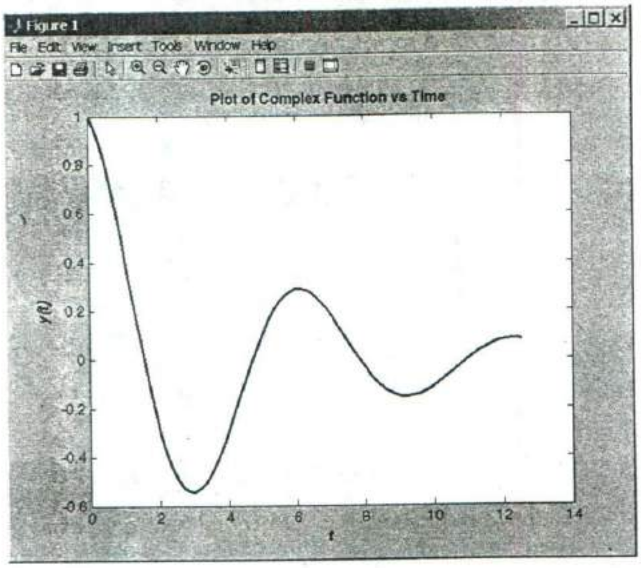


Figure 6.4 Plot of $y(t) = e^{-0.2t}(\cos t + i \sin t)$ using the command plot (t, y).

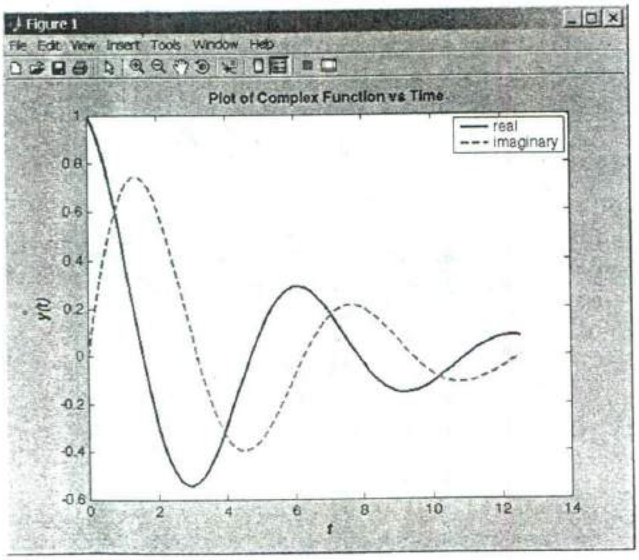


Figure 6.5 Plot of real and imaginary parts of $y(t)$ versus time.

```
t = 0:pi/20:4*pi;
y = exp(-0.2*t).*(cos(t)+i*sin(t));
plot(t,real(y),'b-','LineWidth',2);
hold on;
plot(t,imag(y),'r--','LineWidth',2);
title('\bfPlot of Complex Function vs Time');
xlabel('\bf\itt');
ylabel('\bf\ity(t)');
legend ('real','imaginary');
hold off;
```

Alternatively, the real part of the function can be plotted versus the imaginary part. If a single complex argument is supplied to the `plot` function, it automatically generates a plot of the real part versus the imaginary part. The statements to generate this plot are shown below, and the result is shown in Figure 6.6.

```
t = 0:pi/20:4*pi;
y = exp(-0.2*t).*(cos(t)+i*sin(t));
plot(y,'b-','LineWidth',2);
title('\bfPlot of Complex Function');
xlabel('\bfReal Part');
ylabel('\bfImaginary Part');
```

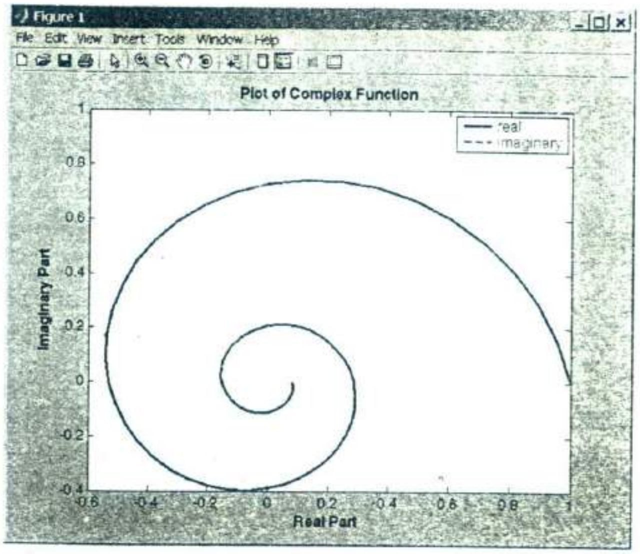


Figure 6.6 Plot of real versus imaginary parts of $y(t)$.

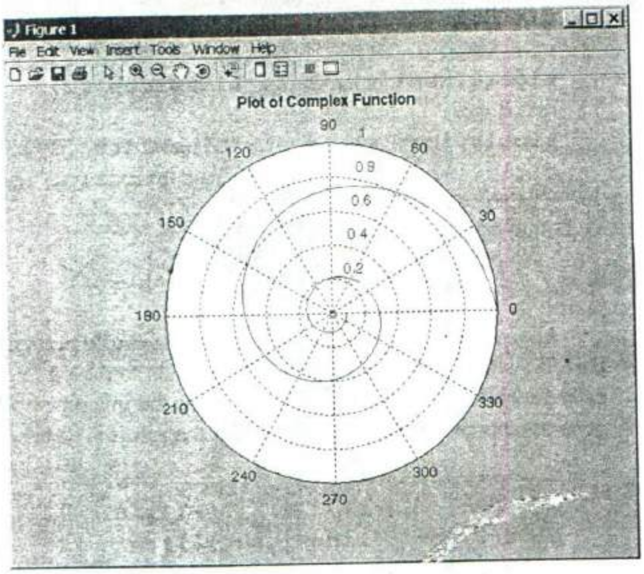


Figure 6.7 Polar plot of magnitude of $y(t)$ versus angle.

Finally, the function can be plotted as a polar plot showing magnitude versus angle. The statements to generate this plot are shown below, and the result is shown in Figure 6.7.

```
t = 0:pi/20:4*ri;
y = exp(-0.2*t).*(cos(t)+i*sin(t));
polar(angle(y),abs(y));
title('\bfPlot of Complex Function');
```

6.2 String Functions

A MATLAB string is an array of type `char`. Each character is stored in two bytes of memory. A character variable is automatically created when a string is assigned to it. For example, the statement

```
str = 'This is a test';
```

creates a 14-element character array. The output of **whos** for this array is

```
» whos str
  Name        Size          Bytes       Class
  str         1x14             28       char array
Grand total is 14 elements using 28 bytes
```

A special function `ischar` can be used to check for character arrays. If a given variable is of type character, then `ischar` returns a true (1) value. If it is not, `ischar` returns a false (0) value.

The following subsections describe MATLAB functions useful for manipulating character strings.

String Conversion Functions

Variables may be converted from the `char` data type to the `double` data type using the `double` function. Thus the statement `double(str)` yields the result:

```
» x = double(str)
x =
 Columns 1 through 12
  84 104 105 115  32 105 115  32  97  32 116 101
 Columns 13 through 14
 115 116
```

Variables can also be converted from the `double` data type to the `char` data type using the `char` function. If x is the 14-element array created above, then the statement `char(x)` yields the result:

```
» z = char(x)
z =
This is a test
```

Creating Two-Dimensional Character Arrays

It is possible to create two-dimensional character arrays, but *each row of such an array must have exactly the same length*. If one of the rows is shorter than the other rows, the character array is invalid and will produce an error. For example, the following statements are illegal because the two rows have different lengths.

```
name = ['Stephen J. Chapman';'Senior Engineer'];
```

The easiest way to produce two-dimensional character arrays is with the `char` function. This function will automatically pad all strings to the length of the largest input string.

```
» name = char('Stephen J. Chapman','Senior Engineer')
name =
Stephen J. Chapman
Senior Engineer
```

Two-dimensional character arrays can also be created with function `strvcat`, which is described below.

Good Programming Practice

Use the `char` function to create two-dimensional character arrays without worrying about padding each row to the same length.

It is possible to remove any extra blanks from a string when it is extracted from an array using the `deblank` function. For example, the following statements remove the second line from array `name`, and compare the results with and without blank trimming.

```
» line2 = name(2,:)
line2 =
Senior Engineer
» line2_trim = deblank(name(2,:))
line2_trim =
Senior Engineer
» size(line2)
ans =
     1    18
» size(line2_trim)
ans =
     1    15
```

Concatenating Strings

Function `strcat` concatenates two or more strings horizontally, ignoring any trailing blanks but preserving blanks within the strings. This function produces the result shown below

```
» result = strcat('String 1 ','String 2')
result =
String 1String 2
```

The result is `'String 1String 2'`. Note that the trailing blanks in the first string were ignored.

Function `strvcat` concatenates two or more strings vertically, automatically padding the strings to make a valid two-dimensional array. This function produces the result shown below

```
» result = strvcat('Long String 1 ','String 2')
result =
Long String 1
String 2
```

Comparing Strings

Strings and substrings can be compared in several ways:

- Two strings, or parts of two strings, can be compared for equality.
- Two individual characters can be compared for equality.
- Strings can be examined to determine whether each character is a letter or whitespace.

Comparing Strings for Equality

You can use four MATLAB functions to compare two strings as a whole for equality. They are:

- `strcmp` determines if two strings are identical.
- `strcmpi` determines if two strings are identical ignoring case. .
- `strncmp` determines if the first `n` characters of two strings are identical
- `strncmpi` determines if the first `n` characters of two strings are identical ignoring case

Function `strcmp` compares two strings, including any leading and trailing blanks, and returns a true (1) if the strings are identical.[1] Otherwise, it returns a false (0). Function `strcmpi` is the same as `strcmp`, except that it ignores the case of letters (that is, it treats `'a'` as equal to `'A'`.)

Function `strncmp` compares the first `n` characters of two strings, including any leading blanks, and returns a true (1) if the characters are identical. Otherwise, it returns a false (0). Function `strncmpi` is the same as `strncmp`, except that it ignores the case of letters.

To understand these functions, consider the two strings:

```
str1 = 'hello';
str2 = 'Hello';
str3 = 'help';
```

Strings `str1` and `str2` are not identical, but they differ only in the case of one letter. Therefore, `strcmp` returns false (0), while `strcmpi` returns true (1).

```
» c = strcmp(str1,str2)
c =
      0
» c = strcmpi(str1,str2)
c =
      1
```

[1]**Caution:** The behavior of this function is different from that of the `strcmp` in C. C programmers can be tripped up by this difference.

Strings `str1` and `str3` are also not identical, and both `strcmp` and `strcmpi` will return a false (0). However, the first three characters of `str1` and `str3` *are* identical, so invoking `strncmp` with any value up to 3 returns a true (`1`):

```
» c = strncmp(str1,str3,2)
c =
1
```

Comparing Individual Characters for Equality and Inequality

You can use MATLAB relational operators on character arrays to test for equality *one character at a time*, as long as the arrays you are comparing have equal dimensions, or one is a scalar. For example, you can use the equality operator (==) to determine which characters in two strings match:

```
» a = 'fate';
» b = 'cake';
» result = a == b
result =
0 1 0 1
```

All of the relational operators (>, >=, <, <=, ==, ~=) compare the ASCII values of corresponding characters.

Unlike C, MATLAB does not have an intrinsic function to define a "greater than" or "less than" relationship between two strings taken as a whole. We will create such a function in an example at the end of this section.

Categorizing Characters Within a String

There are three functions for categorizing characters on a character-by-character basis inside a string:

- `isletter` determines if a character is a letter.
- `isspace` determines if a character is whitespace (blank, tab, or new line).
- `isstrprop('str', 'category')` is a more general function. It determines if a character falls into a user-specified category (e.g., alphabetic, alphanumeric, uppercase, lowercase, numeric, control).

To understand these functions, let's create a string named `mystring`:

```
mystring = 'Room 23a';
```

We will use this string to test the categorizing functions.

Function `isletter` examines each character in the string, producing a `logical` output vector of the same length as `mystring` that contains a true (1) in each location corresponding to a character and a false (0) in the other locations. For example,

```
» a = isletter(mystring)
a =
1 1 1 1 0 0 0 1
```

The first four and the last elements in a are true (1) because the corresponding characters of mystring are letters.

Function isspace also examines each character in the string, producing a logical output vector of the same length as mystring that contains a true (1) in each location corresponding to whitespace and a false (0) in the other locations. "Whitespace" is any character that separates tokens in MATLAB: a space, a tab, a linefeed, carriage return, etc. For example,

```
» a = isspace(mystring)
a =
0 0 0 0 1 0 0 0
```

The fifth element in a is true (1) because the corresponding character of mystring is a space.

Function isstrprop is new in MATLAB 7.0. It is a more flexible replacement for isletter, isspace, and several other functions. This function has two arguments, 'str' and 'category'. The first argument is the string to characterize, and the second argument is the type of category to check for. Some possible categories are given in Table 6.2.

Table 6.2 Selected Categories for Function isstrprop

Category	Description
'alpha'	Return true (1) for each character of the string that is alphabetic, and false (0) otherwise.
'alphanum'	Return true (1) for each character of the string that is alphanumeric, and false (0) otherwise.
	[**Note:** This category replaces function isletter.]
'cntrl'	Return true (1) for each character of the string is that is a control character, and false (0) otherwise.
'digit'	Return true (1) for each character of the string that is a number, and false (0) otherwise.
'lower'	Return true (1) for each character of the string that is a lowercase letter, and false (0) otherwise.
'wspace'	Return true (1) for each character of the string that is whitespace, and false (0) otherwise.
	[**Note:** This category replaces function isspace.]
'upper'	Return true (1) for each character of the string that is an uppercase letter, and false (0) otherwise.
'xdigit'	Return true (1) for each character of the string that is a hexadecimal digit, and false (0) otherwise.

This function examines each character in the string, producing a `logical` output vector of the same length as the input string that contains a true (1) in each location that matches the category, and a false (0) in the other locations. For example, the following function checks to see which characters in `mystring` are numbers:

```
» a = isstrprop(mystring,'digit')
a =
0 0 0 0 0 1 1 0
```

Also, the following function checks to see which characters in `mystring` are lowercase letters:

```
» a = isstrprop(mystring,'lower')
a =
0 1 1 1 0 0 0 1
```

Good Programming Practice

Use function `isstrprop` to determine the characteristics of each character in a string array. This function replaces the older functions `isletter` and `isspace`, which may be deleted in a future version of MATLAB.

Searching and Replacing Characters Within a String

MATLAB provides several functions for searching and replacing characters in a string. Consider a string named `test`:

```
test = 'This is a test!';
```

Function `findstr` returns the starting position of all occurrences of the shorter of two strings within a longer string. For example, to find all occurrences of the string `'is'` inside `test`,

```
» position = findstr(test,'is')
position =
     3     6
```

The string `'is'` occurs twice within `test`, starting at positions 3 and 6.

Function `strmatch` is another matching function. This one looks at the beginning characters of the *rows* of a 2-D character array and returns a list of those rows that start with the specified character sequence. The form of this function is

```
result = strmatch(str,array);
```

For example, suppose that we create a 2-D character array with the function `strvcat`:

```
array = strvcat('maxarray','min value','max value');
```

Then the following statement will return the row numbers of all rows beginning with the letters 'max':

```
» result = strmatch('max',array)
result =
           1
           3
```

Function strrep performs the standard search-and-replace operation. It finds all occurrences of one string within another one and replaces them by a third string. The form of this function is

```
result = strrep(str,srch,repl)
```

where str is the string being checked, srch is the character string to search for. and repl is the replacement character string. For example,

```
» test = 'This is a test!'
» result = strrep(test,'test','pest')
result =
This is a pest!
```

The strtok function returns the characters before the first occurrence of a delimiting character in an input string. The default delimiting characters constitute the set of whitespace characters. The form of strtok is

```
[token,remainder] = strtok(string,delim)
```

where string is the input character string, delim is the (optional) set of delimiting characters, token is the first set of characters delimited by a character in delim, and remainder is the rest of the line. For example,

```
» [token,remainder] = strtok('This is a test!')
token =
This
remainder =
is a test!
```

You can use the strtok function to parse a sentence into words; for example:

```
function all_words = words(input_string)
remainder = input_string;
all_words = '';
while (any(remainder))
   [chopped,remainder] = strtok(remainder);
   all_words = strvcat(all_words,chopped);
end
```

Uppercase and Lowercase Conversion

Functions upper and lower convert all of the alphabetic characters within a string to uppercase and lowercase respectively. For example,

```
» result = upper('This is test 1!')
result =
THIS IS TEST 1!
» result = lower('This is test 2!')
result =
this is test 2!
```

Note that the alphabetic characters were converted to the proper case, while the numbers and punctuation were unaffected.

Trimming Whitespace from Strings

There are two functions that trim leading and/or trailing whitespace from a string. Whitespace characters consists of the spaces, newlines, carriage returns, tabs, vertical tabs, and formfeeds.

Function deblank removes any extra *trailing* whitespace from a string, and function strtrim removes any extra *leading and trailing* whitespace from a string.

For example, the following statements create a 21-character string with leading and trailing whitespace. Function deblank trims the trailing whitespace characters in the string only, while function strtrim trims both the leading and the trailing whitespace characters.

```
» test_string = '   This is a test.
test_string =
   This is a test.
» length(test_string)
ans =
    21
» test_string_trim1= deblank(test_string)
test_string_trim1 =
   This is a test.
» length(test_string_trim1)
ans =
    18
» test_string_trim2 = strtrim(test_string)
test_string_trim2 =
This is a test.
» length(test_string_trim2)
ans =
    15
```

Numeric-to-String Conversions

MATLAB contains several functions to convert numeric values into character strings. We have already seen two such functions, `num2str` and `int2str`. Consider a scalar `x`:

```
x = 5317;
```

By default, MATLAB stores the number `x` as a 1 × 1 `double` array containing the value 5317. The `int2str` (integer to string) function converts this scalar into a 1-by-4 `char` array containing the string `'5317'`:

```
» x = 5317;
» y = int2str(x);
» whos
  Name        Size        Bytes        Class

  x           1x1         8            double array
  y           1x4         8            char array

Grand total is 5 elements using 16 bytes
```

Function `num2str` converts a `double` value into a string, even if it does not contain an integer. It provides more control of the output string format than `int2str`. An optional second argument sets the number of digits in the output string, or specifies an actual format to use. The format specifications in the second argument as similar to those used by `fprintf`. For example,

```
» p = num2str(pi)
p =
3.1416
» p = num2str(pi,7)
p =
3.141593
» p = num2str(pi,'%10.5e')
p =
3.14159e+000
```

Both `int2str` and `num2str` are handy for labeling plots. For example, the following lines use `num2str` to prepare automated labels for the *x*-axis of a plot:

```
function plotlabel(x,y)
plot(x,y)
str1 = num2str(min(x));
str2 = num2str(max(x));
out = ['Value of f from ' str1 ' to ' str2];
xlabel(out);
```

There are also conversion functions designed to change numeric values into strings representing a decimal value in another base, such as a binary or hexadecimal representation. For example, the dec2hex function converts a decimal value into the corresponding hexadecimal string:

```
dec_num = 4035;
hex_num = dec2hex(dec_num)
hex_num =
FC3
```

Other functions of this type include hex2num, hex2dec, bin2dec, dec2bin, base2dec, and dec2base. MATLAB includes on-line help for all of these functions.

