Chapter 1

MATLAB PRIMER

This chapter will serve as a hands-on tutorial for beginners who are unfamiliar with MATLAB. We assume that readers have either the student edition or a full version of MATLAB.[1] Before reading this chapter, the reader should set up MATLAB on the computer, open the command window, and try to type and execute the commands as they are explained. Although this book tries to explain the MATLAB commands as clearly as possible, keep in mind that learning a new language is confusing and tedious in the beginning. Trying the commands and practicing them on the computer will help you to understand them.

Readers are encouraged to solve most of the problems at the end of this chapter. Each one is simple, but your knowledge and skill will increase by a continuous effort to solve them. Readers may also seek help by looking at answer keys at the end of the book, as well as similar problems and answers posted at the author's web site.[2]

Throughout this book you will see `log` represented as $\log_e$. The log function to the base 10, namely $\log_{10}$, will be specifically denoted as `log10`. Trigonometric functions use *radians* but not degrees; however, the angles in graphic views are in degrees.

The results of some computations may vary slightly on different computers, but the differences are typically negligible. Some calculations, however, are sensitive to rounding errors and can produce significantly disparate results on different computers. Such problems are often called ill-conditioned problems and are usually difficult to solve on any computer.

[1] MATLAB-6. However, most explanations are applicable to MATLAB 4 and 5 except for a small number of commands. The features available only with MATLAB-6 are indicated in the text.

[2] http://olen.eng.ohio-state.edu/matlab

1.1 BEFORE STARTING CALCULATIONS

how to open MATLAB: On a Unix workstation, MATLAB can be opened by typing

```
> matlab
```

On a PC, MATLAB can be started by clicking on the start-up menu or a shortcut icon for MATLAB. When MATLAB-6 is started, the window, as illustrated in Figure 1.1, will open. The left side is divided into two subwindows. The right side is the command window, where the most important work for MATLAB is done. Earlier versions did not have such divided subwindows on the left side.

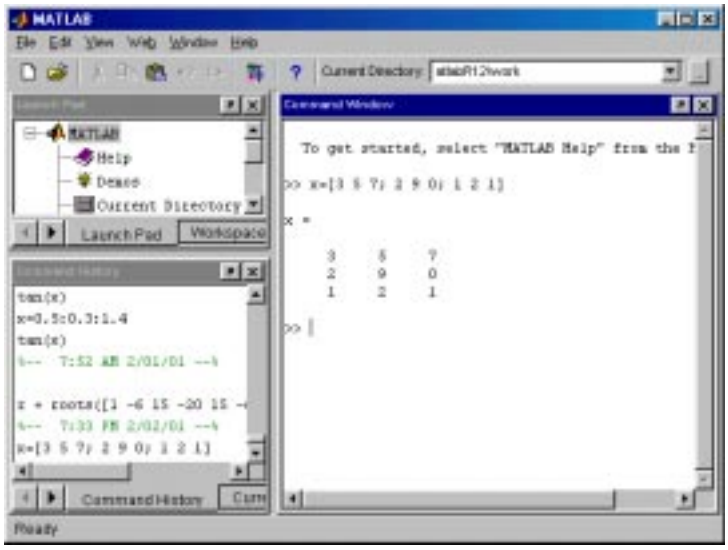

Figure 1.1 MATLAB-6 Desktop.

Once the prompt sign "`>>`"appears in the command window, type the commands that are explained throughout this section. On Windows, the initial (default) working directory is `C:/MATLAB/bin`. If you wish to use a floppy disk as a working directory, type the command to change directory, namely "`cd a:`", in after the `>>` sign, for example

```
>> cd a:
```

If the working directory is `C:/my_directory`, type

```
>> cd ../..
>> cd my_directory
```

On Unix, MATLAB can be opened from any directory. To quit MATLAB, type

```
>> quit
```

help: When the meaning of a command is not clear, type `help` followed by any command in question. The `help` command will give you a concise but precise explanation and will be one of the most frequently used commands as you proceed. If you type `help date` or `help format` as examples, responses are, respectively, as follows:

```
>> help date
   DATE   Current date as date string.
    S = DATE returns a string containing the date in dd-mmm-yyyy format.
    See also NOW, CLOCK, DATENUM.

>> help format
   FORMAT Set output format.
    All computations in MATLAB are done in double precision.
    FORMAT may be used to switch between different output
    display formats as follows:
      FORMAT         Default. Same as SHORT.
      FORMAT SHORT   Scaled fixed point format with 5 digits.
      FORMAT LONG    Scaled fixed point format with 15 digits.
      FORMAT SHORT E Floating point format with 5 digits.
      FORMAT LONG E  Floating point format with 15 digits.
      FORMAT SHORT G Best of fixed or floating point format
                     with 5 digits.
      FORMAT LONG G  Best of fixed or floating point format
                     with 15 digits.
      FORMAT HEX     Hexadecimal format.
      FORMAT +       The symbols +, - and blank are printed
                     for positive, negative, and zero elements.
                     Imaginary parts are ignored.
      FORMAT BANK    Fixed format for dollars and cents.
      FORMAT RAT     Approximation by ratio of small integers.

    Spacing:
      FORMAT COMPACT Suppress extra line-feeds.
      FORMAT LOOSE   Puts the extra line-feeds back in.
```

In the online `help`, keywords are capitalized to make them stand out. Always type commands in lowercase as all command and function names are actually in lowercase.

Be aware that the `help` command is limited to the questions regarding the commands. If you have a question that is more general in nature, click the *Help* menu on the top menu bar and then click on *MATLAB Help*. A window as shown in Figure 1.2 will open with the index page already in front. Type the keyword for your question in the small white box under *Search index for*. The remainder of the process is similar to that of `help` in MS Windows.

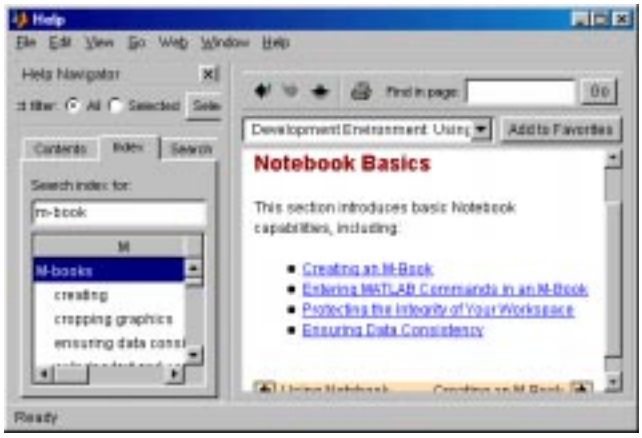

Figure 1.2 Help window.

version: The first thing you should know about MATLAB is what version you are using. To get this information, type `version`.

pwd: `pwd` prints out the current working directory name.

dir: `ls` or `dir` lists all the filenames in the current directory.

cd: `cd` changes the directory.

what: `what` will list M-, MAT-, and MEX-files in the current working directory.[3] The command `what dirname` lists the files in directory `dirname` on the `matlabpath`. It is not necessary to give the full pathname in the position of `dirname`; the last component or last several components are sufficient. For example, `what general` and `what matlab/general` both list the m-files in

[3]m-file: a script or function file (its format is `filename.m`); MAT-file: a file containing binary data (its format is `filename.mat`); MEX-file: MATLAB executable file compiled from Fortran or C (`filename.mex`)

directory `toolbox/matlab/general`.

who: `who` lists the variables in the current workspace. `whos` lists more information about each variable. `who global` and `whos global` list the variables in the global workspace.

clock: The `clock` command prints out numbers such as

```
ans =
      1.0e+03 *
      2.001  0.0030  0.0050  0.0150  0.0140  0.0091
```

The first number, `1.0e+03`, is a multiplier. The numbers in the second line have the following meaning:

```
[year, month, day, hour, minute, second]
```

The clock values can also be printed out in the integer form by `fix(clock)`. The answer is

```
ans =
        2001    3    5   15   19   56
```

where time was year 2001, 3rd month, 5th day, 15 hr, 19 min and 56 sec, approximately six minutes after the first example of `clock` was printed out. The elapsed time of an execution may be measured by `clock`. For example, set `t_0=clock` before a computation starts and `t_1=clock` when completed. Then, `t_1 - t_0` gives the time elapsed for the computation. You can also use `tic` and `toc` to measure the elapsed time.

