

**MATLAB**  
**BEGINNER'S**  
**GUIDE**

## About MATLAB

MATLAB is an interactive software which has been used recently in various areas of engineering and scientific applications. It is not a computer language in the normal sense but it does most of the work of a computer language. Writing a computer code is not a straightforward job, typically boring and time consuming for beginners. One attractive aspect of MATLAB is that it is relatively easy to learn. It is written on an intuitive basis and it does not require in-depth knowledge of operational principles of computer programming like compiling and linking in most other programming languages. This could be regarded as a disadvantage since it prevents users from understanding the basic principles in computer programming. The interactive mode of MATLAB may reduce computational speed in some applications.

The power of MATLAB is represented by the length and simplicity of the code. For example, one page of MATLAB code may be equivalent to many pages of other computer language source codes. Numerical calculation in MATLAB uses collections of well-written scientific/mathematical subroutines such as LINPACK and EISPACK. MATLAB provides Graphical User Interface (GUI) as well as three-dimensional graphical animation.

In general, MATLAB is a useful tool for vector and matrix manipulations. Since the majority of the engineering systems are represented by matrix and vector equations, we can relieve our workload to a significant extent by using MATLAB. The finite element method is a well-defined candidate for which MATLAB can be very useful as a solution tool. Matrix and vector manipulations are essential parts in the method. MATLAB provides a *help* menu so that we can type the *help* command when we need help to figure out a command. The *help* utility is quite convenient for both beginners and experts.

## Vector and Matrix Manipulations

Once we get into MATLAB, we meet a prompt » called the MATLAB prompt. This prompt receives a user command and processes it providing the output on the next line. Let us try the following command to define a matrix.

```
>> A=[1,3,6;2,7,8;0,3,9]
```

Then the output appears in the next line as shown below.

```
A =  
    1    3    6  
    2    7    8  
    0    3    9
```

Thus, a matrix is entered row by row, and each row is separated by the semicolon(;). Within each row, elements are separated by a space or a comma(.). Commands and variables used in MATLAB are case-sensitive. That is, lower case letters are distinguished from upper case letters. The size of the matrix is checked with

```
>> size(A)
```

```
ans = 3  3
```

**Transpose of a matrix** In order to find the transpose of matrix  $A$ , we type

```
>> A'
```

The result is

```
    1    2    0  
ans = 3    7    3  
    6    8    9
```

**Column or row components** MATLAB provides columnwise or rowwise operation of a matrix. The following expression

```
>> A(:,3)
```

yields

```
ans =
```

6

8

9

which is the third column of matrix  $A$ . In addition,

```
>>A(:,1)
```

represents the first row of  $A$  as

```
ans = 1 3 6
```

We can also try

```
>> A(:,1)+A(:,3)
```

as addition of the first and third rows of  $A$  with the result

```
ans= 1 6 15
```

Now let us introduce another matrix  $B$  as

```
>> B = [3,4,5; 6,7,2;8,1,0];
```

Then there seems to be no output on the screen. MATLAB does not prompt output on the screen when an operation ends with the semicolon (;).

If we want to check the  $B$  matrix again, we simply type

```
>>B The screen output will be
```

```
      3  4  5
B =   6  7  2
      8  1  0
```

**Matrix addition** Adding two matrices is straightforward like

```
>> C = A + B
```

```
      4  7  11
C =   8  14  10
      8  4   9
```

**Matrix subtraction** In order to subtract matrix  $B$  from matrix  $A$ , we type

```
>> C = A-B
```

```
      -2  -1  1  
C = -4   0  6  
      - 8  2  9
```

Note that  $C$  is now a new matrix, not the summation of  $A$  and  $B$  anymore.

**Matrix multiplication** Similarly, matrix multiplication can be done as

```
>> C = A*B
```

```
      69  31  11  
C = 112  65  24  
      90  30   6
```

### Matrix Functions

Manipulation of matrices is a key feature of the MATLAB functions. MATLAB is a useful tool for matrix and vector manipulations. Collections of representative MATLAB matrix functions are listed in Table 1. Examples and detailed explanations are provided for each function below.