MATLAB function mat2str converts an array to a string that MATLAB can evaluate. This string is useful input for a function such as eval, which evaluates input strings just as if they were typed at the MATLAB command line. For example, if we define array a as

```
» a = [1 2 3; 4 5 6]
a =
           1           2           3
           4           5           6
```

then the function mat2str will return a string containing the result

```
» b = mat2str(a)
b =
[1 2 3; 4 5 6]
```

Finally, MATLAB includes a special function sprintf that is identical to function fprintf, except that the output goes into a character string instead of the Command Window. This function provides complete control over the formatting of the character string. For example,

```
» str = sprintf('The value of pi = %8.6f.',pi)
str =
The value of pi = 3.141593.
```

This function is extremely useful in creating complex titles and labels for plots.

String-to-Numeric Conversions

MATLAB also contains several functions to change character strings into numeric values. The most important of these function are eval, str2double, and sscanf.

Function eval evaluates a string containing a MATLAB expression and returns the result. The expression can contain any combination of MATLAB functions, variables, constants, and operations. For example, the string a containing the characters '2 * 3.141592' can be converted to numeric form by the following statements:

```
» a = '2 * 3.141592';
» b = eval(a)
b =
          6.2832
» whos
  Name        Size        Bytes       Class

  a           1x8         16          char array
  b           1x1         8           double array

Grand total is 9 elements using 24 bytes
```

Function `str2double` converts character strings into an equivalent `double` value.[2] For example, the string a containing the characters `'3.141592'` can be converted to numeric form by the following statements:

```
» a = '3.141592';
» b = str2double(a)
b =
          3.1416
```

Strings can also be converted to numeric form using the function `sscanf`. This function converts a string into a number according to a format conversion character. The simplest form of this function is

```
value = sscanf(string,format)
```

where `string` is the string to scan and `format` specifies the type of conversion to occur. The two most common conversion specifiers for `sscanf` are `'%d'` for decimals and `'%g'` for floating-point numbers. This function is covered in much greater detail in Chapter 8.

The following examples illustrate the use of `sscanf`.

```
» a = '3.141592';
» value1 = sscanf(a,'%g')
value1 =
          3.1416
» value2 = sscanf(a,'%d')
value2 =
          3
```

Summary

The common MATLAB string functions are summarized in Table 6.3

[2]MATLAB also contains a function--`str2num`--that can convert a string into a number. For a variety of reasons mentioned in the MATLAB documentation, function `str2double` is better than function `str2num`. You should recognize function `str2num` when you see it, but always use function `str2double` in any new code that you write.

Table 6.3 Common MATLAB String Functions

Category	Function	Description
General	`char`	(1) Convert numbers to the corresponding character values. (2) Create a 2D character array from a series of strings.
	`double`	Convert characters to the corresponding numeric codes.
	`blanks`	Create a string of blanks.
	`deblank`	Remove trailing whitespace from a string.
	`strtrim`	Remove leading and trailing whitespace from a string.
String tests	`ischar`	Returns true (1) for a character array
	`isletter`	Returns true (1) for letters of the alphabet.
	`isspace`	Returns true (1) for whitespace.
	`isstrprop`	Returns true (1) for characters matching the specified property.
String operations	`strcat`	Concatenate strings.
	`strvcat`	Concatenate strings vertically.
	`strcmp`	Returns true (1) if two strings are identical.
	`strcmpi`	Returns true (1) if two strings are identical, ignoring case.
	`strncmp`	Returns true (1) if first n characters of two strings are identical.
	`strncmpi`	Returns true (1) if first n characters of two strings are identical, ignoring case.
	`findstr`	Find one string within another one.
	`strjust`	Justify string.
	`strmatch`	Find matches for string.
	`strrep`	Replace one string with another.
	`strtok`	Find token in string.
	`upper`	Convert string to uppercase.
	`lower`	Convert string to lowercase.
Number to string conversion	`int2str`	Convert integer to string.
	`num2str`	Convert number to string.
	`mat2str`	Convert matrix to string.
	`sprintf`	Write formatted data to string.
String to number conversion	`eval`	Evaluate the result of a MATLAB expression.
	`str2double`	Convert string to a double value.
	`str2num`	Convert string to number.
	`sscanf`	Read formatted data from string.
Base Number Conversion	`hex2num`	Convert IEEE hexadecimal string to `double`.
	`hex2dec`	Convert hexadecimal string to decimal integer.
	`dec2hex`	Convert decimal to hexadecimal string.
	`bin2dec`	Convert binary string to decimal integer.
	`dec2bin`	Convert decimal integer to binary string.
	`base2dec`	Convert base B string to decimal integer.
	`dec2base`	Convert decimal integer to base B string.

Example 6.2—String Comparison Function

In C, function strmcp compares two strings according to the order of their characters in the ASCII table (called the **lexicographic order** of the characters) and returns a -1 if the first string is lexicographically less than the second string, a 0 if the strings are equal, and a $+1$ if the first string is lexicographically greater than the second string. This function is extremely useful for such purposes as sorting strings in alphabetic order.

Create a new MATLAB function c_strcmp that compares two strings in a similar fashion to the C function and returns similar results. The function should ignore trailing blanks in doing its comparisons. Note that the function must be able to handle the situation where the two strings are of different lengths.

SOLUTION

1. **State the problem.**
   Write a function that will compare two strings str1 and str2, and return the following results:

- -1 if str1 is lexicographically less than str2.
- 0 if str1 is lexicographically less than str2.
- $+1$ if str1 is lexicographically greater than str2.

   The function must work properly if str1 and str2 do not have the same length, and the function should ignore trailing blanks.

2. **Define the inputs and outputs.**
   The inputs required by this function are two strings, str1 and str2. The output from the function will be a -1, 0, or 1, as appropriate.

3. **Describe the algorithm.**
   This task can be broken down into four major sections:

```
Verify input strings
Pad strings to be equal length
Compare characters from beginning to end, looking
    for the first difference
Return a value based on the first difference
```

   We will now break each of the foregoing major sections into smaller, more detailed pieces. First, we must verify that the data passed to the function is correct. The function must have exactly two arguments, and the arguments must both be characters. The pseudocode for this step is:

```
% Check for a legal number of input arguments.
msg = nargchk(2,2,nargin)
error(msg)
```

```
% Check to see if the arguments are strings
if either argument is not a string
   error('str1 and str2 must both be strings')
else

    (add code here)

end
```

Next, we must pad the strings to equal lengths. The easiest way to do this is to combine both strings into a 2-D array using `strvcat`. Note that this step effectively results in the function ignoring trailing blanks, because both strings are padded out to the same length. The pseudocode for this step is:

```
% Pad strings
strings = strvcat(str1,str2)
```

Now we must compare each character until we find a difference, and return a value based on that difference. One way to do this is to use relational operators to compare the two strings, creating an array of 0s and 1s. We can then look for the first one, which will correspond to the first difference between the two strings. The pseudocode for this step is:

```
% Compare strings
diff = strings(1,:) ~= strings(2,:)
if sum(diff) == 0
   % Strings match
   result = 0
else
   % Find first difference
   ival = find(diff)
   if strings(1,ival) > strings(2,ival)
      result = 1
   else
      result = -1
   end
end
```

4. **Turn the algorithm into MATLAB statements.**
   The final MATLAB code is shown below.

```
function result = c_strcmp(str1,str2)
%C_STRCMP Compare strings like C function "strcmp"
% Function C_STRCMP compares two strings, and returns
% a -1 if str1 < str2, a 0 if str1 == str2, and a
% +1 if str1 > str2.

% Define variables:
%   diff      -- Logical array of string differences
%    msg       -- Error message
%    result    -- Result of function
%    str1      -- First string to compare
%    str2      -- Second string to compare
%    strings   -- Padded array of strings

%  Record of revisions:
%    Date        Programmer        Description of change
%    ====        ==========        =====================
%  01/16/04    S. J. Chapman     Original code

% Check for a legal number of input arguments.
msg = nargchk(2,2,nargin);
error(msg);

% Check to see if the arguments are strings
if  ~(isstr(str1) & isstr(str2))
    error('Both str1 and str2 must both be strings!')
else

   % Pad strings
   strings = strvcat(str1,str2);

   % Compare strings
   diff = strings(1,:) ~= strings(2,:);
   if sum(diff) == 0

      % Strings match, so return a zero!
      result = 0;
   else
      % Find first difference between strings
      ival = find(diff);
      if strings(1,ival(1)) > strings(2,ival(1))
         result = 1;
      else
         result = -1;
      end
   end
end

end % function c_strcmp
```

5. **Test the program.**
   Next, we must test the function using various strings.

```
» result = c_strcmp('String 1','String 1')
```

```
result =
          0
» result = c_strcmp('String 1','String 1 ')
result =
          0
» result = c_strcmp('String 1','String 2')
result =
         -1
» result = c_strcmp('String 1','String 0')
result =
          1
» result = c_strcmp('String','str')
result =
         -1
```

The first test returns a zero, because the two strings are identical. The second test also returns a zero, because the two strings are identical *except for trailing blanks* and trailing blanks are ignored. The third test returns a −1, because the two strings first differ in position 8 and `'1'` < `'2'` at that position. The fourth test returns a 1, because the two strings first differ in position 8 and `'1'` > `'0'` at that position. The fifth test returns a −1, because the two strings first differ in position 1, and `'S'` < `'s'` in the ASCII collating sequence.

This function appears to be working properly.

Quiz 6.1

This quiz provides a quick check to see if you have understood the concepts introduced in Sections 6.1 through 6.2. If you have trouble with the quiz, reread the section, ask your instructor, or discuss the material with a fellow student. The answers to this quiz are found in the back of the book.

1. What is the value of `result` in the following statements?

(*a*)
```
x = 12 + i*5;
y = 5 - i*13;
result = x > y;
```

(*b*)
```
x = 12 + i*5;
y = 5 - i*13;
result = abs(x) > abs(y);
```

(*c*)
```
x = 12 + i*5;
y = 5 - i*13;
```

```
result = real(x) - imag(y);
```

2. If `array` is a complex array, what does the function `plot(array)` do?

3. How can you convert a vector of the `char` data type into a vector of the `double` data type?

For questions 4 through 11, determine whether these statements are correct. If they are, what is produced by each set of statements?

4.
```
str1 = 'This is a test! ';
str2 = 'This line, too.';
res = strcat(str1,str2);
```

5.
```
str1 = 'Line 1';
str2 = 'line 2';
res = strcati(str1,str2);
```

6.
```
str1 = 'This is a test! ';
str2 = 'This line, too.';
res = [str1; str2];
```

7.
```
str1 = 'This is a test! ';
str2 = 'This line, too.';
res = strvcat(str1,str2);
```

8.
```
str1 = 'This is a test! ';
str2 = 'This line, too.';
res = strncmp(str1,str2,5);
```

9.
```
str1 = 'This is a test! ';
res = findstr(str1,'s');
```

10.
```
str1 = 'This is a test! ';
str1(isspace(str1)) = x;
```

11.
```
str1 = 'aBcD 1234 !?';
res = isstrprop(str1,'alphanum');
```

12.
```
str1 = 'This is a test! ';
str1(4:7) = upper(str1(4:7));
```

13.
```
str1 = '   456   '; % Note: Three blanks before & after
str2 = '   abc   '; % Note: Three blanks before & after
str3 = [str1 str2];
str4 = [strtrim(str1) strtrim(str2)];
str5 = [deblank(str1) deblank(str2)];
l1 = length(str1);
l3 = length(str3);
```

```
   l4 = length(str4);
   l5 = length(str4);
```

14.
```
   str1 = 'This way to the egress.';
   str2 = 'This way to the egret.'
   res = strncmp(str1,str2);
```

6.3 Multidimensional Arrays

MATLAB also supports arrays with more than two dimensions. These **multidimensional arrays** are very useful for displaying data that intrinsically has more than two dimensions, or for displaying multiple versions of 2-D data sets. For example, measurements of pressure and velocity throughout a three-dimensional volume are very important in such studies as aerodynamics and fluid dynamics. These areas naturally use multidimensional arrays.

Multidimensional arrays are a natural extension of two-dimensional arrays. Each additional dimension is represented by one additional subscript used to address the data.

It is very easy to create multidimensional arrays. They can be created either by assigning values directly in assignment statements or by using the same functions that are used to create one- and two-dimensional arrays. For example, suppose that you have a two-dimensional array created by the assignment statement

```
» a = [1 2 3 4; 5 6 7 8]
a =
          1         2         3         4
          5         6         7         8
```

This is a 2 × 4 array, with each element addressed by two subscripts. The array can be extended to be a three-dimensional 2 × 4 × 3 array with the following assignment statements.

```
» a(:,:,2) = [ 9 10 11 12; 13 14 15 16];
» a(:,:,3) = [ 17 18 19 20; 21 22 23 24]
a(:,:,1) =
          1         2         3         4
          5         6         7         8
a(:,:,2) =
          9        10        11        12
         13        14        15        16
a(:,:,3) =
         17        18        19        20
         21        22        23        24
```

Individual elements in this multidimensional array can be addressed by the array name followed by three subscripts, and subsets of the data can be created using the colon operators. For example, the value of `a(2,2,2)` is

```
» a(2,2,2)
ans =
          14
```

and the vector `a(1,1,:)` is

```
» a(1,1,:)
ans(:,:,1) =
          1
ans(:,:,2) =
          9
ans(:,:,3) =
          17
```

Multidimensional arrays can also be created using the same functions as other arrays, for example:

```
» b = ones(4,4,2)
b(:,:,1) =
          1          1          1          1
          1          1          1          1
          1          1          1          1
          1          1          1          1
b(:,:,2) =
          1          1          1          1
          1          1          1          1
          1          1          1          1
          1          1          1          1
» c = randn(2,2,3)
c(:,:,1) =
        -0.4326         0.1253
        -1.6656         0.2877
c(:,:,2) =
        -1.1465         1.1892
         1.1909        -0.0376
c(:,:,3) =
         0.3273        -0.1867
         0.1746         0.7258
```

The number of dimensions in a multidimensional array can be found using the `ndims` function, and the size of the array can be found using the `size` function

```
» ndims(c)
ans =
          3
» size(c)
ans =
          2     2     3
```

If you are writing applications that need multidimensional arrays, see the MATLAB Users Guide for more details on the behavior of various MATLAB functions with multidimensional arrays.

Good Programming Practice

Use multidimensional arrays to solve problems that are naturally multivariate in nature, such as aerodynamics and fluid flows.

Also, recall from Chapter 4 that the MATLAB just-in-time compiler cannot compile loops containing arrays with three or more dimensions. If you are working with such arrays, be sure to vectorize your code to increase its speed. Do not rely on the JIT compiler to do the job—it won't.

Good Programming Practice

If you are working with multidimensional arrays, be sure to vectorize your code by hand. The MATLAB JIT compiler cannot handle loops containing multidimensional arrays with three or more dimensions.

6.4 Additional Data Types

MATLAB also includes a `single` data type and several integer data types. They are briefly discussed in the following sections.

The single Data Type

Variables of type `single` are scalars or arrays of 32-bit *single-precision* floating-point numbers. They can hold real, imaginary, or complex values. Variables of type `single` occupy half the memory of variables of type `double`, but they have lower precision and a more limited range. The real and imaginary components of each `single` variable can be positive or negative numbers in the range 10^{-38} to 10^{38}, with six to seven significant decimal digits of accuracy.

The `single` function creates a variable of type `single`. For example, the following statement creates a variable of type `single` containing the value 3.1:

```
» var = single(3.1)
var =
         3.1000
» whos
    Name    Size         Bytes      Class
    var     1x1              4      single array
  Grand total is 1 element using 4 bytes
```

Once a `single` variable is created, it can be used in MATLAB operations just like a `double` variable. In MATLAB, an operation performed between a `single` value and a `double` value has a `single` result,[3] so the result of the following statements will be of type `single`:

```
» b = 7;
» c = var * b
c =
       21.7000
» whos
  Name          Size          Bytes          Class

  b             1x1           8              double array
  c             1x1           4              single array
  var           1x1           4              single array

Grand total is 1 element using 4 bytes
```

The availability of mathematical operations with the `single` data type is a new feature of MATLAB 7.0. Values of type `single` can be used just like values of type `double` in most MATLAB operations. Built-in functions such as `sin`, `cos`, `exp`, and so forth all support the `single` data type, but some M-file functions may not support `single` values yet. (For example, comparisons for near equality between two numbers may be incorrect if the function is expecting `double` values and instead is passed `single` values.) As a practical matter, you will probably never use this data type. Its more limited range and precision make the results more sensitive to cumulative round-off errors or to exceeding the available range. You should consider using this data type only if you have enormous arrays of data that could not fit into your computer memory if they were saved in double precision.

Also, the MATLAB just-in-time compiler cannot compile loops containing `single` values. If you are working with such arrays, be sure to vectorize your code to increase its speed. Do not rely on the JIT compiler to do the job—it won't.

Integer Data Types

MATLAB also includes 8-, 16-, 32-, and 64-bit *signed* and *unsigned* integers. The data types are `int8`, `uint8`, `int16`, `uint16`, `int32`, `uint32`, `int64`, and `uint64`. The difference between a signed and an unsigned integer is the range of numbers represented by the data type. The number of values that can be represented by an integer depends on the number of bits in the integer:

$$\text{number of values} = 2^n \qquad (6\text{-}11)$$

where n is the number of bits. An 8-bit integer can represent 256 values (2^8), a 16-bit integer can represent 65,536 values (2^{16}), and so forth. Signed integers use

[3]CAUTION: This is unlike the behavior of any other computer language that the author has ever encountered. In every other language (Fortran, C, C++, Java, Basic, etc.), the result of an operation between a `single` and a `double` would be of type `double`.

half of the available values to represent positive numbers and half for negative numbers, whereas unsigned integers use all of the available values to represent positive numbers. Therefore, the range of values that can be represented in the `int8` data type is −128 to 127 (a total of 256), while the range of values that can be represented in the `uint8` data type is 0 to 255 (a total of 256). Similarly, the range of values that can be represented in the `int16` data type is −32,768 to 32,767 (a total of 65,536), while the range of values that can be represented in the `uint16` data type is 0 to 65,535. The same idea applies to larger integer sizes.

Integer values are created by the `int8()`, `uint8()`, `int16()`, `uint16()`, `int32()`, `uint32()`, `int64()`, or `uint64()` functions. For example, the following statement creates a variable of type `int8` containing the value 3:

```
» var = int8(3)
var =
        3
» whos
  Name        Size        Bytes        Class
  var         1x1         1            int8 array
Grand total is 1 element using 1 bytes
```

Integers can be converted to other data types using the `double` and `single` functions.

An operation performed between an integer value and a `double` value has an integer result,[4] so the result of the following statements will be of type `int8`:

```
» b = 7;
» c = var * b
c =
       21
» whos
  Name        Size        Bytes        Class
  b           1x1         8            double array
  c           1x1         1            int8 array
  var         1x1         1            int8 array
Grand total is 3 elements using 10 bytes
```

MATLAB uses *saturating integer arithmetic*. If the result of an integer math operation would be larger than the largest possible value that can be represented in that data type, the result will be the largest possible value. Similarly, if the result of an integer math operation would be smaller than the smallest possible value that can be represented in that data type, the result will be the smallest possible value. For example, the largest possible value that can be represented in the

[4]CAUTION: This is unlike the behavior of any other computer language that the author has ever encountered. In every other language (Fortran, C, C++, Java, Basic, etc.), the result of an operation between an integer and a `double` would be of type `double`.

`int8` data type is 127. The result of the operation `int8(100) + int8(50)` will be 127, because 150 is larger than 127, the maximum value that can be represented in the data type.