The `date` command gives similar information, but in a more concise format:

```
ans =
      5-Mar-01
```

path: `path` or `matlabpath` prints MATLAB's current search path. Command `p = path` returns a string `p` containing the path. Command `path(p0)` changes the path to `p0`, which is a string containing the new path. Command `path(p1,p2)` changes the path to the concatenation of the two path strings, `p1` and `p2`. Therefore, `path(path, p3)` appends a new directory, `p3`, to the current path and `path(p3, path)` prepends the new path.

getenv: `getenv('matlabpath')` will show current MATLAB paths. If you have not set any path, the response of the computer is simply two dots.

diary on, diary off: with `diary` you can record all activities on the MATLAB window. The `diary on` command starts writing all keyboard input as well as most screen output to a file named `diary`. `diary off` terminates writing. If file `diary` already exists, the screen output is appended to the existing diary file. A filename other than `diary` may be specified by writing the intended filename after `diary`. Without `on` or `off`, `diary` itself toggles `diary on` and `diary off`. The file may be printed as a hard copy or may be edited later.

!(escape): The ! mark is the operator to escape from MATLAB. With this mark, you have access to the DOS or Unix commands. A text editing software like the *vi* editor may be opened from within MATLAB by using `!vi filename` if you are on an Unix computer. Escape may be used similarly on a PC for some DOS commands. `!erase filename` deletes the file named `filename` on a PC. A subdirectory named `d.subdir` on PC can be created by using `!mkdir d.subdir`. Formatting a diskette from MATLAB on a PC is possible by using `!format a:`. Executing PC or Unix software in this way is not recommended, however, because some programs, particularly graphic or communication software, may harm the computing environment.

demo: The `demo` command guides the user to numerous demonstrations that are selectable from a menu. The MATLAB demonstrations are fun. Visit them often until you understand them in detail.

pathological symptom: MATLAB can get sick for various reasons. Possible causes may include: (1) wrong combination of commands given by the user, (2) its own bug, or (3) software instability due to insufficient memory. Pathological symptoms tend to occur more frequently with an early edition of an updated version that is loaded with many new features. If strange behavior is suspected, shut down MATLAB and restart. If a software error is found, it should be reported to the company. Often, the company's web site will explain and provide a patch to update the software.

1.2 HOW TO DO CALCULATIONS

arithmetic operators: Arithmetic operators `+`, `-`, `*`, and `/` are, respectively, *plus*, *minus*, *multiply*, and *divide*, the same as in traditional programming languages. To express power, the operator `^` is used. MATLAB uses one untraditional operator, `\`, which may be named *inverse division*. This operator yields the reciprocal of division, that is, `a\b` equals `b/a`. For example,

```
c = 3\1
c =
    0.3333
```

It is not recommended that readers use this operator in typical computations, but it will become important in Chapter 3, "Linear Algebra."

calculation with single variables: When a command window is opened, a prompt sign `>>` is seen at the upper left corner of the window. Any command can be written after the prompt sign. In the explanation of the commands, however, the prompt sign will be omitted for simplicity.

As a simple example, let us evaluate:

$$\text{Volume} = \frac{4}{3}\pi r^3, \quad \text{with } r = 2$$

The commands to type on the screen are:

List 1.1a

```
r = 2;
vol = (4/3)*pi*r^3;
```

where `pi` $= \pi$, that is, 3.14159265358979, in MATLAB. Each line is typed after the prompt sign `>>` and the return key is hit when finished typing. Notice in the preceding script that each line is a command and completed by a semicolon. The caret symbol `^` after `r` is the power operator.

When we work in the command window, the computer calculates the answer for each command immediately after the return key is hit. Therefore, the value of `vol` is already in the computer. How can we get the result printed out on the screen?

The quickest way of printing out the result is to type `vol` and hit return. Then the computer prints out

```
vol =
      33.510
```

Another way of printing out the value of `vol` is to omit the semicolon at the end of the second command

List 1.1b

```
r = 2;
vol = (4/3)*pi*r^3
```

Without a semicolon, the result is printed out immediately after the computation. Because displaying every result is cumbersome, however, we generally place a semicolon after each command.

Multiple commands may be written in a single line separated by semicolons. If the results are to be printed out for each command executed, separate commands by commas. The line may be terminated with or without a comma. For example, if you write

```
r = 2,  vol = (4/3)*pi*r^3
```

the values of `r` and `vol` are printed out, but if you write

```
r = 2;  vol = (4/3)*pi*r^3;
```

no results are displayed.

A long command may be split into multiple lines. In Fortran, it is done by a continuation mark on column 6. In MATLAB, the continuation mark is ... and it is placed at the end of the line to be continued; for example,

List 1.2

```
r = 2;
vol = (4/3)*3.14159  ...
      *r^3;
```

The prompt sign will not appear in the line following the continuation mark.

variables and variable names: Variable names and their types do not have to be declared. This is because variable names in MATLAB make no distinction among integer, real, and complex variables. Any variable can take real, complex, and integer values.

In principle, any name can be used as long as it is compatible in MATLAB. We should, however, be aware of two incompatible situations. The first is that the name is not accepted by MATLAB. The second is that the

name is accepted, but it destroys the original meaning of a reserved name. These conflicts can occur with the following types of names:

(a) Names for certain values
(b) Function (subroutine) names
(c) Command names

One method to examine compatibility of the variable name is to test it on the command screen. A valid statement such as `x=9` is responded to as

```
x =
    9
```

which means that the variable is accepted. If `end=4` (a bad example) is attempted, however, it is ignored.

An example of the second conflict is as follows: If `sin` and `cos` are used (as poor examples of variables) with no relation to the trigonometric functions, for example,

```
sin = 3;
cos = sin^2;
```

the calculations proceed; however, `sin` and `cos` can never be used as trigonometric functions thereafter until variables are cleared by issuing the `clear` command or MATLAB is shut down. If any error message concerning a conflict appears, the reader should investigate the cause.

Traditionally, symbols i, j, k, l, m, and n have been used as integer variables or indices. At the same time, i and j are used to denote a unit imaginary value, or $\sqrt{-1}$. In MATLAB, i and j are reserved as unit imaginary value. Therefore, if the computation involves complex variables, it is advisable to avoid i and j as user-defined variables.

Table 1.1 lists samples of reserved variable names that have special meanings. Whether a variable or filename exists may also be checked by using the `exist` command. For more details, type `help exist`.

format: Numbers displayed are five-digit numbers by default:

```
pi
ans =
     3.1416
```

The same numbers, however, may be displayed with 16 digits after the command `format long`; for example,

Table 1.1 Special Numbers and Variable Names

Variable name	Meaning	Value
`eps`	Machine epsilon	2.2204e-16
`pi`	π	3.14159...
`i` and `j`	Unit imaginary	$\sqrt{-1}$
`inf`	Infinity	∞
`NaN`	Not a number	
`date`	Date	
`flops`	Floating point operation count	
`nargin`	Number of function input arguments	
`nargout`	Same for output	

```
format long
pi
ans =
     3.141592653589793
```

In order to return to the short format, use `format short`. Also, with `format short e` and `format long e`, respectively, short and long numbers are printed in floating-point format.

clear: As you execute commands, MATLAB memorizes the variables used. Their values stay in memory until you quit MATLAB or clear the variables. To clear all the variables, use the `clear` command. If only certain variables are to be cleared, name the variables after `clear`; for example,

```
clear x y z
```

clc: If you wish to clear a window, use the command `clc`.

1.3 BRANCH STATEMENTS

if, else, elsif, end: An `if` statement is always closed with an `end` statement; for example,

List 1.3

```
n = 2;    %(Assume n is a user input so it may be changed)
if    n<=5,   price=15;
elseif n>5,  price=12;
end
```

Here the line starting `elseif` can be eliminated if not needed. Do not separate `elseif` into two words. Of course, `elseif` can be repeated when necessary. Notice also in writing the foregoing script that the prompt sign does not appear after `if` until `end` is typed. The foregoing script may be equivalently written as

```
n = 2;    %(Assume n is a user input so it may be changed)
if    n<=5,   price=15;
else  price=12;
end
```

When the `if` statement needs to examine the equality of two terms, use "`==`"as in the C language; for example,

List 1.4

```
n = 2;
if n==2, price=17;
end
```

The *not equal* operator is written as "`~=`,"for example,

List 1.5

```
r = 2;
if r ~= 3, vol = (4/3)*pi*r^3;
end
```

The *greater than*, *less than*, *equal or greater than*, and *equal or less than* operators are, respectively,

```
>
<
>=
<=
```

The logical statements *and* and *or* are denoted by `&` and `|`, respectively. For example, the conditional equation,

$$\text{if } g > 3 \text{ or } g < 0, \text{ then } a = 6$$

is written as

```
if g>3 | g<0, a = 6; end
```

Also, the conditional equation

if $a > 3$ and $c < 0$, $b = 19$

is stated as

```
if a>3 & c<0, b=19; end
```

The `&` and `|` operators can be used in a clustered form, for example,

```
if ((a==2 | b==3) & c<5)   g=1; end
```

Example 1.1

Assuming two arbitrary integers, R and D, are given in the beginning of the script, write a script that prints out only if R is divisible by $D = 3$.