**Table 1.** Basic Matrix Functions

<b>Symbol</b>	<b>Explanations</b>
<b>inv</b>	inverse of a matrix
<b>det</b>	determinant of a matrix
<b>rank</b>	rank of a matrix
<b>cond</b>	condition number of a matrix
<b>eye(n)</b>	the n by n identity matrix
<b>zeros(n,m)</b>	the n by m matrix consisting of all zeros

**Matrix inverse** The inverse of a matrix is as simple as

```
>> inv(A)
```

```
ans =  
1.8571 -0.4286 -0.8571  
-0.8571 0.4286 0.1905  
0.2857 -0.1429 0.0476
```

In order to verify the answer, we can try

```
>> A*inv(A);
```

which should be a 3 by 3 identity matrix.

**Determinant of a matrix**

```
>> d = det(A)
```

produces the determinant of the matrix A. That is,  $d = 21$

**Rank of a matrix** The rank of a matrix A, which is the number of independent rows or columns, is obtained from

```
>> rank(A);
```

**Identity matrix**

```
>> eye(3)
```

yields

```
ans =  
1 0 0  
0 1 0  
0 0 1
```

eye(n) produces an identity matrix of size n by n. This command is useful when we initialize a matrix.

**Matrix of random numbers** A matrix consisting of random numbers can be generated using the following MATLAB function.

```
>> rand(3,3)
```

```
0.2190 0.6793 0.5194
```

```
ans = 0.0470  0.9347  0.8310
      0.6789  0.3835  0.0346
```

That is, `rand(3,3)` produces a 3 by 3 matrix whose elements consist of random numbers. The general usage is `rand(n, m)`.

**trace** Summation of diagonal elements of a matrix can be obtained using the trace operator.

For example,

```
>> C=[1 3 9; 672; 8 -1 -2];
```

Then, `trace(C)` produces 6, which is the sum of diagonal elements of *C*.

**zero matrix**

```
>> zeros(5,4)
```

produces a 5 by 4 matrix consisting of all zero elements. In general, `zeros(n,m)` is used for an *n* by *m* zero matrix.

**condition number** The command `cond(A)` is used to calculate the condition number of a matrix *A*. The condition number represents the degree of singularity of a matrix. An identity matrix has a condition number of unity, and the condition number of a singular matrix is infinity.

```
>>cond(eye(6))
```

```
ans = 1
```

An example matrix which is near singular is

```
A=[1 1;1 1+1e-6]
```

The condition number is

```
>>cond(A)
```

```
ans =
```

```
4.0000e+006
```

Further matrix functions are presented in Table 2. They do not include all matrix functions of the MATLAB, but represent only a part of the whole MATLAB functions. Readers can use the MATLAB Reference Guide or *help* command to check when they need more MATLAB functions.

**Table 2.** Basic Matrix Functions (Continued)

<b>Symbol</b>	<b>Explanations</b>
<b>expm</b>	exponential of a matrix
<b>eig</b>	eigenvalues/eigenvectors of a matrix
<b>lu</b>	LU decomposition of a matrix
<b>svd</b>	singular value decomposition of a matrix
<b>qr</b>	QR decomposition of a matrix
<b>\</b>	used to solve a set of linear algebraic equations

**Matrix exponential** The `expm(A)` produces the exponential of a matrix  $A$ . In other words,

```
>> A =rand(3, 3)
      0.2190 0.6793 0.5194
      A = 0.0470 0.9347 0.8310
          0.6789 0.3835 0.0346
>>expm(A)
      1.2448 0.0305 0.6196
      ans= 1.0376 1.5116 1.3389
          1.0157 0.1184 2.0652
```