It is unlikely that you will need to use the integer data type unless you are working with image data. If you do need more information, please consult the MATLAB documentation.

Limitations of the single and Integer Data Types

The `single` data type and integer data types have been around in MATLAB for a while, but they have been mainly used for purposes such as storing image data. Before MATLAB 7.0, it was not possible to perform mathematical operations (+, −, etc.) with these data types. MATLAB is now evolving to make manipulating these data types easier, but the support is still rough in the current release. There are significant gaps. For example, you can add a `single` and a `double`, or an integer and a `double`, but not a `single` and an integer.

```
» a = single(2.1)
a =
        2.1000
» b = int16(4)
b =
           4
» c = a+b
??? Error using ==> plus
Class of operand is not supported.
```

Unless you have some special need to manipulate images, you will probably never need to use either of these data types.

Good Programming Practice

Do not use the `single` or integer data types, unless you have a special need such as image processing.

6.5 Additional Two-Dimensional Plots

In previous chapters, we have learned to create linear, log-log, semilog, and polar plots. MATLAB supports many additional types of plots that you can use to display your data. This section will introduce you to some of these additional plotting options.

Additional Types of Two-Dimensional Plots

In addition to the two-dimensional plots that we have already seen, MATLAB supports *many* other more specialized plots. In fact, the MATLAB help desk lists more than 20 types of two-dimensional plots! Examples include **stem plots**, **stair plots**, **bar plots**, **pie plots**, and **compass plots**. A *stem plot* is a plot in which each data value is represented by a marker and a line connecting the marker vertically to the *x* axis. A *stair plot* is a plot in which each data point is represented by a horizontal line, and successive points are connected by vertical lines, producing a stair-step effect. A *bar plot* is a plot in which each point is represented by a vertical bar or horizontal bar. A *pie plot* is a plot represented by "pie slices" of various sizes. Finally, a *compass plot* is a type of polar plot in which each value is represented by an arrow whose length is proportional to its value. These plots are summarized in Table 6.4, and examples of all of the plots are shown in Figure 6.8.

Stair, stem, vertical bar, horizontal bar, and compass plots are all similar to `plot`, and they are used in the same manner. For example, the following code produces the stem plot shown in Figure 6.7*a*.

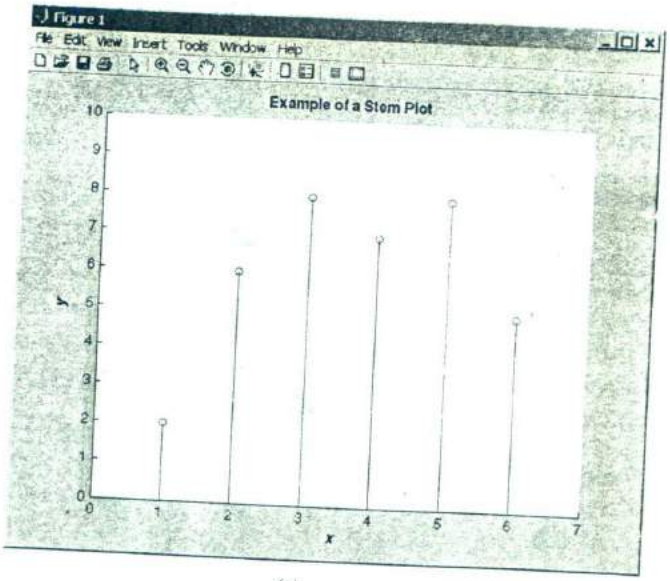
```
x = [1 2 3 4 5 6];
y = [2 6 8 7 8 5];
stem(x,y);
title('\bfExample of a Stem Plot');
xlabel('\bf\itx');
ylabel('\bf\ity');
axis([0 7 0 10]);
```

Table 6.4 Additional Two-Dimensional Plotting Functions

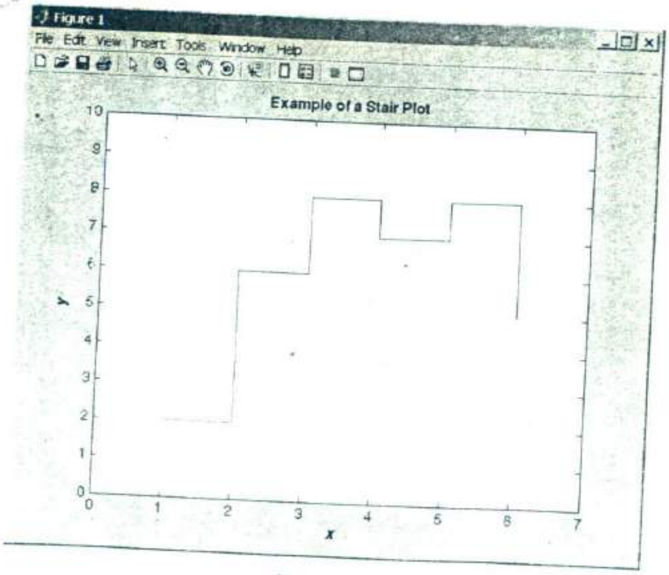
Function	Description
`bar(x,y)`	This function creates a *vertical* bar plot, with the values in `x` used to label each bar and the values in `y` used to determine the height of the bar.
`barh(x,y)`	This function creates a *horizontal* bar plot, with the values in `x` used to label each bar and the values in `y` used to determine the horizontal length of the bar.
`Compass(x,y)`	This function creates a polar plot, with an arrow drawn from the origin to the location of each (x, y) point. Note that the locations of the points to plot are specified in Cartesian coordinates, not polar coordinates.

`pie(x)`
`pie(x,explode)`
	This function creates a pie plot. This function determines the percentage of the total pie corresponding to each value of `x` and plots pie slices of that size. The optional array `explode` controls whether or not individual pie slices are separated from the remainder of the pie.

`Stairs(x,y)`	This function creates a stair plot, with each stair step centered on an (x, y) point.
`stem(x,y)`	This function creates a stem plot, with a marker at each (x, y) point and a stem drawn vertically from that point to the x axis.

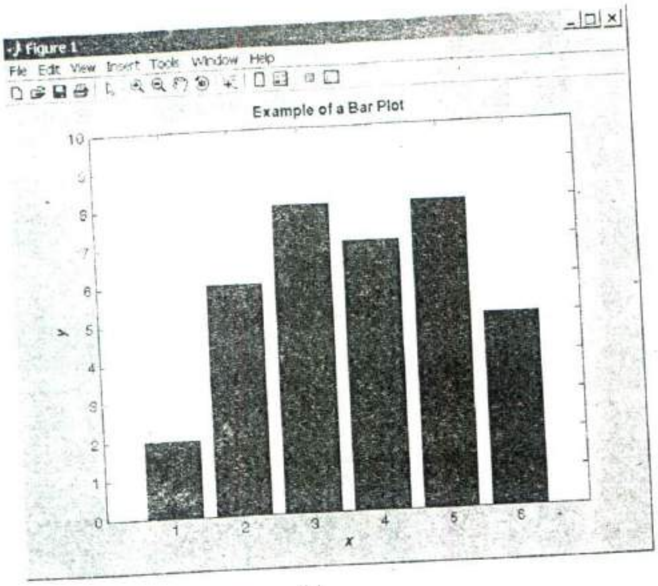


(a)

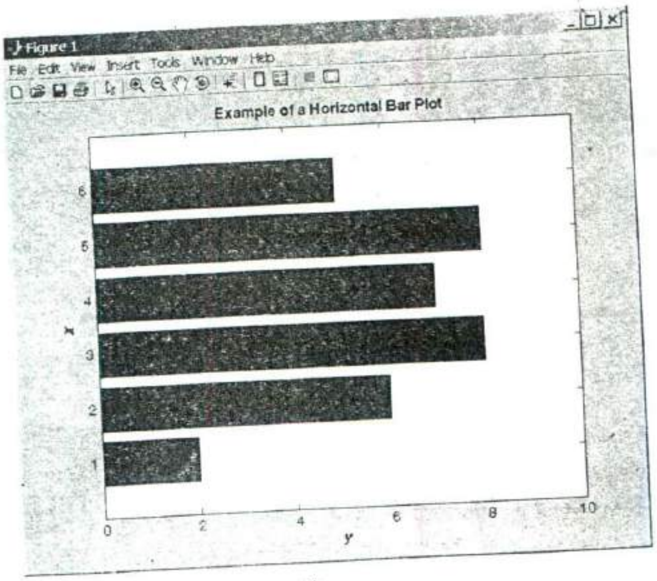


(b)

Figure 6.8 Additional types of 2D plots: *(a)* stem plot; *(b)* stair plot;

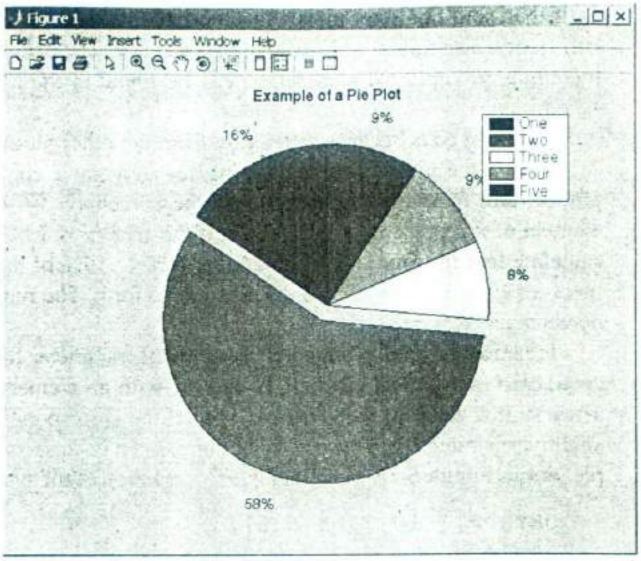


(c)

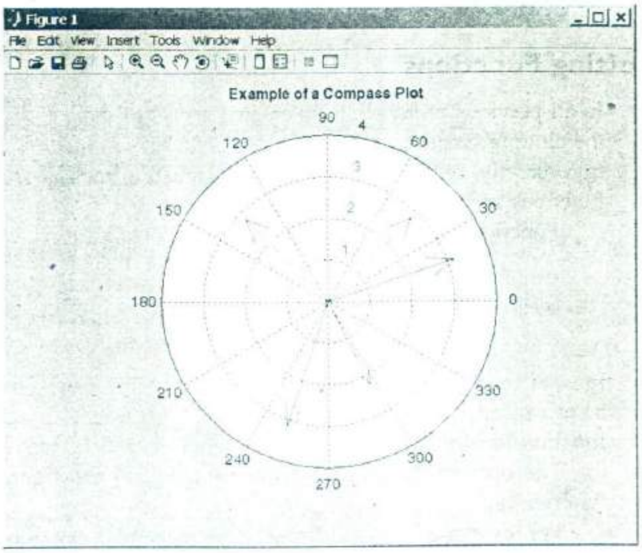


(d)

Figure 6.8 *(continued) (c)* vertical bar plot; *(d)* horizontal bar plot;



(e)



(f)

Figure 6.8 *(continued)* *(e)* pie plot; *(f)* compass plot.

Stair, bar, and compass plots can be created by substituting `stairs`, `bar`, `barh`, or `compass` for `stem` in the above code. The details of all of these plots, including any optional parameters, can be found in the MATLAB on-line help system.

Function `pie` behaves differently from the other plots described previously. To create a pie plot, a programmer passes an array `x` containing the data to be plotted, and function `pie` determines the *percentage of the total pie* that each element of `x` represents. For example, if the array `x` is `[1 2 3 4]`, then `pie` will calculate that the first element `x(1)` is 1/10 or 10% of the pie, the second element `x(2)` is 2/10 or 20% of the pie, and so forth. The function then plots those percentages as pie slices.

Function `pie` also supports an optional parameter, `explode`. If present, `explode` is a logical array of 1s and 0s, with an element for each element in array `x`. If a value in `explode` is 1, then the corresponding pie slice is drawn slightly separated from the pie. For example, the code shown below produces the pie plot in Figure 6.7*e*. Note that the second slice of the pie is "exploded".

```
data = [ 10 37 5 6 6];
explode = [ 0 1 0 0 0];
pie(data,explode);
title('\bfExample of a Pie Plot');
legend('One','Two','Three','Four','Five');
```

Plotting Functions

In all previous plots, we have created arrays of data, and passed those arrays to the plotting function. MATLAB also includes two functions that will plot a function directly, without the necessity of creating intermediate data arrays. These functions are `ezplot` and `fplot`.

Function `ezplot` takes one of the following forms.

```
ezplot( fun );
ezplot( fun, [xmin xmax] );
ezplot( fun, [xmin xmax], figure );
```

In each case, `fun` is a *character string* containing the functional expression to be evaluated. The optional parameter `[xmin xmax]` specifies the range of the function to plot. If it is absent, the function will be plotted between -2π and 2π. The optional parameter `figure` specifies the figure number to plot the function on.

For example, the following statements plot the function $f(x) = \sin x/x$ between -4π and 4π. The output of these statements is shown in Figure 6.9.

```
ezplot('sin(x)/x',[-4*pi 4*pi]);
title('Plot of sin x / x');
grid on;
```

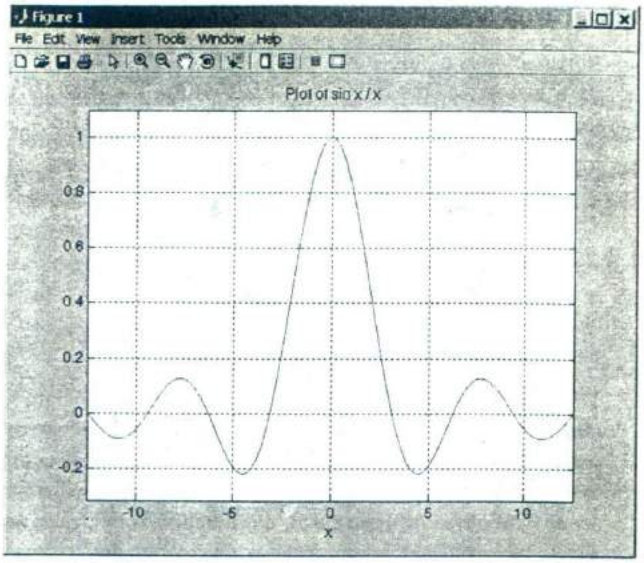


Figure 6.9 The function sin x/x, plotted with function ezplot.

Function `fplot` is similar to but more sophisticated than `ezplot`. The first two arguments are the same for both functions, but `fplot` has the following advantages:

1. Function `fplot` is *adaptive*, meaning that it calculates and displays more data points in the regions where the function being plotted is changing most rapidly. The resulting plot is more accurate at locations where a function's behavior changes suddenly.
2. Function `fplot` supports the use of T_EX commands in titles and axis labels, while function `ezplot` does not.

In general, you should use `fplot` in preference to `ezplot` whenever you plot functions.

Functions `ezplot` and `fplot` are examples of the "function functions" described in Chapter 5.

✱ Good Programming Practice

Use function `fplot` to plot functions directly without having to create intermediate data arrays.

Histograms

A *histogram* is a plot showing the distribution of values within a data set. To create a histogram, the range of values within the data set is divided into evenly spaced bins, and the number of data values falling into each bin is determined. The resulting count can then be plotted as a function of bin number.

The standard MATLAB histogram function is `hist`. The forms of this function are shown below:

```
hist(y)
hist(y,nbins)
hist(y,x);
[n,xout] = hist(y,...)
```

The first form of the function creates and plots a histogram with ten equally spaced bins, while the second form creates and plots a histogram with `nbins` equally spaced bins. The third form of the function allows the user to specify the bin centers to use in an array `x`; the function creates a bin centered around each element in the array. In all three of these cases, the function both creates and plots the histogram. The last form of the function creates a histogram and returns the bin centers in array `xout` and the count in each bin in array `n`, without actually creating a plot.

For example, the following statements create a data set containing 10,000 Gaussian random values, and generate a histogram of the data using 15 evenly spaced bins. The resulting histogram is shown in Figure 6.10.

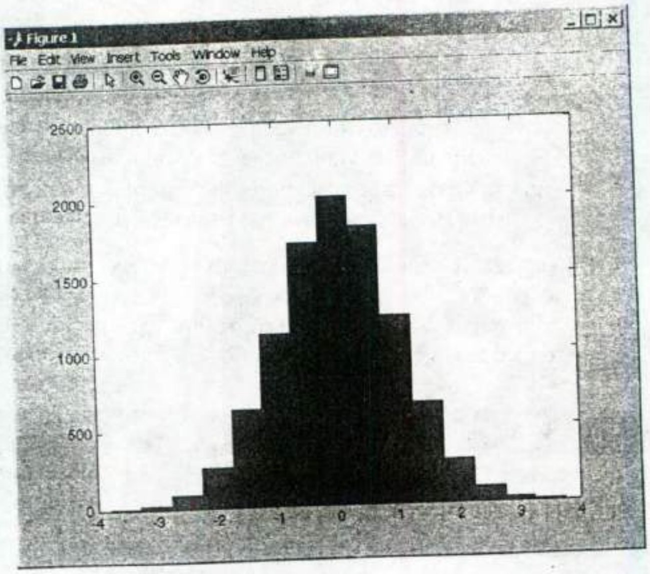


Figure 6.10 A histogram.

```
y = randn(10000,1);
hist(y,15);
```

MATLAB also includes a function `rose` to create and plot a histogram on radial axes. It is especially useful for distributions of angular data. You will be asked to use this function in an end-of-chapter exercise.

6.6 Three-Dimensional Plots

MATLAB also includes a rich variety of three-dimensional plots that can be useful for displaying certain types of data. In general, three-dimensional plots are useful for displaying two types of data:

1. Two variables that are functions of the same independent variable, when you wish to emphasize the importance of the independent variable.
2. A single variable that is a function of two independent variables.

Three-Dimensional Line Plots

A three-dimensional line plot can be created with the `plot3` function. This function is exactly like the two-dimensional `plot` function, except that each point is represented by x, y, and z values instead just of x and y values. The simplest form of this function is

```
plot(x,y,z);
```

where `x`, `y`, and `z` are equal-sized arrays containing the locations of data points to plot. Function `plot3` supports all the same line size, line style, and color options as `plot`, and you can use it immediately using the knowledge that we acquired in earlier chapters.