Solution

Two equivalent ways to examine if R is divisible by another number are: (1) `fix(R/D)=R/D` , and (2) `mod(R,D)=0`. Here, `fix(x)` returns the integer part of the number `x`, while `mod(x,y)` returns the remainder on division of x by `y`. Therefore, a script based on (1) is:

List 1.6a

```
R=45; D=3;
if fix(R/D) == R/D, R, end
```

A script based on (2) is

List 1.6b

```
R=45; D=3;
if mod(R,D) == 0, R, end
```

1.4 LOOPS WITH for/end OR while/end

loops: MATLAB has `for/end` and `while/end` loops. To illustrate a `for/end` loop, let us look at the following script that calculates $y = x^2 - 5x - 3$ for each of $x = 1, 2..9$ in increasing order.

List 1.7a

```
for x=1:9
   y=x^2 - 5*x - 3
end
```

Notice that `for` is terminated by `end`. In the first cycle, x is set to 1, and y is calculated. In the second cycle, x is set to 2 (with an increment of 1), and y is calculated. The same is repeated until the calculation of y is completed for the last value of x.

If x is to be changed with a different increment, the increment can be specified between the initial and last number as follows:

List 1.7a

```
for x=1:0.5:9
   y=x^2 - 5*x - 3
end
```

where the increment is now 0.5.

The order of calculation in the loop can be reversed as follows:

List 1.7b

```
for x=9:-1:1
   y=x^2 - 5*x - 3
end
```

Here, the middle number -1 in 9:-1:1 is the increment in changing x.

It is also possible to compute for any sequence of specified values of x, for example:

List 1.7c

```
for x=[-2 0 15 6]
   y=x^2 - 5*x - 3
end
```

Here, y is computed for x =-2 first, which is followed by x =0, 15, and 6.

The `if/end` statement can be inserted in the loop. In the following example, $y = \sin(x)$ if $\sin(x) > 0$ but $y = 0$ if $\sin(x) < 0$.

List 1.7d

```
for x=0:0.1:10
   y=sin(x);
   if y<0, y=0;end
   y
end
```

In the following script, `c=0` initializes the counter `c` to zero, `x=[-8, ..]` defines an array of the numbers, and `length(x)` is the length of the array `x`. In the `for/end` loop the counter `c` is incremented by one if x(i) is negative. Finally, `c` prints out the count of the negative elements.

List 1.7e

```
c=0; x=[-8, 0, 2, 5, 7, 2, 0, 0, 4, 6, 6, 9];
for i=1:length(x)
   if x(i)<0, c=c+1; end
end
c
```

The loop index can be decremented as

List 1.8

```
c=0; x=[-8, 0, 2, 5, 7, 2, 0, 0, 4, 6, 6, 9];
for i=length(x):-1:1
   if x(i)<0, c=c+1; end
end
c
```

In this example, `-1` between two colon operators is the decrement of the parameter `i` after each cycle of the loop operation.

An alternative way of writing a loop is to use the `while/end`; for example,

List 1.9

```
i = 0;c=0; x=[-8, 0, 2, 5, 7, 2, 0, 0, 4, 6, 6, 9];
while i<length(x)+1
   i=i+1
   if x(i)<0, c=c+1; end
end
c
```

Example 1.2

Write a script that removes the numbers divisible by 4 from an array `x`. Assume the array `x` is given by

```
x=[-8, 0, 2, 5, 7, 2, 0, 0, 4, 6, 6, 9]
```

Solution

The script to the answer is

List 1.9

```
x=[-8, 0, 2, 5, 7, 2, 0, 0, 4, 6, 6, 9];
y=[];
for n=1:length(x)
   if x(n)/4 - fix(x(n)/4)~=0  y=[y,x(n)]; end
end
y
```

In the foregoing script, `fix(x(n)/4)` equals the integer part of `x(n)/4`, so `fix(x(n)/4)-x(n)/4=0` is satisfied if `x(n)` is divisible by 4. Equivalently, `mod(a,b)` may be used that equals zero if `a` is divisible by `b`. The following script shows alternative programming to achieve the same end:

List 1.9

```
x=[-8, 0, 2, 5, 7, 2, 0, 0, 4, 6, 6, 9];
for n=1:length(x)
   if mod(x(n),4)=0  x(n)=[]; end
end
x
```

The result is

```
y =
     2     5     7     2     6     6     9
```

break: The `break` terminates the execution of a `for` or `while` loop. When used in nested loops, only the immediate loop where `break` is located is terminated. In the next example, `break` terminates the inner loop as soon as `j>2*i` is satisfied once, but the loop for `i` is continued until `i=6`:

List 1.10

```
for i=1:6
   for j=1:20
     if j>2*i, break, end
   end
end
```

Another example is

List 1.11

```
r=0
while r<10
  r = input('Type radius (or -1 to stop): ');
  if  r< 0, break, end
  vol = (4/3)*pi*r^3;
  fprintf('Volume = %7.3f\n', vol)
end
```

In the foregoing loop, the radius r is typed through the keyboard. The `fprintf` statement is to print out `vol` with a format, `%7.3f`, which is equivalent to `F7.3` in Fortran. If $0 \leq r < 10$, `vol` is computed and printed out, but if $r < 0$ the loop is terminated. Also, if $r < 10$ is dissatisfied once, the `while`

loop is terminated. More explanations for `input` and `fprintf` are given in later subsections.

In a programming language that has no break command, `goto` would be used to break a loop. MATLAB, on the other hand, has no `goto` statement.

infinite loop: Sometimes a loop that can continue infinitely is used, which may be terminated when a certain condition is met. The following example shows an infinite loop that is broken only if the condition `x > xlimit` is met:

```
while 1
   .
   .
    if x > xlimit, break; end
   .
   .
end
```

1.5 READING AND WRITING

Passing data to and from MATLAB is possible in several different ways. The methods may be classified into three classes:

(a) Interactive operation by keyboard or mouse
(b) Reading from or writing to a data file
(c) Using `save` or `load`

In the remainder of this subsection, only a minimal amount of information regarding reading and writing is introduced. More information can be found in Section 1.8.

input: MATLAB can take input data through the keyboard using the `input` command. To read a number, the synopsis would be

```
z = input('Type your input: ')
```

The string between quote signs, namely, `Type your input:`, is a prompting message to be printed out on the screen. As a value is typed and a return key is hit, the input is saved in `z`. A string input can be typed from the keyboard. The synopsis is

```
z = input('Your name please: ', 's')
```

The second argument `'s'` indicates that the input from the keyboard is a string. The variable `z` becomes an array variable (row vector) unless the string has only one character. A string input can be taken by `input` without `'s'` if the typed string is enclosed by single quote signs. In this case, a prompt message may be written as

```
z = input('Type your name (in single quote signs):')
```

fprintf: Printing out formatted messages and numbers is possible using `fprintf`; for example,

```
fprintf('The volume of the sphere %12.5f.\n', vol)
```

Included between two single quote signs are a string to be printed out, the format for the volume, and a new-line operator. The style of the format is familiar to those who know the C language: `The volume of the sphere` is the string to be printed out, `%12.5f` is the format and similar to `F12.5` in Fortran, and `\n` is the new-line operator that advances the screen position by one line. The new-line operator can be placed anywhere within the string. Finally, `vol` is the variable to be printed out in accordance with the format `%12.5f`. If `\n` is omitted, the next print starts without advancing a line.

The command

```
fprintf('e_format: %12.5e\n', 12345.2)
```

will print out

```
e_format:  1.23452e+04
```

If two print statements are consecutively written without `\n` in the first statement, for example,

```
fprintf('e_format: %12.5e', 12345.2);
fprintf('f_format: %12.3f\n', 7.23462)
```

then all the output will be printed out in a single line as

```
e_format: 1.23452e+04  f_format: 7.235
```

An integer value can be typed using the same format, except that 0 is placed after the decimal point; for example,

```
fprintf('f_format: %12.0f\n', 93)
```

yields

```
f_format: 93
```

When multiple numbers are to be printed on a single line, `fprintf` may be repeatedly used without `\n`, except in the last statement.

With the `fprintf` command, it is possible to write formatted output into a file. To do this, the named file has to be opened by the `fopen` command, for example,

```
vol=55.0
fileid = fopen('file_x','w');
fprintf(fileid,'Volume=  %12.5f\n', vol);
fclose(fileid);
```

will write the output in the file named `file_x`. If no such file exists, a new file is created. If the file exists, the output is appended. If necessary, the existing file `file_x` can be deleted by `!rm file_x` on Unix, or `!erase file_x` on Windows.

disp: Command `disp` displays a number, vector, matrix, or a string on the command window without variable name. Therefore, it may be used to display messages or data on the screen. For example, `disp(pi)` and `disp pi` both print `3.14159` on the command screen. Try also `disp 'This is a test for disp.'`.

sprintf: Writing `sprintf` is very similar to `fprintf` except that `sprintf` writes the output into a string. This statement is often used to create a command in a string that can be executed as `eval(s)`. It is useful when a command is to be created or edited automatically and executed within an m-file.