**Eigenvalues** The eigenvalue problem of a matrix is defined as

$$A\phi = \lambda\phi$$

where  $\lambda$  is the eigenvalue of matrix  $A$ , and  $\phi$  is the associated eigenvector.

```
>> e =eig(A)
```

gives the eigenvalues of  $A$ , and

```
>> [V,D]=eig(A)
```

produces the  $V$  matrix, whose columns are eigenvectors, and the diagonal matrix  $D$  whose values are eigenvalues of the matrix  $A$ . For example,

```
>>A = [5 3 2; 146; 972];
```

```
>>[V,D]=eig(A)
```

```
    0.4127    0.5992    0.0459  
V = 0.5557   -0.7773   -0.6388  
    0.7217    0.1918    0.7680
```

```
    12.5361     0     0  
D =     0    1.7486     0  
     0     0   -3.2847
```

**LU Decomposition** The *LU* decomposition command is used to decompose a matrix into a combination of upper and lower triangular matrices.

```
>>A = [1 3 5; 248; 47 3];
```

```
>> [L, U]=l
```

```
    0.2500    1.0000     0  
L = 0.5000    0.4000    1.0000  
    1.0000     0     0
```

```
    4.0000    7.0000    3.0000  
U =     0    1.2500    4.2500  
     0     0    4.8000
```

In order to check the result, we try

```
>>L*U
```

```
    1    3    5  
ans = 2    4    8  
     4    7    3
```

The lower triangular matrix *L* is not perfectly triangular. There is another command

available

```
>> [L,U,P]=lu(A)
```

```
      1   0   0
L = 0.25  1   0
      0.5  0.4  1
      4.0000  7.0000  3.0000
U =  0   1.2500  4.2500
      0   0   4.8000
      0  0  1
P = 1  0  0
      0  1  0
```

Here, the matrix  $P$  is the permutation matrix such that  $P * A = L * U$ .

**Singular value decomposition** The `svd` command is used for singular value decomposition of a matrix. For a given matrix,

$$A = U\Sigma V'$$

where  $\Sigma$  is a diagonal matrix consisting of non-negative values. For example, we define a matrix  $D$  like

```
>> D = [1 3 7; 2 9 5 ; 2 8 5 ]
```

The singular value decomposition of the matrix is

```
>> [U, Sigma, V]=svd(D)
```

which results in

```
      0.4295  0.8998  -0.0775
U = 0.6629  -0.3723  -0.6495
      0.6133  -0.2276  0.7564
      15.649   0   0
Sigma =  20   4.133   0
          0   0   0.139
```

$$V = \begin{bmatrix} 0.1905 & -0.0726 & 0.9790 \\ 0.7771 & -0.5982 & -0.1956 \\ 0.5999 & 0.7980 & -0.0576 \end{bmatrix}$$

**QR decomposition** A matrix can be also decomposed into a combination of an orthonormal matrix and an upper triangular matrix. In other words,

$$A = QR$$

where  $Q$  is the matrix with orthonormal columns, and  $R$  is the upper triangular matrix. The  $QR$  algorithm has wide applications in the analysis of matrices and associated linear systems. For example,

$$A = \begin{bmatrix} 0.2190 & 0.6793 & 0.5194 \\ 0.0470 & -0.9347 & 0.8310 \\ 0.6789 & 0.3835 & 0.0346 \end{bmatrix}$$

Application of the **qr** operator follows as

```
>> [Q,R]=qr(A)
```

and yields

$$Q = \begin{bmatrix} -0.3063 & -0.4667 & -0.8297 \\ -0.0658 & -0.8591 & 0.5076 \\ -0.9497 & 0.2101 & 0.2324 \end{bmatrix}$$

$$R = \begin{bmatrix} 0.7149 & -0.6338 & -0.2466 \\ 0 & -1.0395 & -0.9490 \\ 0 & 0 & 0.0011 \end{bmatrix}$$

**Solution of linear equations** The solution of a linear system of equations is frequently needed in the finite element method. The typical form of a linear system of algebraic equations is written as

$$Ax = y$$

and the solution is obtained by

```
>>x =inv(A) * y
```

or we can use \ sign as

```
>>x = A\y
```

For example

```
>>A=[1 3 4; 5 7 8; 2 3 5];
```

and

```
>>y = [10; 9; 8];
```

Let us compare two different approaches.

```
>> [inv(^) * y A\y]
```

```
    -4.2500  -4.2500  
ans = 1.7500  1.7500  
    2.2500  2.2500
```