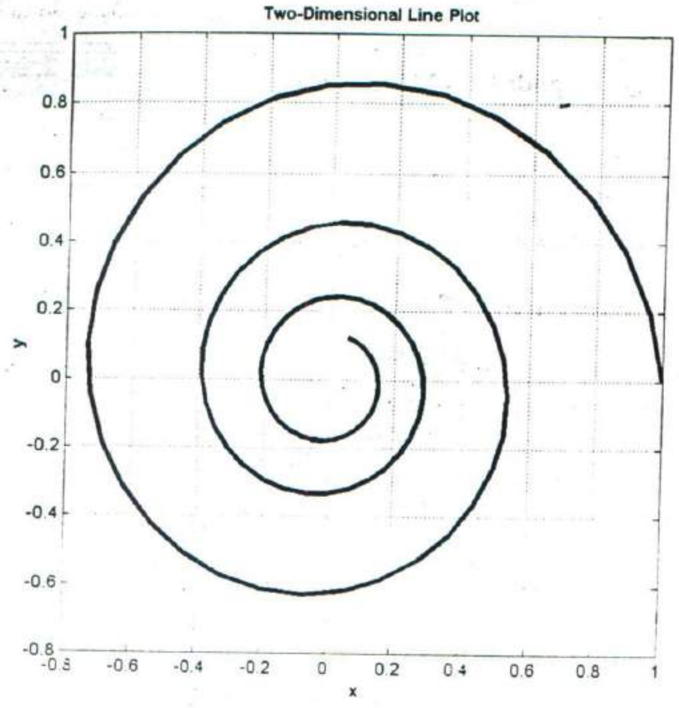
As an example of a three-dimensional line plot, consider the following functions:

$$\begin{aligned} x(t) &= e^{-0.2t}\cos 2t \\ y(t) &= e^{-0.2t}\sin 2t \end{aligned} \tag{6-12}$$

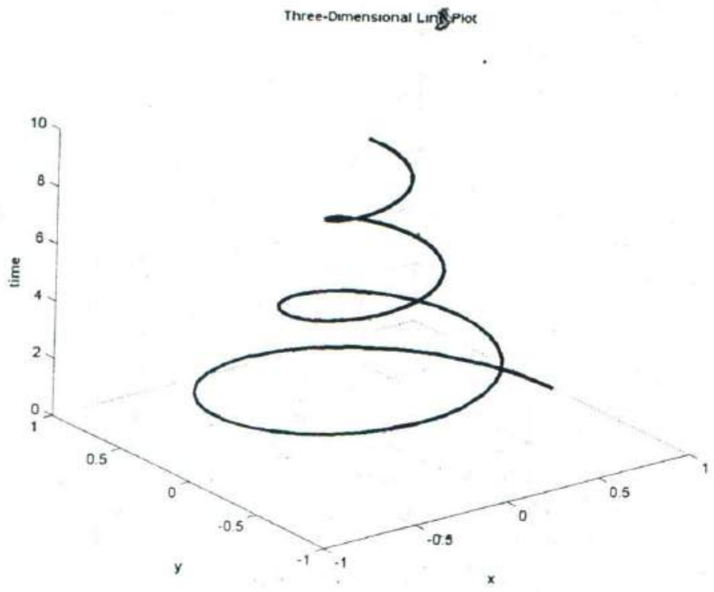
These functions might represent the decaying oscillations of a mechanical system in two dimensions, so x and y together represent the location of the system at any given time. Note that x and y are both functions of the *same* independent variable t.

We could create a series of (x, y) points and plot them using the two-dimensional function `plot` (see Figure 6.11*a*), but if we do so, the importance of time to the behavior of the system will not be obvious in the graph. The following statements create the two-dimensional plot of the location of the object shown in Figure 6.11*a*. It is not possible from this plot to tell how rapidly the oscillations are dying out.

```
t = 0:0.1:10;
x = exp(-0.2*t) .* cos(2*t);
```



(a)



(b)

Figure 6.11 *(a)* A two-dimensional line plot showing the motion in (x, y) space of a mechanical system. This plot reveals nothing about the time behavior of the system. *(b)* A three-dimensional line plot showing the motion in (x, y) space versus time for the mechanical system. This plot clearly shows the time behavior of the system.

```
y = exp(-0.2*t) .* sin(2*t);
plot(x,y);
title('\bfTwo-Dimensional Line Plot');
xlabel('\bfx');
ylabel('\bfy');
grid on;
```

Instead, we could plot the variables with `plot3` to preserve the time information as well as the two-dimensional position of the object. The following statements will create a three-dimensional plot of Equations (6-12).

```
t = 0:0.1:10;
x = exp(-0.2*t) .* cos(2*t);
y = exp(-0.2*t) .* sin(2*t);
plot3(x,y,t);
title('\bfThree-Dimensional Line Plot');
xlabel('\bfx');
ylabel('\bfy');
zlabel('\bftime');
grid on;
```

The resulting plot is shown in Figure 6.11*b*. Note how this plot emphasizes time-dependence of the two variables x and y.

Three-Dimensional Surface, Mesh, and Contour Plots

Surface, mesh, and contour plots are convenient ways to represent data that is a function of *two* independent variables. For example, the temperature at a point is a function of both the East-West location (x) and the North-South (y) location of the point. Any value that is a function of two independent variables can be displayed on a three-dimensional surface, mesh, or contour plot. The more common types of plots are summarized in Table 6.5, and examples of each plot are shown in Figure 6.12.[5]

To plot data using one of these functions, a user must create three equal-sized arrays. The three arrays must contain the x, y, and z values of every point to be plotted. As a simple example, suppose that we wanted to plot the four points $(-1, -1, 1)$, $(1, -1, 2)$, $(-1, 1, 1)$, and $(1, 1, 0)$. To plot these four points, we must create the arrays $\mathtt{x} = \begin{bmatrix} -1 & 1 \\ -1 & 1 \end{bmatrix}$, $\mathtt{y} = \begin{bmatrix} -1 & -1 \\ 1 & 1 \end{bmatrix}$, and $\mathtt{z} = \begin{bmatrix} 1 & 2 \\ 1 & 0 \end{bmatrix}$. Array `x` contains the x values associated with every point to plot, array `y` contains the y values associated with every point to plot, and array `z` contains the z values associated with every point to plot. These arrays are then passed to the plotting function.

[5]There are many variations on these basic plot types. Consult the MATLAB Help Browser documentation for a complete description of these variations.

Table 6.5 Selected Mesh, Surface, and Contour Plot Functions

Function	Description
`mesh(x,y,z)`	This function creates a mesh or wireframe plot, where x is a two-dimensional array containing the *x* values of every point to display, y is a two-dimensional array containing the *y* values of every point to display, and z is a two-dimensional array containing the *z* values of every point to display.
`surf(x,y,z)`	This function creates a surface plot. Arrays x, y, and z have the same meaning as for a mesh plot.
`contour(x,y,z)`	This function creates a contour plot. Arrays x, y, and z have the same meaning as for a mesh plot.

The MATLAB function `meshgrid` makes it easy to create the `x` and `y` arrays required for these plots. The form of this function is

```
[x y] = meshgrid( xstart:xinc:xend, ystart:yinc:yend);
```

where `xstart:xinc:xend` specifies the x values to include in the grid and `ystart:yinc:yend` specifies the y values to be included in the grid.

To create a plot, we use `meshgrid` to create the arrays of x and y values and then evaluate the function to plot at each of those (x, y) locations. Finally, we call function `mesh`, `surf`, or `contour` to create the plot.

For example, suppose that we wish to create a mesh plot of the function

$$z(x, y) \;=\; e^{-0.5[x^2+0.5(x-y)^2]} \tag{6-13}$$

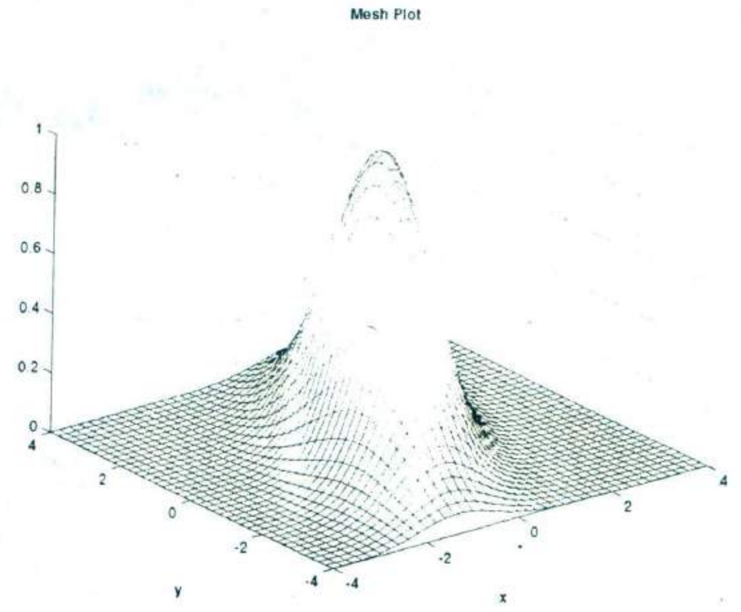
over the interval $-4 \leq x \leq 4$ and $-4 \leq y \leq 4$. The following statements will create the plot, which is shown in Figure 6.12*a*.

```
[x,y] = meshgrid(-4:0.2:4);
z = exp(-0.5*(x.^2+y.^2));
mesh(x,y,z);
xlabel('\bfx');
ylabel('\bfy');
zlabel('\bfz');
```

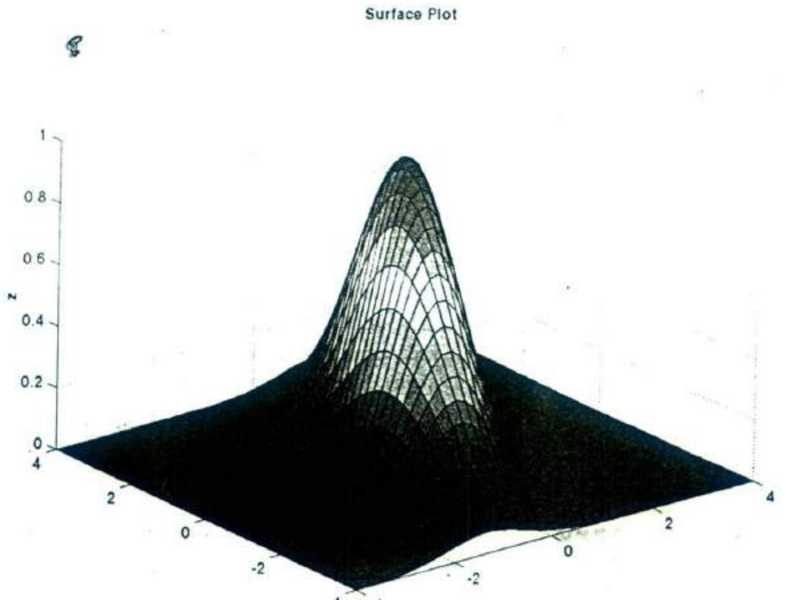
Surface and contour plots may be created by substituting the appropriate function for the `mesh` function.

6.7 Summary

MATLAB supports complex numbers as an extension of the `double` data type. They can be defined using the `i` or `j`, both of which are predefined as to be $\sqrt{-1}$. Using complex numbers is straightforward, except that the relational



(a)



(b)

Figure 6.12 *(a)* A mesh plot of the function $z(x, y) = e^{-0.5[x^2+0.5(x-y)^2]}$. *(b)* A surface plot of the same function.

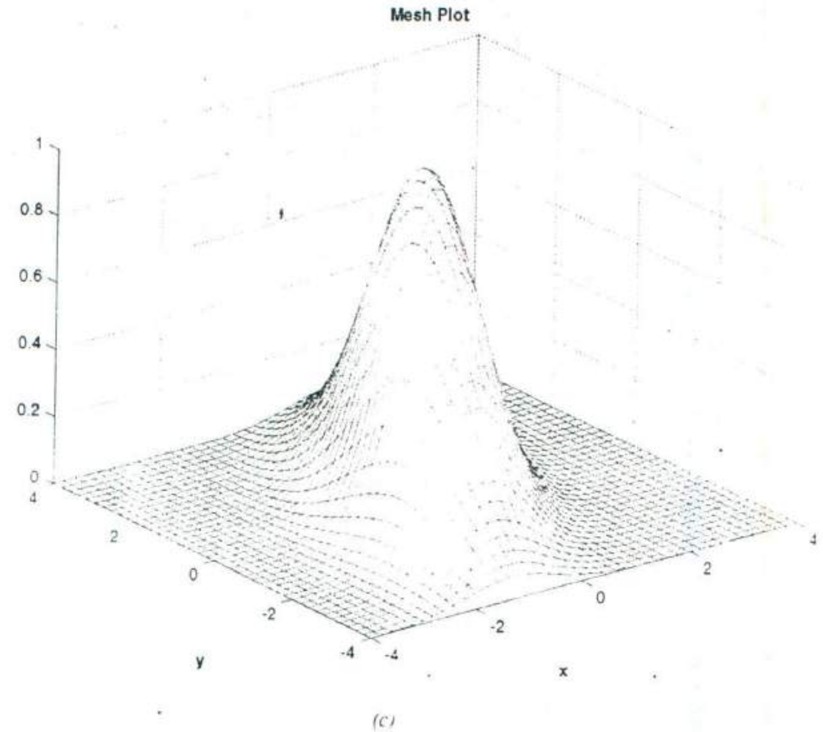


Figure 6.12 (*continued*) (c) A contour plot of the same function.

operators >. >=, <, and <= compare only the *real parts* of complex numbers, not their magnitudes. They must be used with caution when working with complex values.

String functions are functions designed to work with strings, which are arrays of type `char`. These functions allow a user to manipulate strings in a variety of useful ways, including concatenation, comparison, replacement, case conversion, and numeric-to-string and string-to-numeric type conversions.

Multidimensional arrays are arrays with more than two dimensions. They may be created and used in a fashion similar to one- and two-dimensional arrays. Multidimensional arrays appear naturally in certain classes of physical problems.

The `single` data is consists of single-precision floating point numbers. They are created using the `single` function. A mathematical operation between a `single` and a `double` value produces a `single` result.

MATLAB includes signed and unsigned 8-, 16-, 32-, and 64-bit integers. The integer data types are the `int8, uint8, int16(), uint16, int32, uint32, int64`, and `uint64`. Each of these types is created using the corresponding function: `int8(), uint8(), int16(), uint16(). int32(), uint32(). int64()`, or `uint64()`. Mathematical operations (+, −. etc.) can be performed on these data types; the result of an operation between an integer and a `double` has the same type as the integer. If the result of a mathematical operation is too large or too small to be expressed by an integer data type, the result is either the largest or smallest possible integer for that type.

MATLAB includes a rich variety of two- and three-dimensional plots. In this chapter, we introduced stem, stair, bar, compass, mesh, surface, and contour plots.

Summary of Good Programming Practice

The following guidelines should be adhered to:

1. Use the `char` function to create two-dimensional character arrays without worrying about padding each row to the same length.
2. Use function `isstrprop` to determine the characteristics of each character in a string array. This function supercedes the older functions `isletter` and `isspace`, which may be deleted in a future version of MATLAB.
3. Use multidimensional arrays to solve problems that are naturally multivariate in nature, such as aerodynamics and fluid flows.
4. If you are working with multidimensional arrays, be sure to vectorize you code by hand. The MATLAB JIT compiler cannot handle loops containing multidimensional arrays with three or more dimensions.
5. Do not use the `single` or integer data types, unless you have a special need such as image processing.
6. Use function `fplot` to plot functions directly without having to create intermediate data arrays.

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.

`abs`	Returns absolute value (magnitude) of a number.
`angle`	Returns the angle of a complex number, in radians.
`bar(x,y)`	Create a vertical bar plot.
`barh(x,y)`	Create a horizontal bar plot.
`base2dec`	Convert base B string to decimal integer.
`bin2dec`	Convert binary string to decimal integer.
`blanks`	Create a string of blanks.
`char`	(1) Convert numbers to the corresponding character values. (2) Create a 2D character array from a series of strings.
`compass(x,y)`	Create a compass plot.
`conj`	Compute complex conjugate of a number.
`contour`	Create a contour plot.

`deblank`	Remove trailing whitespace from a string.
`dec2base`	Convert decimal integer to base B string.
`dec2bin`	Convert decimal integer to binary string.
`double`	Convert characters to the corresponding numeric codes.
`find`	Find indices and values of nonzero elements in a matrix.
`findstr`	Find one string within another one.
`hex2num`	Convert IEEE hexadecimal string to `double`.
`hex2dec`	Convert hexadecimal string to decimal integer.
`hist`	Create a histogram of a data set.
`full`	Convert a sparse matrix into a full matrix
`imag`	Returns the imaginary portion of the complex number.
`int2str`	Convert integer to string.
`ischar`	Returns true (1) for a character array.
`isletter`	Returns true (1) for letters of the alphabet.
`isreal`	Returns true (1) if no element of array has an imaginary component.
`isstrprop`	Returns true (1) a character has the specified property.
`isspace`	Returns true (1) for whitespace.
`lower`	Convert string to lowercase.
`mat2str`	Convert matrix to string.
`mesh`	Create a mesh plot.
`meshgrid`	Create the (x, y) grid required for mesh, surface, and contour plots.
`nnz`	Number of nonzero matrix elements.
`nonzeros`	Return a column vector containing the nonzero elements in a matrix.
`num2str`	Convert number to string.
`nzmax`	Amount of storage allocated for nonzero matrix elements
`pie(x)`	Create a pie plot.
`plot(c)`	Plots the real versus the imaginary part of a complex array.
`real`	Returns the real portion of the complex number.
`rose`	Create a radial histogram of a data set.
`sscanf`	Read formatted data from string.
`stairs(x,y)`	Create a stair plot.
`stem(x,y)`	Create a stem plot.
`str2double`	Convert string to `double` value.
`str2num`	Convert string to number.
`strcat`	Concatenate strings.
`strcmp`	Returns true (1) if two strings are identical.

`strcmpi`	Returns true (1) if two strings are identical ignoring case.
`strjust`	Justify string.
`strncmp`	Returns true (1) if first `n` characters of two strings are identical.
`strncmpi`	Returns true (1) if first `n` characters of two strings are identical ignoring case.
`strmatch`	Find matches for string.
`strtrim`	Remove leading and trailing whitespace from a string.
`strrep`	Replace one string with another.
`strtok`	Find token in string.
`struct`	Predefine a structure array.
`strvcat`	Concatenate strings vertically.
`surf`	Create a surface plot.
`upper`	Convert string to uppercase.

6.8 Exercises

6.1 Figure 6.13 shows a series *RLC* circuit driven by a sinusoidal AC voltage source whose value is $120\angle 0^\circ$ volts. The impedance of the inductor in this circuit is $Z_L = j2\pi fL$, where j is $\sqrt{-1}$, f is the frequency of the voltage source in hertz, and L is the inductance in henrys. The impedance of the capacitor in this circuit is $Z_C = -j\frac{1}{2\pi fC}$, where C is the capacitance in farads. Assume that $R = 100\ \Omega$, $L = 0.1$ mH, and $C = 0.25$ nF.

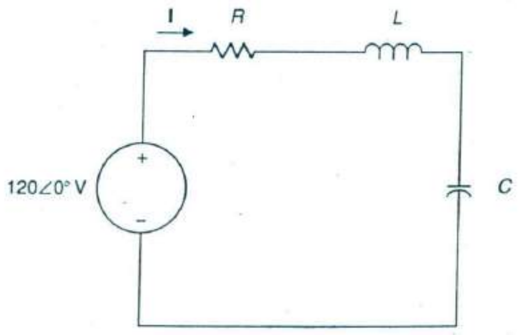


Figure 6.13 A series *RLC* circuit driven by a sinusoidal AC voltage source.

The current **I** flowing in this circuit is given by Kirchhoff's Voltage Law to be

$$\mathbf{I} = \frac{120\angle 0^\circ \text{ V}}{R + j2\pi fL - j\dfrac{1}{2\pi fC}} \tag{6-14}$$

(*a*) Calculate and plot the magnitude of this current as a function of frequency as the frequency changes from 100 kHz to 10 MHz. Plot this information on both a linear and a log-linear scale. Be sure to include a title and axis labels.

(*b*) Calculate and plot the phase angle in degrees of this current as a function of frequency as the frequency changes from 100 kHz to 10 MHz. Plot this information on both a linear and a log-linear scale. Be sure to include a title and axis labels.

(*c*) Plot both the magnitude and phase angle of the current as a function of frequency on two subplots of a single figure. Use log-linear scales.

6.2 Write a function `to_polar` that accepts a complex number `c` and returns two output arguments containing the magnitude `mag` and angle `theta` of the complex number. The output angle should be in degrees.