1.6 ARRAY VARIABLES

one-dimensional array variables: One-dimensional array variables are in a column or a row form, and are closely related to vectors and matrices. In MATLAB, *row array* is synonymous with *row vector*, and *column array* is synonymous with *column vector*. The variable `x` can be defined as a row vector by specifying its elements; for example:

```
x = [0, 0.1, 0.2, 0.3, 0.4, 0.5];
```

To print a particular element, type `x` with its subscript. For example, typing `x(3)` as a command will show

```
ans =
       0.2
```

An equivalent way of defining the same `x` is

```
for i=1:6
   x(i) = (i-1)*0.1;
end
```

The size of the vector does not have to be predeclared as it is adjusted automatically. The number of elements of `x` can be increased by defining additional elements, for example,

```
x(7) = 0.6;
```

A row array variable with a fixed increment or decrement may be equivalently written as

```
x = 2:-0.4:2
```

It yields

```
x = 2.0000  1.6000  1.2000  0.8000  0.4000  -0.0000
```

The definition of a column array is similar to a row array except that the elements are separated by semicolons; for example,

```
z = [0; 0.1; 0.2; 0.3; 0.4; 0.5];
```

An alternative way of defining the same thing is to put a prime after a row array:

```
z = [0, 0.1, 0.2, 0.3, 0.4, 0.5]';
```

The prime operator is the same as the transpose operator in the matrix and vector calculus, so it converts a column vector to a row vector and vice versa. Typing `z` as a command yields

```
z =
    0
    0.1
    0.2
    0.3
    0.4
    0.5
```

If a single element of an array `c` is specified, for example,

```
c(8) = 11;
```

`c(i) = 0` is assumed for `i=1` through `7`. Therefore, typing `c` yields

```
c =
     0   0   0   0   0   0   0   11
```

Likewise

```
clear c
c(1:5)=7
```

produces

```
c =
     7   7   7   7   7
```

Also

```
clear c
c(1:2:7)=5
```

yields

```
c =
     5  0  5  0  5  0  5
```

which can be further changed by

```
c(2:2:7)=1
```

to

```
c =
     5  1  5  1  5  1  5
```

When y and x have the same length and the same form (row or column), the vector y and x can be added, subtracted, multiplied, and divided using the array arithmetic operators as

```
z = x + y
z = x - y
z = x .* y
z = x ./ y
```

which are equivalent, respectively, to

List 1.12

```
for i=1:6;  z(i) = x(i) + y(i); end
for i=1:6;  z(i) = x(i) - y(i); end
for i=1:6;  z(i) = x(i)*y(i);   end
for i=1:6;  z(i) = x(i)/y(i);   end
```

The rules for addition and subtraction are the same as for vectors in linear algebra. However, `.*` and `./` are named array multiplication operators and array division operators, respectively, which are not the same as multiplication and division for matrices and vectors. If the period in `.*` or `./` is omitted, the meaning becomes entirely different (see Chapter 3, "Linear Algebra,"for more details).

The array power operator is illustrated by

```
g = z.^1.2;
```

where `z` is a vector of length 6, a period is placed before the `^` operator, and `g` becomes a vector of the same length. The foregoing statement is equivalent to

```
for i=1:length(g);  g(i) = z(i)^1.2; end
```

where no period is placed before the `^` operator.

The size of an array can be increased by appending an element or a vector (or vectors). As an example, assume

```
x =
    2   3
```

The following command appends `5` to `x` and makes its length 3:

```
x = [x, 5]
```

which returns

```
x =
    2   3   5
```

A column vector may be appended with a number or a vector or vectors. Suppose `y` is a column vector,

```
y =
    2
    3
```

then

```
y = [y; 7]
```

yields

```
y =
    2
    3
    7
```

Here, `7` is appended to the end of the column vector. Notice that a semicolon is used to append to a column vector. An element can be prepended to a vector also, for example, `x = [9,x]` yields

```
x =
    9   2   3   5
```

where `x` on the right side was defined earlier. Similarly, `[-1;y]` yields

```
y =
     -1
      2
      3
      7
```

A reverse procedure is to extract a part of a vector. For the foregoing `y`,

```
w = y(3:4)
```

will define `w` that equals the 3rd and 4th elements of `y`, namely

```
w =
    3
    7
```

length, size: If you don't remember the size of a vector, ask the computer. For a vector

```
x =  [9,  2,  3,  5]
```

the inquiry

```
length(x)
```

is responded to by

```
ans =
      4
```

The answer is the same for a column array. Let us define `y = [9, 2, 3]'`. Then, `length(y)` returns `ans = 3`; however, when you want to know if the vector is a column or row type in addition to the length, use `size`. For example, `size(y)` will return

```
ans =
      3     1
```

where the first number is the number of rows and the second number is the number of columns. From this answer, we learn that `y` is a 3×1 array, that is, a column vector of length 3. For `z=[9,2,3,5]`, `size(z)` will return

```
ans =
       1     4
```

that is, `z` is a row vector of length 4.

deletion of array elements: An element of an array may be deleted as follows. Assume `z=[3, 5, 7, 9]`, and the third element is to be deleted from the array. Then

```
z(3)=[]
```

yields

```
z =
   [3, 5, 9]
```

string variables: String variables are arrays. For example, a string variable `v` defined by

```
v = 'glacier'
```

is equivalent to

```
v = ['g', 'l', 'a', 'c', 'i', 'e', 'r']
```

The variable `v` can be converted to a column string by

```
v = v'
```

which is

```
g
l
a
c
i
e
r
```

two-dimensional array variables: A two-dimensional array, which is synonymous with a matrix in MATLAB, can be defined by specifying its elements. For example, a 3×3 array can be defined by

```
m = [0.1, 0.2, 0.3; 0.4, 0.5, 0.6; 0.7, 0.8, 0.9];
```

Notice that the elements for a row are terminated by a semicolon. Of course, the number of elements in each row must be identical. Otherwise the definition will not be accepted. The statement is equivalent to writing

List 1.13

```
m(1,1)=0.1;
m(1,2)=0.2;
m(1,3)=0.3;
m(2,1)=0.4;
m(2,2)=0.5;
m(2,3)=0.6;
m(3,1)=0.7;
m(3,2)=0.8;
m(3,3)=0.9;
```

Typing `m` as a command yields

```
m =
    0.1000    0.2000    0.3000
    0.4000    0.5000    0.6000
    0.7000    0.8000    0.9000
```

A whole column or a row of a two-dimensional array can be expressed using a colon. For example, `m(1,:)` and `m(:,3)` are the first row of `m` and the third column of `m`, respectively, and treated as vectors. For example,

```
c(1,:) = m(3,:);
c(2,:) = m(2,:);
c(3,:) = m(1,:);
```

yields

```
c =
    0.7000    0.8000    0.9000
    0.4000    0.5000    0.6000
    0.1000    0.2000    0.3000
```

Two-dimensional arrays can be added, subtracted, multiplied, and divided using the array arithmetic operators:

List 1.14a

```
c = a + b
c = a - b
c = a .* b
c = a ./ b
```

Here, `a` and `b` are two-dimensional arrays of the same size. The foregoing statements are equivalent to, respectively,

List 1.14b

```
for i=1:3
   for j=1:3
      c(i,j)  = a(i,j) + b(i,j);
   end
end

for i=1:3
   for j=1:3
      c(i,j)  = a(i,j) - b(i,j);
   end
end

for i=1:3
   for j=1:3
      c(i,j)  = a(i,j)*b(i,j);
   end
end

for i=1:3
   for j=1:3
      c(i,j)  = a(i,j)/b(i,j);
   end
end
```

Note that the expressions in List 1.14a are significantly more compact and clearer than the expressions in List 1.14b.

The statement with the array power operator,

```
g = a.^3
```

is equivalent to

```
for i=1:3
   for j=1:3
      g(i,j)  = a(i,j)^3;
   end
end
```

Column vectors and row vectors are both special cases of a matrix. Therefore, array operators work equally on vectors and matrices. There are two advantages in using the array arithmetic operators. First, programming becomes short. Second, the computational efficiency of MATLAB is higher with the short form than writing the same using loops.

if and arrays: Array variables may be compared in an `if` statement, assuming that `a` and `b` are matrices of the same size:

(1) `if a==b` is satisfied only if `a(i,j)=b(i,j)` for all the elements.

(2) `if a>=b` is satisfied only if `a(i,j)>=b(i,j)` for all the elements.

(3) `if a~=b` is satisfied if `a(i,j)~=b(i,j)` for at least one element.

If two string variables of different lengths are compared by an `if` statement, an arithmetic error occurs because the two arrays must have the same length. In order to compare string variables in `if` statements, all the string variables must be adjusted to a predetermined length by appending blank spaces. For example, instead of

```
a = 'echinopsis'
b = 'thithle'
c = 'cirsium'
d = 'onopordon'
```

write as

```
a = 'echinopsis'
b = 'thithle   '
c = 'cirsium   '
d = 'onopordon '
```

Then, `a`, `b`, and `c` may be compared in `if` statements.

This task may be more easily achieved, however, by `str2mat`. For example, suppose string variables have been given by

```
t1 = 'digitalis'
t2 = 'nicotiana'
t3 = 'basilicum'
t4 = 'lychnis'
t5 = 'chrysanthemum'
```

Then they may be organized in a single string matrix by

```
s = str2mat(t1,t2,t3,t4,t5)
```

The first row of `s` becomes `t1`, the second row `t2`, and so on, with an identical length because blank spaces are added to shorter strings.

Example 1.3

(a) An array of numbers is given by

```
x=[9, 2, -3, -6, 7, -2, 1, 7, 4, -6, 8, 4, 0, -2];
```

Write a script to count the number of negative entries.

(b) Write a script to find the minimum and maximum values in the array given in (a). (Do this in a primitive way without using `min` or `max` commands.)

Solution

(a) The key for the script writing is to use `for/end` and `if/end`. The counter `c` is initialized to 0 in the beginning. Then, `x(i)<0` is examined for each of `x(i)`. If the condition is satisfied, the counter is increased by one. The final answer is typed simply by `c`.

```
x=[9, 2, -3, -6, 7, -2, 1, 7, 4, -6, 8, 4, 0, -2];
c=0;
for i=1:length(x);
  if x(i)<0 c=c+1;end
end
c
```

The answer is

```
c =
     5
```

(b) In the following script, `xmin` and `xmax` are initialized to a large positive and large negative number, respectively. Then, `xmin` and `xmax` are compared to each `x(i)`. If `xmin` is greater than

`x(i)`, `xmin` is set to `x(i)`, and, similarly, if `xmax` is smaller than `x(i)`, `xmax` is set to `x(i)`.

```
x=[9, 2, -3, -6, 7, -2, 1, 7, 4, -6, 8, 4, 0, -2];
xmin=999;  xmax=-999;
for i=1:length(x);
  if xmin>x(i), xmin=x(i);end
  if xmax<x(i), xmax=x(i);end
end
[xmin,xmax]
```

The answer is

```
ans =
       -6     9
```

Comment: Be aware that `min` and `max` commands find the minimum and maximum, respectively, and make the task easier.

Example 1.4

(a) Write a script to examine if a given number is a prime number or not. Read the integer to be examined by the `input` command. Do not use `factor` or `isprime` commands. (Hint: Check to see if the number is divisible by all the lower numbers except unity).

(b) Write a script to print out the prime numbers less than 100 in increasing order. Print out the total number of the prime numbers found, a list of the prime numbers, and the sum of the prime numbers less than 100. (Do not use `factor` or `isprime` commands).

Solution

(a) In the following script, the number to be examined, `i`, is given through the `input`. The `ans` is initialized to 1. Then the number `i` is divided by 2 through i-1. If `fix(i/n)` equals `i/n` for any n, `ans` is set to 0 and the loop is terminated by `break`. If the loop is completed without resetting `ans=1`, the `i` is a prime number.

```
clear
i=input('Type an integer greater than 1: ');
ans=1;
for n=2:i-1
    if i/n==fix(i/n), ans=0; break;end
```

```
end
if ans==0, fprintf('No, %2.0f is not a prime number.\n',i)
else       fprintf('Yes, %2.0f is a prime number.\n',i)
end
```

A sample output is

```
Type an integer greater than 1: 5
Yes,  5 is a prime number.
```

(b) The script for this part is essentially the same as for (a), except the prime number tested is changed in another loop from 2 to 99. The array `prime` is initialed to 2 because it is the first prime number. If `i` examined is a prime number, then it is appended to `prime`.

```
clear
prime=[2];
for i=3:99
  ans=1;
  for n=2:i-1
    if i/n==fix(i/n), ans=0; break ;end
  end
  if ans==1, prime=[prime,i];end
end
k=length(prime);
fprintf('\n\nPart (b) answer\n')
fprintf('Number of prime numbers = %3.0f\n',k)
fprintf('Prime numbers less than 100\n',k)
for j=1:length(prime)
  fprintf('%4.0f',prime(j));
  if j==fix(j/10)*10, fprintf('\n');end
end
fprintf('\nThe sum of the prime numbers = %4.0f\n',sum(prime))
```

The results are

```
Part (b) Answer
Number of prime numbers =  25
Prime numbers less than 100
   2   3   5   7  11  13  17  19  23  29
  31  37  41  43  47  53  59  61  67  71
  73  79  83  89  97
The sum of the prime numbers = 1060
```

1.7 UNIQUE ASPECT OF NUMBERS IN MATLAB

In ordinary programming languages, numbers are classified into several categories such as single, double, real, integer, and complex. In MATLAB, all variables are treated equally in double precision. There is no distinction between integer and real variables, nor between real and complex variables. How to assign a value to a variable is entirely up to the user. If a variable is to be used as an integer, just set the value as an integer. Integers are recognized as far as they are recognizable from the mantissa and exponents in the memory. No distinction between real and complex variables is unique to MATLAB, but it provides great advantages. In Fortran, for example, real variables and complex variables cannot share the same subroutines.

As a simple example, consider roots of a quadratic polynomial

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

The solution may be written as

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

In Fortran or C, one has to separate the solutions to two cases:

(a) $b^2 \geq 4ac$,

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

(b) $b^2 < 4ac$,

$$x = \frac{-b \pm i\sqrt{4ac - b^2}}{2a}$$

where i equals $\sqrt{-1}$, and the solutions in the second case are complex values. In MATLAB, however, no separation is necessary. Regardless of the sign of the value inside the square root, the roots are computed by

```
x1 = (-b + sqrt(b^2 - 4*a*c))/(2*a)
x2 = (-b - sqrt(b^2 - 4*a*c))/(2*a)
```

If the roots are complex, MATLAB automatically treats the variables as complex.

Accuracy of computations is affected by a number of factors. The key parameters that affect computational accuracy in a programming language are

Smallest positive number: *x_min*
Largest positive number: *x_max*
Machine epsilon: *eps*[4]

In Table 1.2, these three numbers in MATLAB are compared to those in Fortran on a few typical computers.

Table 1.2 Comparison of the Range of Numbers and Machine Epsilon

Software Precision	MATLAB	Fortran(workstation) Single (Double)	Fortran(Cray)
x_min	4.5e-324	2.9e-39(same)	4.6e-2476
x_max	9.9e+307	1.7e+38(same)	5.4e+2465
eps	2.2e-16	1.2e-7(2.8e-17)	1.3e-29

The foregoing table shows that the machine epsilon (*eps*) of MATLAB is equivalent to that of double precision in Fortran on typical workstations.[5] MATLAB treats all numbers in double precision. The *x_min* of MATLAB is significantly smaller than in Fortran on workstations and PCs, and *x_max* is significantly larger. Indeed, *x_min* and *x_max* are next to those of Cray. The wide range of numbers on MATLAB is indeed a significant advantage when exponential functions or functions with singularities are computed.

If the reader would like to verify *x_min*, *x_max*, and *eps*, run the following scripts (the last number appearing on the screen is the answer):

List 1.15

```
% To find x_min
x=1; while x>0, x=x/2, end
```

List 1.16

```
% To find x_max
x=1; while x<inf, x=x*2, end
```

[4]Machine epsilon is the difference between 1 and the next floating value greater than 1. In other words, no floating value between 1 and 1+*eps* can be expressed. Any number that falls in between is, therefore, rounded to 1 or 1+*eps*. One half of the machine epsilon can be interpreted as a maximum relative rounding error associated with a floating value. Obviously, the smaller the *eps*, the more accurate the numbers in the computing environment.

[5]S. Nakamura, *Applied Numerical Methods in C*, Prentice-Hall, 1992.

List 1.17

```
% To find machine epsilon
x=1;  while x>0,   x=x/2; ex = x*0.98 + 1; ex=ex - 1;
        if ex > 0, ex, end
      end
```

If a value becomes greater than *x_max*, the number is treated (in MATLAB) as ∞, denoted by `inf`. If you type `inf` on the command window, the response is

```
ans =
      inf
```

Typing `x = 1/inf` will yield

```
ans =
      0
```

Sometimes, however, the answer becomes `NaN`, which means *not a number*. For example, if you try to compute `i*inf`, the answer of MATLAB is

```
ans =
      NaN
```

1.8 MATHEMATICAL FUNCTIONS OF MATLAB

Like any other programming language, MATLAB has numerous mathematical functions ranging from elementary to high levels. Elementary functions may be classified into the following three categories:

(a) Trigonometric functions
(b) Other elementary functions
(c) Functions that do chores

Table 1.3 shows the functions in the first two categories. The functions in the third category are explained in Section 1.9.