6.3 Write a function `to_complex` that accepts two input arguments containing the magnitude `mag` and angle `theta` of the complex number in degrees and returns the complex number `c`.

6.4 In a sinusoidal steady-state AC circuit, the voltage across a passive element is given by Ohm's Law:

$$\mathbf{V} = \mathbf{I}Z \tag{6-15}$$

where **V** is the voltage across the element, **I** is the current though the element, and Z is the impedance of the element. Note that all three of these values are complex and that these complex numbers are usually specified in the form of a magnitude at a specific phase angle expressed in degrees. For example, the voltage might be $\mathbf{V} = 120\angle 30^\circ$ V.

Write a program that reads the voltage across an element and the impedance of the element and calculates the resulting current flow. The input values should be given as magnitudes and angles expressed in degrees, and the resulting answer should be in the same form. Use the function `to_complex` from Exercise 6.3 to convert the numbers to rectangular for the actual computation of the current, and the function `to_polar` from Exercise 6.2 to convert the answer into polar form for display.

6.5 Write a function that will accept a complex number `c` and plot that point on a Cartesian coordinate system with a circular marker. The plot should include both the x and y axes, plus a vector drawn from the origin to the location of `c`.

6.6 Plot the function $v(t) = 10e^{(-0.2+j\pi)t}$ for $0 \le t \le 10$ using the function `plot(t,v)`. What is displayed on the plot?

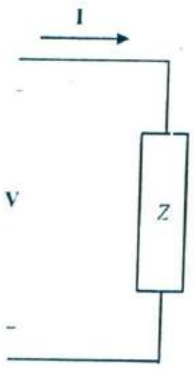


Figure 6.14 The voltage and current relationship on a passive AC circuit element.

6.7 Plot the function $v(t) = 10e^{(-0.2+j\pi)t}$ for $0 \le t \le 10$ using the function `plot(v)`. What is displayed on the plot this time?

6.8 Create a polar plot of the function $v(t) = 10e^{(-0.2+j\pi)t}$ for $0 \le t \le 10$.

6.9 Plot the function $v(t) = 10e^{(-0.2+j\pi)t}$ for $0 \le t \le 10$ using function `plot3`, where the three dimensions to plot are the real part of the function, the imaginary part of the function, and time.

6.10 **Euler's Equation** Euler's equation defines *e* raised to an imaginary power in terms of sinusoidal functions as follows:

$$e^{j\theta} = \cos\theta + i\sin\theta \tag{6-16}$$

Create a two-dimensional plot of this function as u varies from 0 to 2p. Create a three-dimensional line plot using function plot3 as u varies from 0 to 2p (the three dimensions are the real part of the expression, the imaginary part of the expression, and u).

6.11 Create a mesh, surface plot, and contour plot of the function $z = e^{x+iy}$ for the interval $-1 \le x \le 1$ and $-2\pi \le y \le 2\pi$. In each case, plot the real part of z versus x and y.

6.12 Write a program that accepts an input string from the user and determines how many times a user-specified character appears within the string. (*Hint:* Look up the `'s'` option of the `input` function using the MATLAB Help Browser.)

6.13 Modify the previous program so that it determines how many times a user-specified character appears within the string without regard to the case of the character.

6.14 Write a program that accepts a string from a user with the `input` function, chops that string into a series of tokens, sorts the tokens into ascending order, and prints them out.

6.15 Write a program that accepts a series of strings from a user with the `input` function, sorts the strings into ascending order, and prints them out.

6.16 Write a program that accepts a series of strings from a user with the `input` function, sorts the strings into ascending order disregarding case, and prints them out.

6.17 MATLAB includes functions `upper` and `lower`, which shift a string to uppercase and lowercase respectively. Create a new function called `caps`, which capitalizes the first letter in each word, and forces all other letters to be lowercase. (*Hint:* Take advantage of functions `upper`, `lower`, and `strtok`.)

6.18 Write a function that accepts a character string and returns a `logical` array with true values corresponding to each printable character that is *not* alphanumeric or whitespace (for example, $, %, #, etc.) and false values everywhere else.

6.19 Write a function that accepts a character string and returns a `logical` array with true values corresponding to each vowel and false values everywhere else. Be sure that the function works properly for both lowercase and uppercase characters.

6.20 Plot the function $y = e^{-x} \sin x$ for x between 0 and 2 in steps of 0.1. Create the following plot types: *(a)* stem plot; *(b)* stair plot; *(c)* bar plot; *(d)* compass plot. Be sure to include titles and axis labels on all plots.

6.21 Suppose that George, Sam, Betty, Charlie, and Suzie contributed $5, $10, $7, $5, and $15 respectively to a colleague's going-away present. Create a pie chart of their contributions. What percentage of the cost was paid by Sam?

6.22 Plot the function $f(x) = 1/\sqrt{x}$ over the range $0.1 \le x \le 10.0$ using function `fplot`. Be sure to label your plot properly.

CHAPTER 7

Advanced Features: Sparse Arrays, Cell Arrays, Structures, and Function Handles

This chapter deals with four very useful features of MATLAB: sparse arrays, cell arrays, structures, and function handles.

Sparse arrays are a special type of array in which memory is allocated only for the nonzero elements in the array. They provide an extremely useful and compact way to represent large arrays containing many zero values.

Cell arrays are a very flexible type of array that can hold any sort of data. Each element of a cell array can hold any type of MATLAB data, and different elements within the same array can hold different types of data. They are used extensively in MATLAB Graphical User Interface (GUI) functions.

Structures are a special type of array with named subcomponents. Each structure can have any number of subcomponents, each with its own name and data type. Structures are the basis of MATLAB objects.

Function handles provide an alternative way to access a function. They are more flexible than simple function names. Function handles make it easy to pass functions to other functions for processing; in addition, they make it easy to save data within a function between calls.

7.1 Sparse Arrays

We learned about ordinary MATLAB arrays in Chapter 2. When an ordinary array is declared, MATLAB creates a memory location for every element in the array. For example, the function `a = eye(10)` creates 100 elements arranged as a 10×10 structure. In this array, 90 of those elements are zero! This matrix requires 100 elements, but only 10 of them contain nonzero data. This is an example of a **sparse array** or **sparse matrix**. A sparse matrix is a large matrix in which the vast majority of the elements are zero.

```
» a = 2 * eye(10);
a =
     2     0     0     0     0     0     0     0     0     0
     0     2     0     0     0     0     0     0     0     0
     0     0     2     0     0     0     0     0     0     0
     0     0     0     2     0     0     0     0     0     0
     0     0     0     0     2     0     0     0     0     0
     0     0     0     0     0     2     0     0     0     0
     0     0     0     0     0     0     2     0     0     0
     0     0     0     0     0     0     0     2     0     0
     0     0     0     0     0     0     0     0     2     0
     0     0     0     0     0     0     0     0     0     2
```

Now suppose that we create another 10 × 10 matrix b defined as follows:

```
b =
     1     0     0     0     0     0     0     0     0     0
     0     2     0     0     0     0     0     0     0     0
     0     0     2     0     0     0     0     0     0     0
     0     0     0     1     0     0     0     0     0     0
     0     0     0     0     5     0     0     0     0     0
     0     0     0     0     0     1     0     0     0     0
     0     0     0     0     0     0     1     0     0     0
     0     0     0     0     0     0     0     1     0     0
     0     0     0     0     0     0     0     0     1     0
     0     0     0     0     0     0     0     0     0     1
```

If these two matrices are multiplied together, the result is

```
» c = a * b
c =
     2     0     0     0     0     0     0     0     0     0
     0     4     0     0     0     0     0     0     0     0
     0     0     4     0     0     0     0     0     0     0
     0     0     0     2     0     0     0     0     0     0
     0     0     0     0    10     0     0     0     0     0
     0     0     0     0     0     2     0     0     0     0
     0     0     0     0     0     0     2     0     0     0
     0     0     0     0     0     0     0     2     0     0
     0     0     0     0     0     0     0     0     2     0
     0     0     0     0     0     0     0     0     0     2
```

The process of multiplying these two sparse matrices together requires 1900 multiplications and additions; but because most of the terms being added and multiplied are zeros, it is largely wasted effort.

This problem gets worse rapidly as matrix size increases. For example, suppose that we were to generate two 200 × 200 sparse matrices a and b as follows:

```
a = 5 * eye(200);
b = 3 * eye(200);
```

Each matrix now contains 20,000 elements, of which 19,800 are zero! Furthermore, multiplying these two matrices together requires **7,980,000** additions and multiplications.

It should be apparent that storing and working with large sparse matrices, most of whose elements are zero, is a serious waste of both computer memory and CPU time. Unfortunately, many real-world problems naturally create sparse matrices, so we need some efficient way to solve problems involving them.

A large electric power system is an excellent example of a real-world problem involving sparse matrices. Large electric power systems can have a thousand or more electrical busses at generating plants and transmission and distribution substations. If we wish to know the voltages, currents, and power flows in the system, we must first solve for the voltage at every bus. For a 1000-bus system, this involves the simultaneous solution of 1000 equations in 1000 unknowns, which is equivalent to inverting a matrix with 1,000,000 elements. Solving this matrix requires millions of floating point operations.

However, each bus in the power system is probably connected to an average of only two or three other busses, so 996 of the 1000 terms in each row of the matrix will be zeros, and most of the operations involved in inverting the matrix will be additions and multiplications by zeros. The calculation of the voltages and currents in this power system would be much simpler and more efficient if the zeros could be ignored in the solution process.

The `sparse` Attribute

MATLAB has a special version of the `double` data type that is designed to work with sparse arrays. In this special version of the `double` data type, *only the nonzero elements of an array are allocated memory locations*, and the array is said to have the "sparse" attribute. An array with the sparse attribute actually saves three values for each nonzero element: the value of the element itself along with the row and column numbers where the element is located. Even though three values must be saved for each nonzero element, this approach is *much* more memory efficient than allocating full arrays if a matrix has only a few nonzero elements.

To illustrate the use of sparse matrices, we will create a 10 × 10 identity matrix:

```
» a = eye(10)
a =
     1     0     0     0     0     0     0     0     0     0
     0     1     0     0     0     0     0     0     0     0
     0     0     1     0     0     0     0     0     0     0
     0     0     0     1     0     0     0     0     0     0
     0     0     0     0     1     0     0     0     0     0
     0     0     0     0     0     1     0     0     0     0
     0     0     0     0     0     0     1     0     0     0
     0     0     0     0     0     0     0     1     0     0
     0     0     0     0     0     0     0     0     1     0
     0     0     0     0     0     0     0     0     0     1
```

If this matrix is converted to a sparse matrix using function `sparse`, the results are:

```
» as = sparse(a)
as =
    (1,1)        1
    (2,2)        1
    (3,3)        1
    (4,4)        1
    (5,5)        1
    (6,6)        1
    (7,7)        1
    (8,8)        1
    (9,9)        1
   (10,10)       1
```

Note that the data in the sparse matrix is a list of row and column addresses, followed by the nonzero data value at that point. This is a very efficient way to store data as long as most of the matrix is zero, but if there are many nonzero elements, it can take up even more space than the full matrix because of the need to store the addresses.

If we examine arrays `a` and `as` with the `whos` command, the results are:

```
» whos
  Name         Size      Bytes      Class
  a            10x10       800      double array
  as           10x10       164      double array (sparse)

Grand total is 110 elements using 964 bytes
```

The `a` array occupies 800 bytes, because there are 100 elements with 8 bytes of storage each. The `as` array occupies 164 bytes, because there are 10 nonzero elements with 8 bytes of storage each plus 20 array indices occupying 4 bytes each, and 4 bytes of overhead. Note that the sparse array occupies much less memory than the full array.

The function `issparse` can be used to determine whether or not a given array is sparse. If an array is sparse, then `issparse(array)` returns true (1).

The power of the sparse data type can be seen by considering a 1000 × 1000 matrix z with an average of 4 nonzero elements per row. If this matrix is stored as a full matrix, it will require 8,000,000 bytes of space. On the other hand, if it is converted to a sparse matrix, the memory usage will drop dramatically.

```
» zs = sparse(z);
» whos
  Name      Size            Bytes     Class
  z         1000x1000     8000000     double array
  zs        1000x1000       51188     sparse array

Grand total is 1003932 elements using 8051188 bytes
```

Generating Sparse Matrices

MATLAB can generate sparse matrices by converting a full matrix into a sparse matrix with the `sparse` function or by directly generating sparse matrices with the MATLAB functions `speye`, `sprand`, and `sprandn`, which are the sparse equivalents of `eye`, `rand`, and `randn`. For example, the expression `a = speye(4)` generates a 4 × 4 sparse matrix.

```
» a = speye(4)
a =
    (1,1)     1
    (2,2)     1
    (3,3)     1
    (4,4)     1
```

The expression `b = full(a)` converts the sparse matrix into a full matrix.

```
» b = full(a)
b =
     1     0     0     0
     0     1     0     0
     0     0     1     0
     0     0     0     1
```

Working with Sparse Matrices

Once a matrix is sparse, individual elements can be added to it or deleted from it using simple assignment statements. For example, the following statement generates a 4 × 4 sparse matrix, and then adds another nonzero element to it.

```
» a = speye(4)
a =
    (1,1)     1
    (2,2)     1
    (3,3)     1
    (4,4)     1
```

```
» a(2,1) = -2
a =
   (1,1)      1
   (2,1)     -2
   (2,2)      1
   (3,3)      1
   (4,4)      1
```

MATLAB allows full and sparse matrices to be freely mixed and used in any combination. The result of an operation between a full matrix and a sparse matrix may be either a full matrix or a sparse matrix depending on which result is the most efficient. Essentially any matrix technique that is supported for full matrices is also available for sparse matrices.

A few of the common sparse matrix functions are listed in Table 7.1.

Table 7.1 Common MATLAB Sparse Matrix Functions

Function	Description
	Create Sparse Matrices
`speye`	Create a sparse identity matrix.
`sprand`	Create a sparse uniformly-distributed random matrix.
`sprandn`	Create a sparse normally-distributed random matrix.
	Full-to-Sparse Conversion Functions
`sparse`	Convert a full matrix into a sparse matrix.
`full`	Convert a sparse matrix into a full matrix.
`find`	Find indices and values of nonzero elements in a matrix.
	Working with Sparse Matrices
`nnz`	Number of nonzero matrix elements.
`nonzeros`	Return a column vector containing the nonzero elements in a matrix.
`nzmax`	Amount of storage allocated for nonzero matrix elements.
`spones`	Replace nonzero sparse matrix elements with ones.
`spalloc`	Allocate space for a sparse matrix.
`issparse`	Returns 1 (true) for sparse matrix.
`spfun`	Apply function to nonzero matrix elements.
`spy`	Visualize sparsity pattern as a plot.

Example 7.1—Solving Simultaneous Equations with Sparse Matrices

To illustrate the ease with which sparse matrices can be used in MATLAB, we will solve the following simultaneous system of equations with both full and sparse matrices.

$$
\begin{aligned}
1.0x_1 + 0.0x_2 + 1.0x_3 + 0.0x_4 + 0.0x_5 + 2.0x_6 + 0.0x_7 - 1.0x_8 &= 3.0\\
0.0x_1 + 1.0x_2 + 0.0x_3 + 0.4x_4 + 0.0x_5 + 0.0x_6 + 0.0x_7 + 0.0x_8 &= 2.0\\
0.5x_1 + 0.0x_2 + 2.0x_3 + 0.0x_4 + 0.0x_5 + 0.0x_6 - 1.0x_7 + 0.0x_8 &= -1.5\\
0.0x_1 + 0.0x_2 + 0.0x_3 + 2.0x_4 + 0.0x_5 + 1.0x_6 + 0.0x_7 + 0.0x_8 &= 1.0\\
0.0x_1 + 0.0x_2 + 1.0x_3 + 1.0x_4 + 1.0x_5 + 0.0x_6 + 0.0x_7 + 0.0x_8 &= -2.0\\
0.0x_1 + 0.0x_2 + 0.0x_3 + 1.0x_4 + 0.0x_5 + 1.0x_6 + 0.0x_7 + 0.0x_8 &= 1.0\\
0.5x_1 + 0.0x_2 + 0.0x_3 + 0.0x_4 + 0.0x_5 + 0.0x_6 + 1.0x_7 + 0.0x_8 &= 1.0\\
0.0x_1 + 1.0x_2 + 0.0x_3 + 0.0x_4 + 0.0x_5 + 0.0x_6 + 0.0x_7 + 1.0x_8 &= 1.0
\end{aligned}
$$

SOLUTION To solve this problem, we will create full matrices of the equation coefficients and convert them to sparse form using the `sparse` function. Then we will solve the equation both ways, comparing the results and the memory required.

The script file to perform these calculations is shown below.

```
%  Script file: simul.m
%
%  Purpose:
%    This program solves a system of 8 linear equations in 8
%    unknowns (a*x = b), using both full and sparse matrices.
%
%  Record of revisions:
%      Date        Programmer          Description of change
%      ====        ==========          =====================
%    01/18/04    S. J. Chapman         Original code
%
% Define variables:
%   a             -- Coefficients of x (full matrix)
%   as            -- Coefficients of x (sparse matrix)
%   b             -- Constant coefficients (full matrix)
%   bs            -- Constant coefficients (sparse matrix)
%   x             -- Solution (full matrix)
%   xs            -- Solution (sparse matrix)

% Define coefficients of the equation a*x = b for
% the full matrix solution.
a = [ 1.0   0.0   1.0   0.0   0.0   2.0   0.0  -1.0; ...
      0.0   1.0   0.0   0.4   0.0   0.0   0.0   0.0; ...
      0.5   0.0   2.0   0.0   0.0   0.0  -1.0   0.0; ...
      0.0   0.0   0.0   2.0   0.0   1.0   0.0   0.0; ...
      0.0   0.0   1.0   1.0   1.0   0.0   0.0   0.0; ...
      0.0   0.0   0.0   1.0   0.0   1.0   0.0   0.0; ...
      0.5   0.0   0.0   0.0   0.0   0.0   1.0   0.0; ...
      0.0   1.0   0.0   0.0   0.0   0.0   0.0   1.0];

b = [ 3.0  2.0 -1.5  1.0 -2.0  1.0  1.0  1.0]';
```

```
% Define coefficients of the equation a*x = b for
% the sparse matrix solution:
as = sparse(a);
bs = sparse(b);

% Solve the system both ways
disp ('Full matrix solution:');
x = a\b

disp ('Sparse matrix solution:');
xs = as\bs

% Show workspace
disp('Workspace contents after the solutions:')
whos
```

When this program is executed, the results are:

```
» simul
Full matrix solution:
x =
    0.5000
    2.0000
   -0.5000
   -0.0000
   -1.5000
    1.0000
    0.7500
   -1.0000
Sparse matrix solution:
xs =
   (1,1)        0.5000
   (2,1)        2.0000
   (3,1)       -0.5000
   (5,1)       -1.5000
   (6,1)        1.0000
   (7,1)        0.7500
   (8,1)       -1.0000
Workspace contents after the solutions:
 Name     Size        Bytes      Class
  a        8x8         512       double array
  as       8x8         276       double array (sparse)
  b        8x1          64       double array
  bs       8x1         104       double array (sparse)
  x        8x1          64       double array
  xs       8x1          92       double array (sparse)
Grand total is 115 elements using 1112 bytes
```

The answers are the same for both solutions. Note that the sparse solution does not contain a solution for x_4, because that value is zero and zeros aren't carried in a sparse matrix! Also, note that the sparse form of matrix b actually takes up more space than the full form. This happens because the sparse representation must store the indices as well as the values in the arrays, so it is less efficient if most of the elements in an array are nonzero.