Mathematical functions in MATLAB have two distinct differences from those in other programming languages such as Fortran and C: (1) mathematical functions work for complex variables without any discrimination, and (2) mathematical functions work for vector and matrix arguments.

complex arguments: To show how the functions of MATLAB work for imaginary or complex variables, let us try

```
cos(2 + 3*i)
```

where `i` equals the unit imaginary number, or equivalently square root of -1. Then the answer is

Table 1.3 Elementary Mathematical Functions

Trigonometric functions	Remarks
`sin(x)`	
`cos(x)`	
`tan(x)`	
`asin(x)`	
`acos(x)`	
`atan(x)`	$-\pi/2 \geq \text{atan}(x) \geq \pi/2$
`atan2(y,x)`	Same as `atan(y/x)` but $-\pi \geq \text{atan}(y, x) \geq \pi$
`sinh(x)`	
`cosh(x)`	
`tanh(x)`	
`asinh(x)`	
`acosh(x)`	
`atanh(x)`	
Other elementary functions	**Remarks**
`abs(x)`	Absolute value of x
`angle(x)`	Phase angle of complex value: If x = real, angle = 0 If $x = \sqrt{-1}$, angle = $\pi/2$
`sqrt(x)`	Square root of x
`real(x)`	Real part of complex value x
`imag(x)`	Imaginary part of complex value x
`conj(x)`	Complex conjugate x
`round(x)`	Round to the nearest integer
`fix(x)`	Round a real value toward zero
`floor(x)`	Round toward $-\infty$
`ceil(x)`	Round x toward $+\infty$
`sign(x)`	+1 if $x > 0$; -1 if $x < 0$
`mod(x,y)`	Remainder upon division: x - y*fix(x/y)
`rem(x,y)`	Remainder upon division: x - y*fix(x/y). Different from `mod` if $y \leq 0$
`exp(x)`	Exponential base e
`log(x)`	Log base e
`log10(x)`	Log base 10
`factor(x)`	Factorize x into prime numbers
`isprime(x)`	1 if x is a prime number, 0 if not
`factorial(x)`	$x!$

```
ans =
     -4.1896 - 9.1092i
```

For another example, consider the arccosine function, which is the inverse of the cosine function defined by

$$y = \text{acos}(x) = \cos^{-1}(x)$$

The command

```
acos(0.5)
```

yields

```
ans =
       1.0472
```

The argument x in $\text{acos}(x)$ is ordinarily limited to the range $-1 \leq x \leq 1$ (this is the way `acos` function works in Fortran). In MATLAB, however, `acos` accepts any value in $-\infty < x < \infty$ because the values of $\text{acos}(x)$ are not restricted to real values. Indeed, if we try

```
acos(3)
```

then

```
ans =
      0 + 1.7627i
```

array arguments: Most functions in MATLAB can take vectors and matrices as argument. For example, if

```
x =
     1     2     3
     9     8     7
```

then `sin(x)` will yield

```
ans =
    0.8415    0.9093    0.1411
    0.4121    0.9894    0.6570
```

which is a matrix of the same size as `x`. The computation performed here is equivalent to

List 1.18

```
for i=1:2
  for j=1:3
    x(i,j) = sin(x(i,j))
  end
end
```

If `x` is a column or row array, `sin(x)` becomes a column or row array accordingly.

1.9 FUNCTIONS THAT DO CHORES

Besides functions that compute straightforward mathematical functions listed in Table 1.3, there are several functions that do chores.

sort: `sort` reorders elements of a vector to ascending order. This command is useful if data in a random order have to be reordered in ascending order. The argument `x` can be a row vector, column vector, or a matrix. If `x` is a matrix, reordering is performed for each column. A few examples are given here:

```
sort([2 1 5])
ans =
     1     2     5

sort([2 1 5]')
ans =
     1
     2
     5

sort([9 1 5; 2 8 4])
ans =
     2     1     4
     9     8     5
```

sum: `sum(x)` computes the summation of the elements of a vector or matrix `x`. For both a column vector or a row vector, `sum` computes the total of the elements. If `x` is a matrix, the sum of each column is computed and a row vector consisting of the summation of each column is returned. A few examples are given here:

```
sum([2 1 5])
ans =
     8

sum([2 1 5]')
ans =
     8

sum([2 1 5; 9 8 5])
ans =
    11     9    10
```

max, min: `max(x)` finds the maximum in vector `x`, and `min(x)` finds the minimum. Argument `x` can be a row or column vector, or a matrix. If `x` is a matrix, the answer is a row vector containing the maximum or minimum of each column of `x`. (The rule is the same as that for `sort` and `sum`.)

mod: `mod(x,y)` returns the remainder of division of `x` by `y`. Therefore, the returned values becomes zero if `x` is divisible by `y`. In other words, `mod(x,y)` becomes zero if `x` is an integer multiple of `y`.

rand: Random numbers can be generated by `rand`. Its synopsis is `rand(n)`, where `n` specifies the size of the matrix of random numbers to be returned. If `n` $= 1$, a single random number is returned. For `n` > 1, a $n \times n$ matrix of random numbers is returned. Unless otherwise specified, the random numbers generated in this way are in $0 \leq x \leq 1$. If `rand` is called repeatedly, a sequence of random numbers is generated. The random number generator may be initialized by giving a seed number. The synopsis of initialization is

```
rand('seed', k)
```

where `k` is the seed number. It must be greater than unity. When the seed number is the same, the sequence of the random numbers becomes the same. If, however, the sequence is desired to be randomly different whenever the random generator is started, a randomly chosen seed number must be given. It could be the pollen count of the day, or the time in seconds, or a number drawn at a state lottery during the week, although finding a truly random number from natural phenomena or our daily life is not easy (see Example 1.2).

eval: Commands can be edited as a string and then executed by `eval`. The string can be read as input, or created within a script. For example

```
x=0:0.1:10
string=input('Type a function name, and hit return:  ','s');
s=['plot(', 'x,',string,'(x))'];
eval(s)
```

If the input is `cos`, for example, the script plots `y=cos(x)` for `x=0:0.1:10`.

1.10 DEVELOPING A PROGRAM AS AN M-FILE

Executing commands from a window is suitable only if the amount of typing is small, or if you want to explore ideas interactively. When commands are more than a few lines long, however, the user should write a script m-file, or a function m-file, because the m-files are saved to disk and can be corrected as many times as needed.[6] The m-file can include anything the user would write directly in the command window. Beginners should try to develop short m-files first and execute them.

In MATLAB-6, an m-file can be executed from the editor window. Click `Save and run` in the *Debug* menu.

echo on, echo off: When an m-file is executed, the statements in the m-file are not usually printed on the screen. After echo is turned on with the `echo on` command, however, the statements are printed out. By doing this, the user can see which part of the m-file is being executed. To turn off echo, type `echo off`.

comment statements: The percent sign in m-files indicates that any statements after this sign on the same line are comments and are to be ignored for computations. Comments added to m-files in this way can help explain the meaning of variables and statements.

Example 1.5

Random numbers may be used to play a game. The `x=rand(1)` command generates a random number between 0 and 1 and sets `x` to that

[6] m-files are classified into two categories: script m-file and function m-file. Script corresponds to a main program in traditional programming languages, while function corresponds to subprogram, subroutine, or function in traditional languages.

number. Consider 13 spade cards which have been well shuffled. The probability of picking up one particular card from the stack is 1/13. Write a program to emulate the action of picking up one spade card by a random number. The game is to be continued by returning the card to the stack and shuffling again after each game is over.

Solution

Suppose the interval between 0 and 1 is divided into 13 equally spaced subintervals. Each is defined by $(n-1)/13 < x < n/13$ where $n = 1, 2, ..13$. Then, if a random number falls in the nth interval, we can say that the nth card is drawn. Actually, n can be found by multiplying x by 13 and rounding up to the nearest higher integer.

Of course, before using `rand`, we have to initialize `rand` with a seed number. If the same seed is used, an identical sequence of random numbers is generated. One way to pick up a seed number is to use the `clock` command. For example, `c=clock` will set `c` to a row vector of length 6. The product of the second through the last numbers, namely `c(2)*c(3)*c(4)*c(5)*c(6)`, has approximately 3×10^7 combinations and changes every second throughout one year.

The following m-file determines a card every time it is executed. The game is repeated by answering the prompted question by `r`, but is terminated by typing any letter other than `r`. This m-file is saved by `List1_19.m`, so it can be executed from the command window by typing `List1_19`.

List 1.19

```
c=clock;
k=c(2)*c(3)*c(4)*c(5)*c(6);
rand('seed', k)
for k=1:20
  n=ceil(13*rand(1));
  fprintf('Card number drawn:   %3.0f\n', n)
  disp(' ')
  disp('Type r and hit Return to repeat')
  r = input('or any letter to terminate  ','s');
  if r ~= 'r', break, end
end
```

One interesting but useful feature of an m-file is that it can call other m-files. The calling m-file is a parent m-file, while the called m-files are children m-files. This implies that one script may be broken into one parent m-file

and multiple children m-files. The children m-files are similar to function m-files, which are explained in the next section. The difference, however, is that the parent and children m-files can see all the variables among them while function m-files can see only those variables given through arguments.

1.11 HOW TO WRITE YOUR OWN FUNCTIONS

Functions in MATLAB, which are saved as separate m-files, are equivalent to subroutines and functions in other languages.

function that returns only one variable: Let us consider a function m-file for the following equation:

$$f(x) = \frac{2x^3 + 7x^2 + 3x - 1}{x^2 - 3x + 5e^{-x}} \tag{1.11.1}$$

Assuming the m-file is saved as `demof_.m`, its script is illustrated by

List 1.20

```
function y = demof_(x)
y = (2*x.^3+7*x.^2+3*x-1)./(x.^2-3*x+5*exp(-x));
```

Notice that the name of the m-file is identical to the name of the function, which appears on the right side of the equality sign. In the m-file, array arithmetic operators are used, so the argument `x` can be a scalar as well as a vector or matrix. Once `demof_.m` is saved as an m-file, it can be used in the command window or in another m-file. The command

```
y = demof_(3)
```

yields

```
y =
    502.1384
```

If the argument is a matrix, for example,

```
demof_([3,1; 0, -1])
```

the result becomes a matrix also:

```
ans =
  502.1384  -68.4920
   -0.2000    0.0568
```

function that returns multiple variables: A function may return more than one variable. Suppose a function that evaluates mean and standard deviation of data. To return the two variables, a vector is used on the left side of the function statement, for example,

List 1.21

```
function [mean,stdv] = mean_st(x)
n=length(x);
mean = sum(x)/n;
stdv = sqrt(sum(x.^2)/n - mean.^2);
```

To use this function, the left side of the calling statement should also be a vector. The foregoing script is to be saved as `mean_st.m`. Then,

```
x = [ 1 5 3 4 6 5 8 9 2 4];
[m, s] = mean_st(x)
```

yields

```
m =
    4.7000
s =
    2.3685
```

function that uses another function: The argument of a function may be the name of another function. For example, suppose a function that evaluates a weighted average of a function at three points as

$$f_{av} = \frac{f(a) + 2f(b) + f(c)}{4} \tag{1.11.2}$$

where $f(x)$ is the function to be named in the argument. The following script illustrates a MATLAB function `f_av.m` that computes Eq.(1.11.2):

List 1.22

```
function wa = f_av(f_name, a, b, c)
wa = (feval(f_name,a) + 2*feval(f_name,b) ...
                          + feval(f_name,c))/4;
```

In the foregoing script, `f_name` (a string variable) is the name of the function $f(x)$. If $f(x)$ is the `sine` function, `f_name` equals `'sin'`. The `feval(f_name,x)` is a MATLAB command that evaluates the function named `f_name` for the argument `x`. For example, `y = feval('sin',x)` becomes equivalent to `y=sin(x)`.

Example 1.6

Evaluate Eq.(1.11.2) for the function defined by Eq.(1.11.1) with $a = 1, b = 2$, and $c = 3$. Equation (1.11.1) has been written as `demof_.m` in List 1.20.

Solution

We assume `f_av.m` (List 1.22) has been saved as an m-file. Then, the command

```
A = f_av('demof_', 1, 2, 3)
```

yields

```
89.8976
```

The number of input and output arguments of `feval` must be consistent with the input and output format of the function `f_name`. For example, if the function `f_name` needs four input variables and returns three output variables, the statement to call `feval` would be:

```
[p, q, s] = feval( f_name, u, v, w, z)
```

where `p`, `q`, and `s` are output, while `f_name`, `u`, `v`, `w`, and `s` are input.

debugging of function m-files: Debugging function m-files is more difficult than script m-files. One reason is that you cannot see the values of variables by typing the variable names unless debugging commands are used. The most basic but effective method of developing a function m-file is to comment-out the function statement on the first line by placing `%` before `function` and then test the m-file as a script m-file. Put the function statement back after a thorough examination of the m-file.

Using debugging commands is recommended only for advanced MATLAB users.

1.12 SAVING AND LOADING DATA

save, load: If `save` is used by itself, as in

```
save
```

all the variables are saved in the default file `matlab.mat`. The `load` command is the inverse of the `save` command and retrieves all the variables saved by `save`.

The filename may be specified by placing it after `save`; for example:

```
save file_name
```

saves all the variables in the file named `file_name.mat`. When you wish to retrieve the variables, write

```
load file_name
```

If only selected variables are to be saved, write the variable names after `file_name`; for example:

```
save file_name a b c
```

In this example, `a`, `b`, and `c` are saved in the file named `file_name`. Do not separate `file_name` and variables by a comma. All the variables are saved in double precision binary. When you wish to load the data in `file_name.mat`, type

```
load file_name
```

without variable names. Then, all of `a`, `b`, and `c` are retrieved.

save filename data -ascii: `save` can be used to save data in ASCII format. Both `save` and `load` with the ASCII option are important because they make possible export and import of data from MATLAB.

To use the ASCII format, `-ascii` or `/ascii` is appended after the variable names. For example,

```
save data.tmp  x  -ascii
```

saves variable `x` in 8-digit ASCII to the file named `data.tmp`. The `save` command can save more than one variable; for example,

```
x = [1, 2, 3, 4 ]
y = [-1, -2, -3]'
save data2.tmp  x  y  -ascii
```

If you open the `data2.tmp` file, it looks like

```
 1.0000000e+00  2.0000000e+00  3.0000000e+00  4.0000000e+00
-1.0000000e+00
-2.0000000e+00
-3.0000000e+00
```

The `load` command allows you to convert a data file into a variable, but loading a file in ASCII format is not quite the inverse of `save` in ASCII format. The reason is that while `save` in ASCII can write multiple variables, `load` reads the entire data file into only one variable. Furthermore, the filename becomes the variable's name. For example, if a file named `y_dat.e` is loaded by

```
load y_dat.e
```

the content is loaded to the variable named `y_dat` regardless of the extension name.

Therefore, the data file `y_dat` must be in one of the following data forms only:

(a) a single number
(b) a row vector
(c) a column vector
(d) a matrix

If multiple variables have to be loaded, each variable should be prepared in a separate ASCII data file.

Data files prepared by Fortran or C in ASCII (or text) format can be loaded by `load` as long as the data structure is one of the four forms. For more advanced methods of exporting and importing data files, consult the MATLAB User's Guide.

creating file name automatically: it often becomes desirable to create filenames automatically within an m-file. If a whole command, including the filename, is written as a string, it may be executed by `eval`. In the following script, `xdata` is assumed to be computed for each `k` and saved in separate files named fname001, fname002, ... in ASCII format.

```
for k=1:kmax
  %Here are some statements to produce xdata for each k.
  %kmax is the maximum number of k (less than 1000).
  if k<10, s=['save fname00',num2str(k), ' xdata -ascii']
  elseif k>=10 & k<100, s=['save fname0',num2str(k), ' xdata -ascii']
  elseif k>=100, s=['save fname',num2str(k), ' xdata -ascii']
  eval(s)
end
```

1.13 HOW TO MAKE HARD COPIES

One frequently asked question is how to make hard copies of the prints on the screen. To produce a copy of prints on the screen, use `diary` introduced in Section 1.1. If `diary` is used without a specific filename, the filename becomes `diary` in the directory. The file can be printed out as a text file. Graphic figures are not captured in `diary`.

PROBLEMS

In resolving the problems that follow, prepare your answers using MATLAB. Run the statements or scripts on MATLAB. Make sure you correct your script until you get the right answers. If you are required to submit your answers for the problems, do not submit the printout without attaching a short summary by hand in case your instructors have trouble understanding it. Highlight the key numbers (final answers) in the printout so the instructor can find them without a struggle. If the problems are difficult and you need assistance, look at similar problems and answers given in *Additional Exercises* at http://olen.eng.ohio-state.edu/matlab

(1.1) Guess the MATLAB response to the following statements. Examine your answers by executing them on MATLAB.

```
a = [1 2 3; 4 5 6]'
b = [9;7;5;3;1]
c = b(2:4)
d = b(4:-1:1)
e = sort(b)
f = [3,b']
```

(1.2) Arrays `x` and `y` are defined by

```
x=[7 4 3]
y=[-1 -2 -3]
```

(a) Define a new array `u` by prepending `y` to `x`.
(b) Define a new array `v` by appending `y` to `x`.