7.2 Cell Arrays

A **cell array** is a special MATLAB array whose elements are *cells*, containers that can hold other MATLAB arrays. For example, one cell of a cell array might contain an array of real numbers, another an array of strings, and yet another a vector of complex numbers (see Figure 7.1).

In programming terms, each element of a cell array is a *pointer* to another data structure, and those data structures can be of different types. Figure 7.2 illustrates this concept. Cell arrays are great ways to collect information about a problem, since all of the information can be kept together and accessed by a single name.

Cell arrays use braces {} instead of parentheses () for selecting and displaying the contents of cells. This difference is due to the fact that *cell arrays contain data structures instead of data*. Suppose that the cell array a is defined as shown

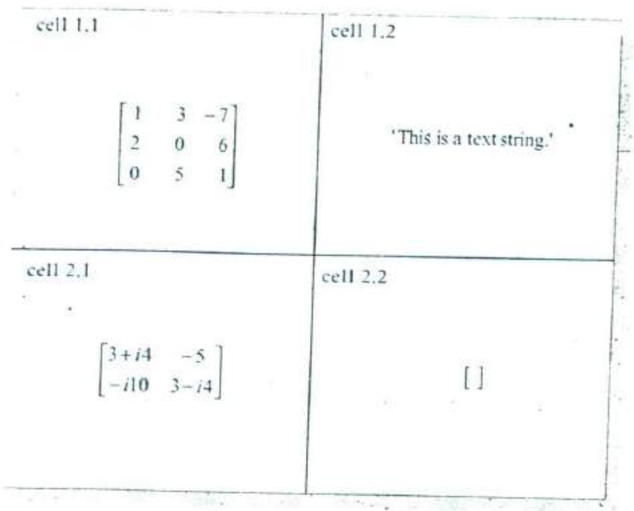
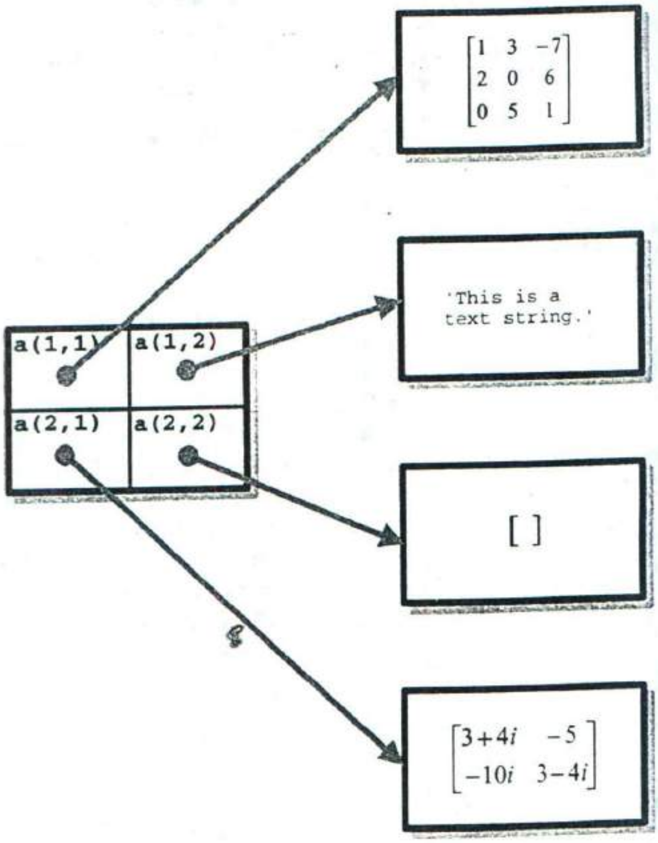

cell 1,1
$\begin{bmatrix} 1 & 3 & -7 \\ 2 & 0 & 6 \\ 0 & 5 & 1 \end{bmatrix}$
cell 1,2
'This is a text string.'
cell 2,1
$\begin{bmatrix} 3+i4 & -5 \\ -i10 & 3-i4 \end{bmatrix}$
cell 2,2
[ ]

Figure 7.1 The individual elements of a cell array may point to real arrays, complex arrays, string, other cell arrays, or even empty arrays.



a(1,1)
a(1,2)
a(2,1)
a(2,2)
$\begin{bmatrix}1 & 3 & -7\\ 2 & 0 & 6\\ 0 & 5 & 1\end{bmatrix}$
'This is a text string.'
[ ]
$\begin{bmatrix}3+4i & -5\\ -10i & 3-4i\end{bmatrix}$

Figure 7.2 Each element of a cell array holds a *pointer* to another data structure, and different cells in the same cell array can point to different types of data structures.

in Figure 7.2. Then the contents of element `a(1,1)` is a data structure containing a 3 × 3 array of numeric data, and a reference to `a(1,1)` displays the *contents* of the cell, which is the data structure.

```
» a(1,1)
ans =
    [3x3 double]
```

By contrast, a reference to `a{1,1}` displays *the contents of the contents of the cell.*

```
» a{1,1}
ans =
     1     3    -7
     2     0     6
     0     5     1
```

In summary, the notation `a(1,1)` refers to the contents of cell `a(1,1)` (which is a data structure), while the notation `a{1,1}` refers to the contents of the data structure within the cell.

Programming Pitfalls

Be careful not to confuse () with {} when addressing cell arrays. They are very different operations!

Creating Cell Arrays

Cell arrays can be created in the following two ways:

- By using assignment statements
- By preallocating a cell array using the `cell` function

The simplest way to create a cell array is to directly assign data to individual cells, one cell at a time. However, preallocating cell arrays is more efficient, so you should preallocate really large cell arrays.

Allocating Cell Arrays Using Assignment Statements

You can assign values to cell arrays one cell at a time using assignment statements. There are two ways to assign data to cells, known as **content indexing** and **cell indexing**.

Content indexing involves placing braces "{}" around the cell subscripts, together with cell contents in ordinary notation. For example, the following statement create the 2 × 2 cell array in Figure 7.2:

```
a{1,1} = [1 3 -7; 2 0 6; 0 5 1];
a{1,2} = 'This is a text string.';
a{2,1} = [3+4*i -5; -10*i 3 - 4*i];
a{2,2} = [];
```

This type of indexing defines the *contents of the data structure contained in a cell.*

Cell indexing involves placing braces "{}" around the data to be stored in a cell, together with cell subscripts in ordinary subscript notation. For example, the following statement create the 2 × 2 cell array in Figure 7.2:

```
a(1,1) = {[1 3 -7; 2 0 6; 0 5 1]};
a(1,2) = {'This is a text string.'};
a(2,1) = {[3+4*i -5; -10*i 3 - 4*i]};
a(2,2) = {[]};
```

This type of indexing *creates a data structure containing the specified data and then assigns that data structure to a cell.*

These two forms of indexing are completely equivalent, and they may be freely mixed in any program.

Programming Pitfalls

Do not attempt to create a cell array with the same name as an existing numeric array. If you do this, MATLAB will assume that you are trying to assign cell contents to an ordinary array, and it will generate an error message. Be sure to clear the numeric array before trying to create a cell array with the same name.

Preallocating Cell Arrays with the `cell` Function

The `cell` function allows you to preallocate empty cell arrays of the specified size. For example, the following statement creates an empty 2 × 2 cell array.

```
a = cell(2,2);
```

Once a cell array in created, you can use assignment statements to fill values in the cells.

Using Braces {} as Cell Constructors

It is possible to define many cells at once by placing all of the cell contents between a single set of braces. Individual cells on a row are separated by commas, and rows are separated by semicolons. For example, the following statement creates a 2 × 3 cell array:

```
b = {[1 2], 17, [2;4]; 3-4*i, 'Hello', eye(3)}
```

Viewing the Contents of Cell Arrays

MATLAB displays the data structures in each element of a cell array in a condensed form that limits each data structure to a single line. If the entire data structure can be displayed on the single line, it is. Otherwise, a summary is displayed. For example, cell arrays `a` and `b` would be displayed as:

```
» a
a =
    [3x3 double]     [1x22 char]
    [2x2 double]              []
» b
b =
           [1x2 double]   [    17] [2x1 double]
    [3.0000- 4.0000i]   'Hello'   3x3 double]
```

Note that MATLAB *is displaying the data structures*, complete with brackets or apostrophes, not the entire contents of the data structures.

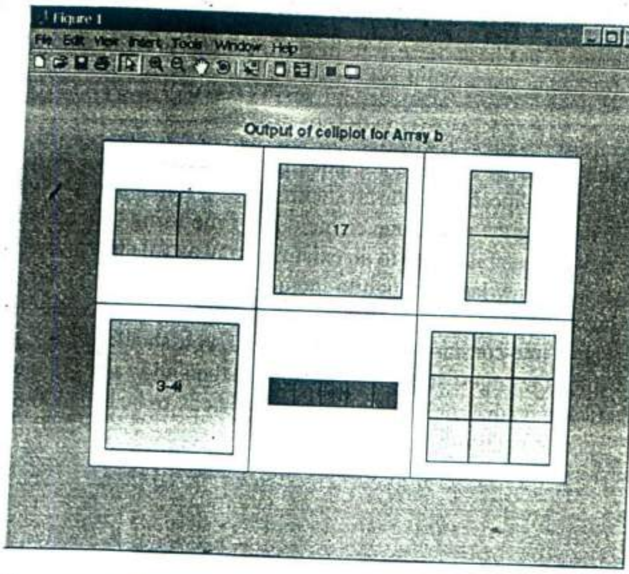


Figure 1
File Edit View Insert Tools Window Help
Output of cellplot for Array b
17
3-4i

Figure 7.3 The structure of cell array b is displayed as a nested series of boxes by function `cellplot`.

If you would like to see the full contents of a cell array, use the `celldisp` function. This function displays *the contents of the data structures in each cell.*

```
» celldisp(a)
a{1,1} =
      1    3   -7
      2    0    6
      0    5    1
a{2,1} =
   3.0000 + 4.0000i   -5.0000
        0 -10.0000i    3.0000 - 4.0000i
a{1,2} =
This is a text string.
a{2,2} =
     []
```

For a high-level graphical display of the structure of a cell array, use function `cellplot`. For example, the function `cellplot(b)` produces the plot shown in Figure 7.3.

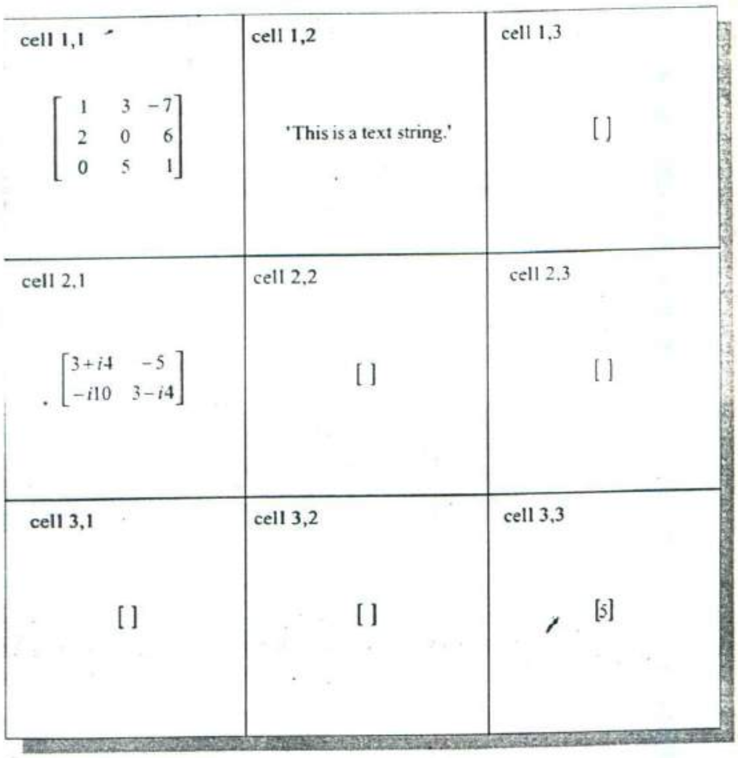
Extending Cell Arrays

If a value is assigned to a cell array element that does not currently exist, the element will be automatically created, and any additional cells necessary to preserve the shape of the array will be automatically created. For example, suppose that array `a` has been defined to be a 2 × 2 cell array as shown in Figure 7.1. If the following statement is executed

```
a{3,3} = 5
```

the cell array will be automatically extended to 3 × 3, as shown in Figure 7.4.

Preallocating cell arrays with the `cell` function is much more efficient than extending the arrays one element at a time using assignment statements. When a new element is added to an existing array as we did above, MATLAB must create a new array large enough to include this new element, copy the old data into the new array, add the new value to the array, and then delete the old array. This is a very time-consuming process. Instead, you should always allocate the cell array to be the largest size that you can, and then add values to it one element at a time. If you do that, only the new element needs to be added; the rest of the array can remain undisturbed.



cell 1,1
$\begin{bmatrix} 1 & 3 & -7 \\ 2 & 0 & 6 \\ 0 & 5 & 1 \end{bmatrix}$
cell 1,2
'This is a text string.'
cell 1,3
[ ]
cell 2,1
$\begin{bmatrix} 3+i4 & -5 \\ -i10 & 3-i4 \end{bmatrix}$
cell 2,2
[ ]
cell 2,3
[ ]
cell 3,1
[ ]
cell 3,2
[ ]
cell 3,3
[5]

Figure 7.4 The result of assigning a value to `a{3,3}`. Note that four other empty cells were created to preserve the shape of the cell array.

The program shown below illustrates the advantages of preallocation. It creates a cell array containing 50,000 strings added one at a time, with and without preallocation.

```
% Script file: test_preallocate.m
%
% Purpose:
%   This program tests the creation of cell arrays with
%   and without preallocation.
%
% Record of revisions:
%     Date          Programmer          Description of change
%     ====          ==========          =====================
%   01/18/04      S. J. Chapman         Original code
%
% Define variables:
%   a              -- Cell array
%   maxvals        -- Maximum values in cell array

% Create array without preallocation
clear all
maxvals = 50000;
tic
for ii = 1:maxvals
   a{ii} = ['Element ' int2str(ii)];
end
disp( ['Elapsed time without preallocation = ' num2str(toc)] );

% Create array with preallocation
clear all
maxvals = 50000;
tic
a = cell(1,maxvals);
for ii = 1:maxvals
   a{ii} = ['Element ' int2str(ii)];
end
disp( ['Elapsed time with preallocation = ' num2str(toc)] );
```

When this program is executed using MATLAB 7.0 on a 2.4 GHz Pentium IV computer, the results are as shown below. The advantages of preallocation are obvious.

```
» test_preallocate
Elapsed time without preallocation = 13.079
Elapsed time with preallocation    = 4.313
```

✱ Good Programming Practice

Always preallocate all cell arrays before assigning values to the elements of the array. This practice greatly increases the execution speed of a program.

Deleting Cells in Arrays

To delete an entire cell array, use the `clear` command. Subsets of cells may be deleted by assigning an empty array to them. For example, assume that `a` is the 3 × 3 cell array defined above.

```
» a
a =
    [3x3 double]    [1x22 char]    []
    [2x2 double]             []    []
              []             []   [5]
```

It is possible to delete the entire third row with the statement

```
» a(3,:) = []
a =
    [3x3 double]    [1x22 char]    []
    [2x2 double]             []    []
```

Using Data in Cell Arrays

The data stored inside the data structures within a cell array may be used at any time, with either content indexing or cell indexing. For example, suppose that a cell array `c` is defined as

```
c = {[1 2;3 4], 'dogs'; 'cats', i}
```

The contents of the array stored in cell `c(1,1)` can be accessed as follows

```
» c{1,1}
ans =
       1     2
       3     4
```

and the contents of the array in cell `c(2,1)` can be accessed as follows

```
» c{2,1}
ans =
cats
```

Subsets of a cell's contents can be obtained by concatenating the two sets of subscripts. For example, suppose that we would like to get the element (1, 2) from the array stored in cell `c(1,1)` of cell array `c`. To do this, we would use the expression `c{1,1}(1,2)`, which says: select element (1, 2) from the contents of the data structure contained in cell `c(1,1)`.

```
» c{1,1}(1,2)
ans =
     2
```

Cell Arrays of Strings

It is often convenient to store groups of strings in a cell array instead of storing them in rows of a standard character array, because each string in a cell array can have a different length, while every row of a cell array must have an identical length. This fact means that *strings in cell arrays do not have to be padded with blanks*. Many MATLAB Graphical User Interface functions use cell arrays for precisely this reason, as is shown in Chapter 10.

Cell arrays of strings can be created in one of two ways. Either the individual strings can be inserted into the array with brackets, or else function `cellstr` can be used to convert a 2-D string array into a cell array of strings.

The following example creates a cell array of strings by inserting the strings into the cell array one at a time; it then displays the resulting cell array. Note that the individual strings can be of different lengths.

```
» cellstring{1} = 'Stephen J. Chapman';
» cellstring{2} = 'Male';
» cellstring{3} = 'SSN 999-99-9999';
» cellstring
  'Stephen J. Chapman'    'Male'    'SSN 999-99-9999'
```

Function `cellstr` creates a cell array of strings from a 2-D string array. Consider the character array

```
» data = ['Line 1         ';'Additional Line']
data =
Line 1
Additional Line
```

This 2 × 15 character array can be converted into an cell array of strings with the function `cellstr` as follow:

```
» c = cellstr(data)
c =
   'Line 1'
   'Additional Line'
```

and it can be converted back to a standard character array using function `char`

```
» newdata = char(c)
newdata =
Line 1
Additional Line
```

The Significance of Cell Arrays

Cell arrays are extremely flexible, since any amount of any type of data can be stored in each cell. As a result, cell arrays are used in many internal MATLAB data structures. We must understand them in order to use many features of the MATLAB Graphical User Interface, which we will study in Chapter 10.

In addition, the flexibility of cell arrays makes them regular features of functions with variable numbers of input arguments and output arguments. A special input argument, `varargin`, is available within user-defined MATLAB functions to support variable numbers of input arguments. This argument appears as the last item in an input argument list, and it returns a cell array; therefore, *a single dummy input argument can support any number of actual arguments*. Each actual argument becomes one element of the cell array returned by `varargin`. If it is used, `varargin` must be the *last* input argument in a function, following all of the required input arguments.

For example, suppose that we are writing a function that may have any number of input arguments. This function could be implemented as shown:

```
function test1(varargin)
disp(['There are ' int2str(nargin) ' arguments.']);
disp('The input arguments are:');
disp(varargin);

end % function test1
```

When this function is executed with varying numbers of arguments, the results are:

```
» test1
There are 0 arguments.
The input arguments are:
» test1(6)
There are 1 arguments.
The input arguments are:
    [6]
» test1(1,'test 1',[1 2;3 4])
There are 3 arguments.
The input arguments are:
    [1]    'test 1'    [2x2 double]
```

As you can see, the arguments become a cell array within the function.