(1.3) On MATLAB, make a two-dimensional array such that the first row equals `x` and the second row equals `y`, where both `x` and `y` are defined in the previous problem.

(1.4) Eliminate the `for/end` loop in the following script by using the array arithmetic operators:

```
x=11:15
for k=1:length(x)
z(k)=x(k)^2 + 2.3*x(k)^0.5;
end
z
```

(1.5) Rewrite the following script without using the array arithmetic operators:

```
x=[4 1 2]
z=1./(1 + x.^2)
```

(1.6) A vector is given by

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

Write a script that doubles the negative numbers in `a`. Run the script and show your output.

(1.7) Write a script that removes positive numbers from `a` given in Problem (1.6). Run the script and show your printout of the answer.

(1.8) Write a script that generates an array such as

`x=[1 4 2 4 2 4 .. 4 1]`

where the length of the array is an odd number, the numbers in the even addresses are 4, and the numbers in the odd addresses are 2 (except the first and the last numbers are 1). Run your script and show how it works for any odd length of `x`, given by input, which is equal to 3 or greater.

(1.9) Write a script that generates an array such as

`x=[1 3 3 2 3 3 2 3 3 .. 2 3 3 1]`

where the length of the array is `3n+1` with an integer `n`, `3` always repeats two times following `2` except for the first number `1`, and the last number is `1`. Run your script and show how it works for any `n`.

(1.10) A vector is given by

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

Write a script that calculates the sum of positive numbers of `a`. Run the script and show your printout of the answer.

(1.11) The pricing scheme of computer printers sold by a company is as follows:

(a) the basic price of a printer is \$150 if the customer buys only one printer

(b) the second printer is \$120 if the customer buys more than 1 printer

(c) the third printer and beyond is \$110.

Write a script that calculates the total price for up to 10 printers purchased by one customer.

The number of printers should be specified at the beginning of the script. The number of printers may be read by the input command. Calculate the total price for 10 printers by running your script.

(1.12) Develop a script that generates a price chart for up to 10 printers for the printer store described in the previous problem. (Use the `fprintf` command to print out the table. Utilizing a price table recalculated by hand calculations is not acceptable.)

(1.13) A company selling printer cartridges has the following price policy. The first cartridge is $50, the second is $35, but the third one is free. This pricing pattern repeats as the number of cartridges sold increases, namely, the fourth is $50, the fifth is $35 and the sixth is free, and so on. Most customers buy multiples of three to take advantage of every third. However, large organizations like universities and government agencies buy the exact number of cartridges they need regardless of the advantage of the free phurchase. Thus, the company must prepare a price chart for any number of cartridges sold. Write a script to prepare a price chart for up to 20 cartridges. Make sure the output of your chart is correct by hand calculations.

(1.14) Bubble sort is an iterative method for reordering a sequence of numbers in increasing order. For example, consider `x=[7 1 2 4 8 5]`. In the first iteration cycle, 7 in the first position and 1 in the second position are compared. Since 7 is greater than 1, they are exchanged in the array. Next, 7, which is now in the second position, is compared to 2 in the third position. Then an exchange takes place because 7 is greater than 2. The same procedure is repeated until the last position is involved. The second cycle is the same using the result of the first cycle. The cycles are repeated until no exchange becomes necessary. Write a script to reorder `x` in increasing order. Do not use the `sort` command.

(1.15) A matrix `m` is defined by

$$m = \begin{bmatrix} 1 & 2 & 5 \\ 3 & 1 & 2 \\ 4 & 1 & 3 \end{bmatrix}$$

How do you read `m` into MATLAB by the `input` statement? Show that your procedure is correct on MATLAB.

(1.16) What is the result of `sum(m)`, `max(m)`, and `min(m)` on MATLAB, where `m` is the matrix defined in the previous problem?

(1.17) Write an m-file that examines if a positive integer is a prime number or not without using `factor` or `isprime`.

(1.18) Array `x` is given by `x=[1:99]`. Remove all prime numbers from `x`, then calculate the sum.

(1.19) Write a script to compute

$$S = \sum_{n=1}^{10} \frac{n}{n+1}$$

(a) using a `for/end` loop, but not array operators or `sum`.
(b) using array operators and `sum`, but not any `for/end` loop.
What is the value of S?

(1.20) Write a script that converts any element equal to 1 in a matrix to -1. Run the m-file for the 3×3 array (matrix) `m` defined in Problem 1.15. Print out your script and its results.

(1.21) Define `v` by

```
v = 'glacier'
```

or

```
v = ['g', 'l', 'a', 'c', 'i', 'e', 'r']
```

Write a script to find the address of `i` in `v`.

(1.22) A matrix (or array) generated by `m=rand(4,4)` is an array of random numbers. Write a script that prints out the addresses (row and column numbers) of the numbers that are less than 0.5. Execute your script and make sure it is correct. Print out your script and the answers.

(1.23) Develop a function m-file, `fun_es(x)`, to compute the following function:

$$y = 0.5e^{x/3} - x^2 \sin x$$

The argument must accept a scalar as well as a vector. Test your function by typing the following on MATLAB:

```
fun_es(3)
fun_es([1 2 3])
```

(1.24) Repeat the task of Problem (1.23) for the function:

$$y = \sin(x)\log(1+x) - x^2, \quad x > 0$$

Denote the function `fun_lg(x)`.

(1.25) (a) Write a function m-file that calculates the solution of

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

Its synopsis is `quad_rt(a,b,c)`, where `a`, `b`, and `c` are allowed to be vectors.
(b) Test the function for `a=3, b=1, c=1`.
(c) Test the function for `a=[3 1 2]`, `b=[1 -4 9]`, `c=[1 3 -5]`.

(1.26) The reader is assumed to have completed `fun_es` and `fun_lg` developed for Problems 1.23 and 1.24. Now, develop a function, `t_es(x)`, that:
(a) asks for the name of the function to be evaluated
(b) lets the user type the function name
(c) evaluates the function by `feval` and returns the functional values
(d) stops if the choice is neither `fun_es` nor `fun_lg`.

Test your `t_es` by asking to evaluate `fun_es(3)` and `fun_lg(3)`.

(1.27) Two variables, `x` and `y`, are saved in the `out_asc.m` file:

```
x = 1:5
y = [-1:-1:-5]'
save out_asc x y -ascii
```

How does the file look when the file is opened as an m-file? Is it possible to read both `x` and `y` from the same file? If `x` and `y` have to be saved in ASCII format and also have to be loaded later, what should you do?

(1.28) A vector is given:

```
A = [1 2 3 4 5 6 7 8 9 0]
```

Write a script to print out the vector content using the `fprintf` command in a loop such that the printout will look like:

```
Vector A is
[  1, 2, 3, 4, 5, 6, 7, 8, 9, 0]
Print completed.
```

Note that each number except the last is followed by a comma and one blank space. Your script should work for any other definition of vector `A`.

(1.29) Write an m-file that asks the player to type 0 or 1. If 1, the software finds 10 numbers randomly from 1 through 6. Print out the 10 numbers found in a row vector form. If the player's input is 0, the program is stopped. Obtain a seed random number by `clock` as follows:

```
c = clock;
sdnum = c(1)*c(2)*c(3)*c(4)*c(5)*c(6);
```

(1.30) Develop an m-file named `fun_xa` that evaluates the following series:

$$f(x) = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + .. + \frac{x^n}{n!}$$

The values of x and n are to be given by `input`. Test the function by comparing the result with hand calculations for $x = 1$ and $n = 4$. The foregoing series is a truncated Maclaurin expansion of e^x and converges for $-\infty < x < \infty$. Knowing this, test your function for selected x values, such as $x = 0.5, 3.0$ and -1, with $n = 1, 2, 3, 5, 10$, and compare with e^x.

(1.31) Develop an m-file named `fun_xb` that evaluates the following series:

$$f(x) = x - \frac{x^2}{2} + \frac{x^3}{3} - .. + (-1)^{n+1}\frac{x^n}{n}$$

The values of x and n are to be given by `input`. Test the function by comparing the result with hand calculations for $x = 1$ and $n = 4$. The foregoing series is a truncated Maclaurin expansion of $\log(1+x)$ and converges for $-1 < x < 1$. Knowing this, test your function for selected x value, such as $x = -0.5$ and 0.5, with $n = 1, 2, 3, 5, 10, 20, 50$, and compare with $\log(1+x)$. (Convergence becomes increasingly difficult as x approaches -1 or 1.)