A sample function making use of variable numbers of arguments is shown below. Function `plotline` accepts an arbitrary number of 1 × 2 row vectors, with each vector containing the (x, y) position of one point to plot. The function plots a line connecting all of the (x, y) values together. Note that this function also accepts an optional line specification string and passes that specification on to the plot function.

```
function plotline(varargin)
%PLOTLINE Plot points specified by [x,y] pairs.
% Function PLOTLINE accepts an arbitrary number of
% [x,y] points and plots a line connecting them.
% In addition, it can accept a line specification
% string, and pass that string on to function plot.

% Define variables:
%  ii        -- Index variable
%  jj        -- Index variable
%  linespec  -- String defining plot characteristics
%  msg       -- Error message
%  varargin  -- Cell array containing input arguments
%  x         -- x values to plot
%  y         -- y values to plot

%  Record of revisions:
%     Date        Programmer        Description of change
%     ====        ==========        =====================
%   01/18/04    S. J. Chapman     Original code

% Check for a legal number of input arguments.
% We need at least 2 points to plot a line...
msg = nargchk(2,Inf,nargin);
error(msg);

% Initialize values
jj = 0;
linespec = '';

% Get the x and y values, making sure to save the line
% specification string, if one exists.
for ii = 1:nargin

  % Is this argument an [x,y] pair or the line
  % specification?
  if ischar(varargin{ii})

     % Save line specification
     linespec = varargin{ii};

  else

     % This is an [x,y] pair. Recover the values.
     jj = jj + 1;
```

```
         x(jj) = varargin{ii}(1);
         y(jj) = varargin{ii}(2);
      end
   end

   % Plot function.
   if isempty(linespec)
      plot(x,y);
   else
      plot(x,y,linespec);
   end

   end % function plotline
```

When this function is called with the arguments shown below, the plot shown in Figure 7.5 is the result. Try the function with different numbers of arguments and see for yourself how it behaves.

```
   plotline([0 0],[1 1],[2 4],[3 9],'k--');
```

There is also a special output argument, `varargout`, to support variable numbers of output arguments. This argument appears as the last item in an output

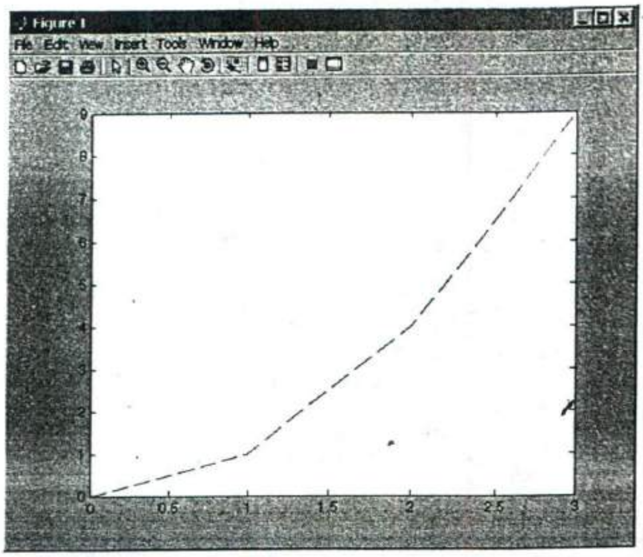


Figure 7.5 The plot produced by function `plotline`.

argument list, and it returns a cell array; therefore, *a single dummy output argument can support any number of actual arguments*. Each actual argument becomes one element of the cell array stored in `varargout`.

If it is used, `varargout` must be the *last* output argument in a function, following all of the required input arguments. The number of values to be stored in `varargout` can be determined from function `nargout`, which specifies the number of actual output arguments for any given function call.

A sample function `test2` is shown below. This function detects the number of output arguments expected by the calling program, using the function `nargout`. It returns the number of random values in the first output argument and then fills the remaining output arguments with random numbers taken from a Gaussian distribution. Note that the function uses `varargout` to hold the random numbers; consequently, there can be an arbitrary number of output values.

```
function [nvals,varargout] = test2(mult)
% nvals is the number of random values returned
% varargout contains the random values returned
nvals = nargout - 1;
for ii = 1:nargout-1
    varargout{ii} = randn * mult;
end

end % function test2
```

When this function is executed, it produces the results shown below.

```
» test2(4)
ans =
    -1
» [a b c d] = test2(4)
a =
     3
b =
   -1.7303
c =
   -6.6623
d =
    0.5013
```

Good Programming Practice

Use cell array arguments `varargin` and `varargout` to create functions that support varying numbers of input and output arguments.

Table 7.2 **Common MATLAB Cell Functions**

Function	Description
`cell`	Predefine a cell array structure.
`celldisp`	Display contents of a cell array.
`cellplot`	Plot structure of a cell array.
`cellstr`	Convert a 2-D character array to a cell array of strings.
`char`	Convert a cell array of strings to a 2-D character array.

Summary of `cell` Functions

The common MATLAB cell functions are summarized in Table 7.2.

7.3 Structure Arrays

An *array* is a data type in which there is a name for the whole data structure, but individual elements within the array are known only by number. Thus, the fifth element in the array named `arr` would be accessed as `arr(5)`. All of the individual elements in an array must be of the *same* type.

A *cell array* is a data type in which there is a name for the whole data structure, but individual elements within the array are known only by number. However, the individual elements in the cell array may be of *different* types.

In contrast, a **structure** is a data type in which each individual element is has a name. The individual elements of a structure are known as **fields**, and each field in a structure may have a different type. The individual fields are addressed by combining the name of the structure with the name of the field. separated by a period.

Figure 7.6 shows a sample structure named `student`. This structure has five fields, called `name`, `addr1`, `city`, `state`, and `zip`. The field called "`name`" would be addressed as `student.name`.

A **structure array** is an array of structures. Each structure in the array will have identically the same fields, but the data stored in each field can differ. For example, a class could be described by an array of the structure `student`. The first student's name would be addressed as `student(1).name`, the second student's city would be addressed as `student(2).city`, and so forth.

Creating Structure Arrays

Structure arrays can be created in the following two ways:

- A field at a time using assignment statements
- All at once using the `struct` function

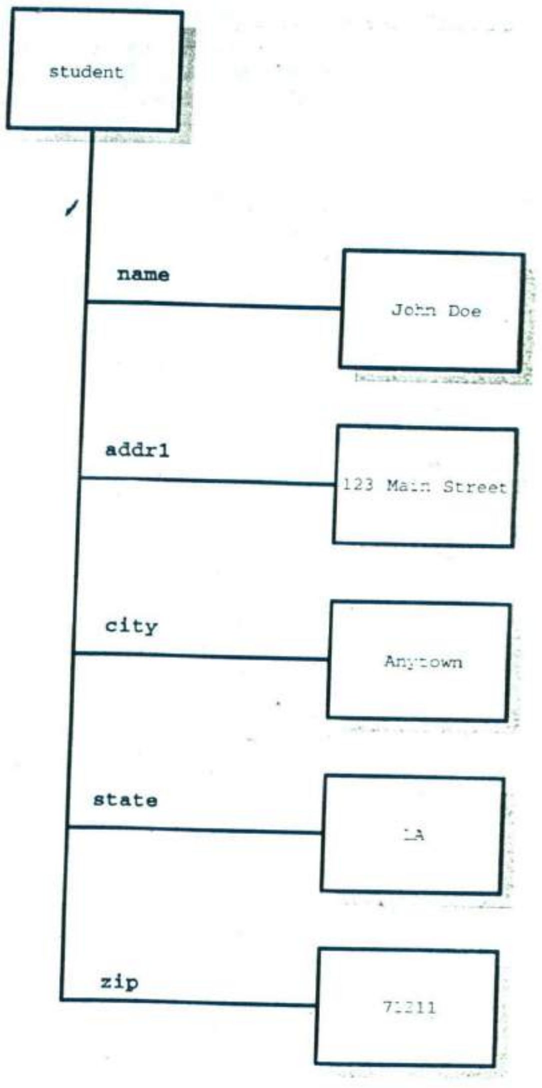


Figure 7.6 A sample structure. Each element within the structure is called a field, and each field is addressed by name.

Building a Structure with Assignment Statements

You can build a structure one field at a time using assignment statements. Each time that data is assigned to a field, that field is automatically created. For example, the structure shown in Figure 7.6 can be created with the following statements:

```
» student.name='John Doe';
» student.addr1='123 Main Street';
» student.city ='Anytown';
» student.zip='71211'
student =
     name:  'John Doe'
    addr1:  '123 Main Street'
     city:  'Anytown'
    state:  'LA'
      zip:  '71211'
```

A second student can be added to the structure by adding a subscript to the structure name (*before* the period).

```
» student(2).name = 'Jane Q. Public'
student =
1x2 struct array with fields:
    name
    addr1
    city
    state
    zip
```

student is now a 1 × 2 array. Note that when a structure array has more than one element, only the field names are listed, not their contents. The contents of each element can be listed by typing the element separately in the Command Window:

```
» student(1)
ans =
      name: 'John Doe'
     addr1: '123 Main Street'
      city: 'Anytown'
     state: 'LA'
       zip: '71211'
» student(2)
ans =
      name:  'Jane Q. Public'
     addr1:  []
      city:  []
     state:  []
       zip:  []
```

Note that *all of the fields of a structure are created for each array element whenever that element is defined*, even if they are not initialized. The uninitialized fields will contain empty arrays, which can be initialized with assignment statements at a later time.

The field names used in a structure can be recovered at any time using the `fieldnames` function. This function returns a list of the field names in a cell array of strings, and is very useful for working with structure arrays within a program.

Creating Structures with the `struct` Function

The `struct` function allows you to preallocate a structure or an array of structures/The basic form of this function is

```
str_array = struct('field1',val1,'field2',val2, ...)
```

where the arguments are field names and their initial values. With this syntax function `struct` initializes every field to the specified value.

To preallocate an entire array with the `struct` function to the *last value* the array. All of the values before that will be automatically created at the same time. For example, the statements shown below create an array containing 1000 sturctures of type `student`.

```
student(1000) = struct('name',[].'addr1',[], ...
                       'city',[],'state',[],'zip',[])
student =
1x1000 struct array with fields:
    name
    addr1
    city
    state
    zip
```

All of the elements of the structure are preallocated, which will speed up any program using the structure.

There is another version of the `struct` function that will preallocate an array and at the same time assign initial values to all of its field. You will be asked to do this in an end of chapter exercise.

Adding Fields to Structures

If a new field name is defined for any element is a structure array, the field is automatically added to all of the elements in the array. For example, suppose that we add some exam scores to Jane Public's record:

```
» student(2).exams = [90 82 88]
student =
1x2 struct array with fields:
    name
    addr1
    city
    state
    zip
    exams
```

There is now a field called exams in every record of the array, as shown below. This field will be initialized for `student(2)`, and will be an empty array for all other students until appropriate assignment statements are issued.

```
» student(1)
ans =
         name: 'John Doe'
        addr1: '123 Main Street'
         city: 'Anytown'
        state: 'LA'
          zip: '71211'
        exams: []
» student(2)
ans =
         name: 'Jane Q. Public'
        addr1: []
         city: []
        state: []
          zip: []
        exams: [90 82 88]
```

Removing Fields from Structures

A field may be removed from a structure array using the `rmfield` function. The form of this function is:

```
struct2 = rmfield(str_array,'field')
```

where `str_array` is a structure array, `'field'` is the field to remove, and `struct2` is the name of a new structure with that field removed. For example, we can remove the field `'zip'` from structure array `student` with the following statement:

```
» stu2 = rmfield(student,'zip')
stu2 =
1x2 struct array with fields:
    name
    addr1
    city
    state
    exams
```

Using Data in Structure Arrays

Now let's assume that structure array student has been extended to include three students and all data has been filled in, as shown in Figure 7.7. How do we use the data in this structure array?

To access the information in any field of any array element, just name the array element followed by a period and the field name:

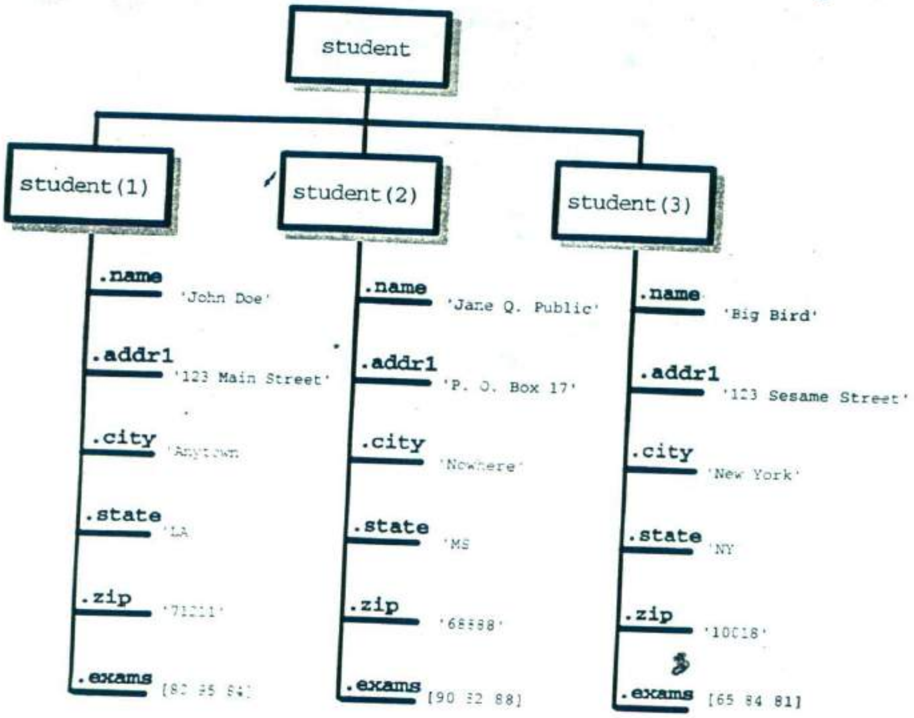


Figure 7.7 The student array with three elements and all fields filled in.

```
» student(2).addr1
ans =
P. O. Box 17
» student(3).exams
ans =
    65    84    81
```

To access an individual item within a field, add a subscript after the field name. For example, the second exam of the third student is

```
» student(3).exams(2)
ans =
    84
```

The fields in a structure array can be used as arguments in any function that supports that type of data. For example, to calculate `student(2)`'s exam average, we could use the function

```
» mean(student(2).exams)
ans =
    86.6667
```

Unfortunately, we can *not* extract the values from a given field across multiple array elements at the same time. For example, we cannot get access to an array of zip codes with the expression `student.zip`. That expression returns the three zip codes of the three students in three separate arrays. If we want to get the zip codes of all of the students in a single array, we must use a `for` loop:

```
for ii = 1:length(student)
   zip(ii) = student(ii).zip;
end
```

Similarly, if we wanted to get the average of *all* exams from *all* students, we cannot use the function `mean(student.exams)`. Instead, we must build up an array containing all the exam scores by accessing each student's exams separately and then call `mean` with that array.

```
exam_list = [];
for ii = 1:length(student)
   exam_list = [exam_list student(ii).exams];
end
mean(exam_list)
```

The `getfield` and `setfield` Functions

Two MATLAB functions are available to make structure arrays easier to use in programs. Function `getfield` gets the current value stored in a field and function `setfield` inserts a new value into a field. The structure of function `getfield` is

```
f = getfield(array,{array_index},'field',{field_index})
```

where the `field_index` is optional and `array_index` is optional for a 1-by-1 structure array. The function call corresponds to the statement

```
f = array(array_index).field(field_index);
```

but it can be used even if the programmer doesn't know the names of the fields in the structure array at the time the program is written.

For example, suppose that we needed to write a function to read and manipulate the data in an unknown structure array. This function could determine the field names in the structure using a call to `fieldnames`, and could then read the data using function `getfield`. To read the zip code of the second student, the function would be

```
» zip = getfield(student,{2},'zip')
zip =
      68888
```

Similarly, a program could modify values in the structure using function `setfield`. The structure of function `setfield` is

```
f = setfield(array,{array_index},'field',{field_index},value)
```

where `f` is the output structure array, the `field_index` is optional, and `array_index` is optional for a 1-by-1 structure array. The function call corresponds to the statement

```
array(array_index).field(field_index) = value;
```

Dynamic Field Names

Beginning with MATLAB 7.0, there is an alternative way to access the elements of a structure: **dynamic field names**. A dynamic field name is a string enclosed in parentheses at a location where a field name is expected. For example, the name of student 1 can be retrieved with either static or dynamic field names as shown below:

```
» student(1).name                   % Static field name
ans =
John Doe
» student(1).('name')               % Dynamic field name
ans =
John Doe
```

Dynamic field names perform the same function as static field names, but *dynamic field names can be changed during program execution*. This allows a user to access different information in the same function within a program.

For example, the following function accepts a structure array and a field name and calculates the average of the values in the specified field for all elements in the structure array. It returns that average (and optionally the number of values averaged) to the calling program.

```
function [ave, nvals] = calc_average(structure,field)
%CALC_AVERAGE Calculate the average of values in a field.
% Function CALC_AVERAGE calculates the average value
% of the elements in a particular field of a structure
% array. It returns the average value and (optionally)
% the number of items averaged.

% Define variables:
%   arr        -- Array of values to average
%   ave        -- Average of arr
%   ii         -- Index variable
%
```

```
% Record of revisions:
%     Date        Programmer        Description of change
%     ====        ==========        =====================
%   01/18/04    S. J. Chapman     Original code
%
% Check for a legal number of input arguments.
msg = nargchk(2,2,nargin);
error(msg);

% Create an array of values from the field
arr = [];
for ii = 1:length(structure)
   arr = [arr structure(ii).(field)];
end

% Calculate average
ave = mean(arr);

% Return number of values averaged
if nargout == 2
   nvals = length(arr);
end

end % function calc_average
```

A program can average the values in different fields by simply calling this function multiple times with different structure names and different field names. For example, we can calculate the average values in fields exams and zip as follows:

```
» [ave,nvals] = calc_average(student,'exams')
ave =
   83.2222
nvals =
     9
» ave = calc_average(student,'zip')
ave =
       50039
```

Using the size Function with Structure Arrays

When the size function is used with a structure array, it returns the size of the structure array itself. When the size function is used with a *field* from a particular element in a structure array, it returns the size of that field instead of the size of the whole array. For example,

```
» size(student)
ans =
      1     3
» size(student(1).name)
ans =
      1     8
```

Nesting Structure Arrays

Each field of a structure array can be of any data type, including a cell array or a structure array. For example, the following statements define a new structure array as a field under array `student` to carry information about each class that the student in enrolled in.

```
student(1).class(1).name = 'COSC 2021'
student(1).class(2).name = 'PHYS 1001'
student(1).class(1).instructor = 'Mr. Jones'
student(1).class(2).instructor = 'Mrs. Smith'
```

After these statements are issued, `student(1)` contains the following data. Note the technique used to access the data in the nested structures.

```
» student(1)
ans =
        name:  'John Doe'
       addr1:  '123 Main Street'
        city:  'Anytown'
       state:  'LA'
         zip:  '71211'
       exams:  [80 95 84]
       class:  [1x2 struct]
» student(1).class
ans =
1x2 struct array with fields:
    name
    instructor
» student(1).class(1)
ans =
          name:  'COSC 2021'
    instructor:  'Mr. Jones'
» student(1).class(2)
ans =
          name:  'PHYS 1001'
    instructor:  'Mrs. Smith'
» student(1).class(2).name
ans =
PHYS 1001
```

Table 7.3 Common MATLAB Structure Functions

Function	**Description**
`fieldnames`	Return a list of field names in a cell array of strings.
`getfield`	Get current value from a field.
`rmfield`	Remove a field from a structure array.
`setfield`	Set new value into a field.
`struct`	Pre-define a structure array.

Summary of `structure` Functions

The common MATLAB structure functions are summarized in Table 7.3 on the above.

7.4 Function Handles

A **function handle** is a MATLAB data type that holds information to be used in referencing a function. When you create a function handle, MATLAB captures all the information about the function that it needs to execute it later on. Once the handle is created, it can be used to execute the function at any time.

As is shown in Chapter 10, function handles are key to the operation of MATLAB graphical user interfaces. We will learn about them here, and apply that knowledge in Chapter 10.

Creating and Using Function Handles

A function handle can be created either of two possible way: the `@` operator or the `str2func` function. To create a function handle with the `@` operator, just place it in front of the function name. To create a function handle with the `str2func` function, call the function with the function name in a string. For example, suppose that function `my_func` is defined as follows:

```
function res = my_func(x)
res = x.^2 - 2*x + 1;
end % function my_func
```

Then either of the following lines will create a function handle for function `my_func`:

```
hndl = @my_func
hndl = str2func('my_func');
```

Once a function handle has been created, the function can be executed by naming the function handle followed by any calling parameters. The result will be exactly the same as if the function itself were named.

```
» hndl = @my_func
hndl =
    @my_func
» hndl(4)
ans =
     9
» my_func(4)
ans =
     9
```

If a function has no calling parameters, then the function handle must be followed by empty parentheses when it is used to call the function:

```
» h1 = @randn;
» h1()
ans =
   -0.4326
```

After a function handle is created, it appears in the current workspace with the data type "function handle":

```
» whos
Name      Size      Bytes     Class
ans       1x1           8     double array
h1        1x1          16     function_handle array
hndl      1x1          16     function_handle array

Grand total is 3 elements using 40 bytes
```

A function handle can also be executed using the `feval` function. This provides a convenient way to execute function handles within a MATLAB program.

```
» feval(hndl,4)
ans =
     9
```

It is possible to recover the function name from a function handle using the `func2str` function.

```
» func2str(hndl)
ans =
my_func
```

This feature is very useful when we want to create descriptive messages, error messages, or labels inside a function that accepts and evaluates function handles. For example, the function shown below accepts a function handle in the first argument, and plots the function at the points specified in the second argument. It also prints out a title containing the name of the function being plotted.

```
function plotfunc(fun,points)
% PLOTFUNC Plots a function between the specified points.
% Function PLOTFUNC accepts a function handle, and
% plots the function at the points specified.

% Define variables:
%   fun        -- Function handle
%   msg        -- Error message
%
% Record of revisions:
%     Date         Programmer            Description of change
%     ====         ==========            =====================
%   01/21/04     S. J. Chapman           Original code

% Check for a legal number of input arguments.
msg = nargchk(2,2,nargin);
error(msg);

% Get function name
fname = func2str(fun);

% Plot the data and label the plot
plot(points,fun(points));
title(['\bfPlot of ' fname '(x) vs x']);
xlabel('\bfx');
ylabel(['\bf' fname '(x)']);
grid on;

end % function plotfunc
```

For example, this function can be used to plot the function sin x from -2π to 2π with the following statement:

```
plotfunc(@sin, [-2*pi:pi/10:2*pi])
```

The resulting function is shown in Figure 7.8.

Some common MATLAB functions used with function handles are summarized in Table 7.4.

The Significance of Function Handles

Either function names or function handles can be used to execute most functions. However, function handles have certain advantages over function names. These advantages include:

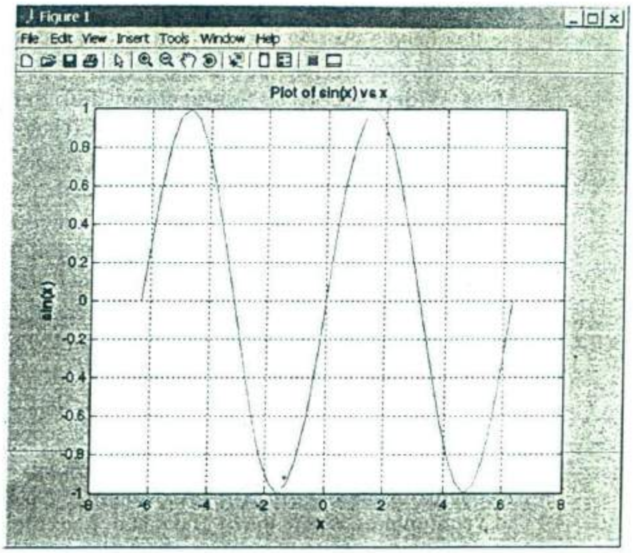


Figure 7.8 Plot of function sin x from -2π to 2π, created using function `plotfunc`.

1. **Passing Function Access Information to Other Functions.** As we saw in the previous section, you can pass a function handle as an argument in a call to another function. The function handle enables the receiving function to call the function attached to the handle. You can execute a function handle from within another function *even if the handle's function is not in the scope of the evaluating function*. This is because the function handle has a complete description of the function to execute; the calling function does not have to search for it.
2. **Improved Performance in Repeated Operation.** MATLAB performs a search for a function at the time you create a function handle and then

Table 7.4 MATLAB Functions that Manipulate Function Handles

Function	Description
`@`	Create a function handle.
`feval`	Evaluate a function using a function handle.
`func2str`	Recover the function name associated with a given function handle.
`functions`	Recover miscellaneous information from a function handle. The data is returned in a structure.
`str2func`	Create a function handle from a specified string.

stores this access information in the handle itself. Once defined, you can use this handle over and over without having to look it up again. This makes function execution faster.

3. **Allow Wider Access to Subfunctions and Private Functions.** All MATLAB functions have a certain scope. They are visible to other MATLAB entities within that scope but not visible outside of it. You can call a function directly from another function that is within its scope, but *not* from a function outside that scope. Subfunctions, private functions, and nested functions are limited in their visibility to other MATLAB functions. You can invoke a subfunction only from another function that is defined within the same M-file. You can invoke a private function only from a function in the directory immediately above the `private` subdirectory. You can invoke a nested function only from within the host function or another nested function at the same level. However, when you create a handle to a function that has limited scope, the function handle stores all the information MATLAB needs to evaluate the function from *any* location in the MATLAB environment. If you create a handle to a subfunction within the M-file that defines the subfunction, you can then pass the handle to code that resides outside of that M-file and evaluate the subfunction from beyond its usual scope. The same holds true for private functions and nested functions.
4. **Include More Functions per M-File for Easier File Management.** You can use function handles to help reduce the number of M-files required to contain your functions. The problem with grouping a number of functions in one M-file has been that this defines them as subfunctions, and thus reduces their scope in MATLAB. Using function handles to access these subfunctions removes this limitation. This enables you to group functions as you want and reduce the number of files you have to manage.

Function Handles and Nested Functions

When MATLAB invokes an ordinary function, a special workspace is created to contain the function's variables. The function executes to completion, after which the workspace is destroyed. All the data in the function workspace is lost, except for any values labeled `persistent`. If the function is executed again, a completely new workspace is created for the new execution.

By contrast, when a host function creates a handle for a nested function and returns that handle to a calling program, the host function's workspace is created and *remains in existence for as long as the function handle remains in existence*. Since the nested function has access to the host function's variables, MATLAB has to preserve the host's function's data as long as there is any chance that the nested function will be used. This means that *we can save data in a function between uses*.

This idea is illustrated in the function shown below. When function `count_calls` is executed, it initializes a local variable `current_count` to a user-specified initial count and then creates and returns a handle to the nested function `increment_count`. When `increment_count` is called using that function handle, the count is increased by one and the new value is returned.

```
function fhandle = count_calls(initial_value)

% Save initial value in a local variable
% in the host function.
current_count = initial_value;

% Create and return a function handle to the
% nested function below.
fhandle = @increment_count;

   % Define a nested function to increment counter
   function count = increment_count
   current_count = current_count + 1;
   count = current_count;
   end % function increment_count

end % function count_calls
```

When this program is executed, the results are as shown below. Each call to the function handle increments the count by one.

```
» fh = count_calls(4);
» fh()
ans =
       5
» fh()
ans =
       6
» fh()
ans =
       7
```

Even more importantly, *each function handle created for a function has its own independent workspace*. If we create two different handles for this function, each one will have its own local data and they will be independent of each other. As you can see, we can increment either counter independently by calling the function with the proper handle.

```
» fh1 = count_calls(4);
» fh2 = count_calls(20);
» fh1()
ans =
       5
» fh1()
ans =
       6
```

```
» fh2()
ans =
      21
» fh1()
ans =
       7
```

You can use this feature to run multiple counters and so forth within a program without them interfering with each other.

Quiz 7.1

This quiz provides a quick check to see if you have understood the concepts introduced in Sections 7.1 through 7.4. If you have trouble with the quiz, reread the section, ask your instructor, or discuss the material with a fellow student. The answers to this quiz are found in the back of the book.

1. What is a sparse array? How does it differ from a full array? How can you convert from a sparse array to a full array and vice versa?
2. What is a cell array? How does it differ from an ordinary array?
3. What is the difference between content indexing and cell indexing?
4. What is a structure? How does it differ from ordinary arrays and cell arrays?
5. What is the purpose of `varargin`? How does it work?
6. What is a function handle? How do you create a function handle? How do you call a function using a function handle?
7. Given the definition of array *a* shown below, what will be produced by each of the following sets of statements? (*Note:* some of these statements may be illegal. If a statement is illegal, explain why.)

```
    a{1,1} = [1 2 3; 4 5 6; 7 8 9];
    a(1,2) = {'Comment line'};
    a{2,1} = j;
    a{2,2} = a{1,1} - a{1,1}(2,2);
```

(*a*) `a(1,1)`
(*b*) `a{1,1}`
(*c*) `2*a(1,1)`
(*d*) `2*a{1,1}`
(*e*) `a{2,2}`
(*f*) `a(2,3) = {[-17; 17]}`
(*g*) `a{2,2}(2,2)`

8. Given the definition of structure array `b` shown below, what will be produced by each of the following sets of statements? (*Note:* some

of these statements may be illegal. If a statement is illegal, explain why.)

```
b(1).a = -2*eye(3);
b(1).b = 'Element 1';
b(1).c = [1 2 3];
b(2).a = [b(1).c' [-1; -2; -3] b(1).c'];
b(2).b = 'Element 2';
b(2).c = [1 0 -1];
```

(*a*) `b(1).a - b(2).a`
(*b*) `strncmp(b(1).b,b(2).b,6)`
(*c*) `mean(b(1).c)`
(*d*) `mean(b.c)`
(*e*) `b`
(*f*) `b(1).('b')`
(*g*) `b(1)`

9. What will be returned by the following function, if it is called with the expression `myfun(@cosh)`?

```
function res = myfun(x)
res = func2str(x);
end % function myfun
```

7.5 Summary

Sparse arrays are special arrays in which memory is allocated only for nonzero elements. Three values are saved for each nonzero element—a row number, a column number, and the value itself. This form of storage is much more efficient than for arrays for the situation where only a tiny fraction of the elements are nonzero. MATLAB includes functions and intrinsic calculations for sparse arrays, so they can be freely and transparently mixed with full arrays.

Cell arrays are arrays whose elements are *cells*, containers that can hold other MATLAB arrays. Any sort of data may be stored in a cell, including structure arrays and other cell arrays. They provide a very flexible way to store data and are used in many internal MATLAB graphical user interface functions.

Structure arrays are a data type in which each individual element is given a name. The individual elements of a structure are known as fields, and each field in a structure may have a different type. The individual fields are addressed by combining the name of the structure with the name of the field, separated by a period. Structure arrays are useful for grouping together all of the data related to a particular person or thing into a single location.

Function handles are a special data type containing all the information required to invoke a function. Function handles are created with the @ operator or the `str2func` function and are used by naming the handle following by parentheses and the required calling arguments. If a function handle is created for a nested function, the workspace of the host function will be preserved between calls to the nested function using the function handle.

`@`	Create a function handle.
`cell`	Predefine a cell array structure.
`celldisp`	Display contents of a cell array.
`cellplot`	Plot structure of a cell array.
`cellstr`	Convert a 2-D character array to a cell array of strings.
`feval`	Evaluate a function using a function handle.
`fieldnames`	Return a list of field names in a cell array of strings.
`func2str`	Get the name of the function pointed to by the specified function handle.
`functions`	Recover miscellaneous information from a function handle in a structure.
`getfield`	Get current value from a field.
`full`	Convert a sparse matrix into a full matrix
`nnz`	Number of nonzero matrix elements.
`nonzeros`	Return a column vector containing the nonzero elements in a matrix.
`nzmax`	Amount of storage allocated for nonzero matrix elements.
`rmfield`	Remove a field from a structure array.
`setfield`	Set new value into a field.
`spalloc`	Allocate space for a sparse matrix.
`sparse`	Convert a full matrix into a sparse matrix.
`speye`	Create a sparse identity matrix.
`spfun`	Apply function to nonzero matrix elements.
`spones`	Replace nonzero sparse matrix elements with ones
`sprand`	Create a sparse uniformly–distributed random matrix.
`sprandn`	Create a sparse normally–distributed random matrix.
`sprintf`	Write formatted data to string.
`spy`	Visualize sparsity pattern as a plot
`str2func`	Create a function handle for the function named in a string argument.
`struct`	Preallocate a structure array

Summary of Good Programming Practice

The following guidelines should be adhered to:

1. Always preallocate all cell arrays before assigning values to the elements of the array. This practice greatly increases the execution speed of a program.
2. Use cell array arguments `varargin` and `varargout` to create functions that support varying numbers of input and output arguments.

MATLAB Summary

The summary (see page 354) lists all of the MATLAB commands and functions described in this chapter, along with a brief description of each one.

7.6 Exercises

7.1 Write a MATLAB function that will accept a cell array of strings and sort them into ascending order according to the lexicographic order of the ASCII character set. (You may use function `c_strcmp` from Chapter 6 for the comparisons if you wish.)

7.2 Write a MATLAB function that will accept a cell array of strings and sort them into ascending order according to *alphabetical order*. (This implies that you must treat 'A' and 'a' as the same letter.)

7.3 Create a sparse 100 × 100 array `a` in which about 5% of the elements contain normally distributed random values and all of the other elements are zero (use function `sprandn` to generate these values). Next, set all of the diagonal elements in the array to 1. Next, define a 100-element sparse column array `b` and initialize that array with 100 uniformly distributed values produced by function `rand`. Answer the following questions about these arrays:

(a) Create a full array `a_full` from the sparse array `a`. Compare the memory required to store the full array and the sparse array. Which is more efficient?
(b) Plot the distribution of values in array `a` using function `spy`.
(c) Create a full array `b_full` from the sparse array `b`. Compare the memory required to store the full array and the sparse array. Which is more efficient?
(d) Solve the system of equations `a * x = b` for using both the full arrays and the sparse arrays. How do the two sets of answers compare? Time the two solutions. Which one is faster?

7.4 Create a function that accepts any number of numeric input arguments and sums up all of individual elements in the arguments. Test your function by passing it the four arguments a $= 10$, b $= \begin{bmatrix} 4 \\ -2 \\ 2 \end{bmatrix}$, c $= \begin{bmatrix} 1 & 0 & 3 \\ -5 & 1 & 2 \\ 1 & 2 & 0 \end{bmatrix}$, and d $= [1 \quad 5 \quad -2]$.

7.5 Modify the function of the previous exercise so that it can accept either ordinary numeric arrays or cell arrays containing numeric values. Test your function by passing it to the two arguments a and b, where a $= \begin{bmatrix} 1 & 4 \\ -2 & 3 \end{bmatrix}$, b{1} $= [1 \quad 5 \quad 2]$, and b{2} $= \begin{bmatrix} 1 & -2 \\ 2 & 1 \end{bmatrix}$.

7.6 Create a structure array containing all of the information needed to plot a data set. At a minimum, the structure array should have the following fields:

■ `x_data`	x-data (one or more data sets in separate cells)
■ `y_data`	y-data (one or more data sets in separate cells)
■ `type`	linear, semilogx, etc.
■ `plot_title`	plot title
■ `x_label`	x-axis label
■ `y_label`	y-axis label
■ `x_range`	x-axis range to plot
■ `y_range`	y-axis range to plot

You may add additional fields that would enhance your control of the final plot.

After this structure array has been created, create a MATLAB function that accepts an array of this structure and produces one plot for each structure in the array. The function should apply intelligent defaults if some data fields are missing. For example, if the `plot_title` field is an empty matrix, the function should not place a title on the graph. Think carefully about the proper defaults before starting to write your function!

To test your function, create a structure array containing the data for three plots of three different types and pass that structure array to your function. The function should correctly plot all three data sets in three different figure windows.

7.7 Define a structure `point` containing two fields `x` and `y`. The `x` field will contain the x-position of the point, and the `y` field will contain the y-position of the point. Then write a function `dist3` that accepts two points, and returns the distance between the two points on the Cartesian plane. Be sure to check the number of input arguments in your function.

7.8 Write a function that will accept a structure as a argument and return two cell arrays containing the names of the fields of that structure as well as the data types of each field. Be sure to check that the input argument is a structure and generate an error message if it is not.

7.9 Write a function that will accept a structure array of `student` as defined in this chapter, and calculate the final average of each one assuming that all exams have equal weighting. Add a new field to each array to contain the final average for that student, and return the updated structure to the calling program. Also, calculate and return the final class average.

7.10 Write a function that will accept two arguments, the first a structure array and the second a field name stored in a string. Check to make sure that these input arguments are valid. If they are not valid, print out an error message. If they are valid and the designated field is a string, concatenate all of the strings in the specified field of each element in the array and return the resulting string to the calling program.

7.11 **Calculating Directory Sizes.** Function `dir` returns the contents of a specified directory. The `dir` command returns a structure array with four fields, as shown below:

```
» d = dir('chap7')
d =
36x1 struct array with fields:
   name
   date
   bytes
   isdir
```

The field `name` contains the names of each file, `date` contains the last modification date for the file, `bytes` contains the size of the file in bytes, and `isdir` is 0 for conventional files and 1 for directories. Write a function that accepts a directory name and path and returns the total size of all files in the directory, in bytes.

7.12 **Recursion.** A function is said to be *recursive* if the function calls itself. Modify the function created in Problem 7.11 so that it calls itself when it finds a subdirectory and sums up the size of all file in the current directory plus all subdirectories.

7.13 **Function Generators.** Write a nested function that evaluates a polynomial of the form $y = ax^2 + bx + c$. The host function `gen_func` should have three calling arguments—`a`, `b`, and `c`—to initialize the coefficients of the polynomial. It should also create and return a function handle for the nested function `eval_func`. The nested function `eval_func(x)` should calculate a value of `y` for a given value of `x`, using the values of `a`, `b`, and `c` stored in the host function. This is effectively a function generator, since each combination of `a`, `b`, and `c` values produces a function handle that evaluates a unique polynomial. Then perform the following steps:

(a) Call `gen_func(1,2,1)` and save the resulting function handle in variable `h1`. This handle now evaluates the function $y = x^2 + 2x + 1$.

(b) Call `gen_func(1,4,3)` and save the resulting function handle in variable `h2`. This handle now evaluates the function $y = x^2 + 4x + 3$.

(c) Write a function that accepts a function handle and plots the specified function between two specified limits.

(d) Use this function to plot the two polynomials generated in parts (*a*) and (*b*) above.

7.14 **Function Generators.** Generalize the function generator of the previous problem to handle polynomial of arbitrary dimension. Test it by creating function handles and plots the same way that you did in the previous problem. [*Hint:* Use `varagrin`.]

7.15 Look up function `struct` in the MATLAB Help Browser, and learn how to preallocate a structure and simultaneously initialize all of the elements in the structure array to the same value. Then create a 2000 element array of type `student`, with the values in every array element initialized with the fields shown below:

```
 name: 'John Doe'
addr1: '123 Main Street'
 city: 'Anytown'
state: 'LA'
  zip: '71211'
```