### Loop and Logical Statements

There are some logical statements available in MATLAB which help us in writing combinations of MATLAB commands. Furthermore, loop commands can be used as in other programming languages. In fact, we can duplicate the majority of existing programs using MATLAB commands, which significantly reduces the size of the source codes. A collection of loop and logical statements in MATLAB is presented in Table 3

**for loop** The **for** is a loop command which ends with end command.

```
>>for i = 1: 100
```

```
a(i, i) = 2 * i;
```

```
end
```

In the above example,  $i$  is a loop index which starts from 1 and ends at 100. There may be also multiple loops.

```
>> for i=1: 100
```

```
for j = 1 : 50
```

```
for k = 1 : 50
```

```
a(i,j) = b(i,k) *c(k,j) +a(i,j);
```

```
end
```

```
end
```

```
end
```

**Table 3** Loop and Logical Statements

<b>Symbol</b>	<b>Explanations</b>
<b>for</b>	loop command similar to other languages used
<b>while</b>	for a loop combined with conditional statement
<b>if</b>	produces a conditional statement
<b>elseif, else</b>	used in conjunction with if command
<b>break</b>	breaks a loop when a condition is satisfied

**while** The **while** command is useful for an infinite loop in conjunction with a conditional statement. The general synopsis for the **while** command is as follows:

**while** *condition*

*statements*

end

For example,

```
i= 1
```

```
while (i < 100)
```

```
i = i + 1; end
```

Another example of the while command is

```
n = 1000; var = [];
```

```
while (n > 0) n = n/2-1; var = [var,n];
```

```
end
```

The result is

```
var =
```

```
Columns 1 through 7
```

```
499.0000 248.5000 123.2500 60.6250 29.3125 13.6563 5.8281
```

```
Columns 8 through 9
```

```
1.9141 -0.0430
```

where we used [ ] in order to declare an empty matrix.

**if, elseif, else** The **if**, **elseif**, and **else** commands are conditional statements which are used in combination.

```
if condition #1 statement #1
elseif condition #2 statement
#2
else
statement #3
end
```

For example,

```
n = 100;
if (rem(n,3) == 0)
x = 0;
elseif (rem(n, 3) == 1)
x=1;
else
x=2;
end
```

,where **rem(x,y)** is used to calculate the remainder of  $x$  divided by  $y$ .

**Table 4** Loop and Logical Statements

Symbol	Explanations
==	two conditions are equal
~=	two conditions are not equal
<=>=)	one is less (greater) than or equal to the other
<>)	one is less (greater) than the other
&	<i>and</i> operator - two conditions are met
~	<i>not</i> operator
	<i>or</i> operator - either condition is met

**break** The **break** command is used to exit from a loop such as **if** and **while**. For example,

```
for i = 1: 100 f = i+1;
    if(i == 10) break;
end
end
```

**Logical and relational operators** The logical and relational operators of MATLAB are as listed in Table 4

The above command sets are used in combination.

### Writing Function Subroutines

MATLAB provides a convenient tool, by which we can write a program using collections of MATLAB commands. This approach is similar to other common programming languages. It is quite useful especially when we write a series of MATLAB commands in a text file. This text file is edited and saved for later use. .

The text file should have *filename.m* format normally called *m-file*. That is, all MATLAB subroutines should end with *.m* extension, so that MATLAB recognizes them as MATLAB compatible files. The general procedure is to make a text file using any text editor. If we generate a file called *func1.m*, then the file *func1.m* should start with the following file header

```
function[ov1,ov2, •••] = func1(iv1,iv2,...)
```

where  $iv_1, iv_2, \dots$  are input variables while  $ov_1, ov_2, \dots$  are output variables. The input variables are specific variables and the output variables are dummy variables, for which we can use any variables.

For example, let us solve a second order algebraic equation  $ax^2 + bx + c = 0$ . The solution is given in analytical form as

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

We want to write an *m-file* with the name *secroot.m*, which produces the analytical

solution.

```
function [r1,r2]=secroot(a,b,c); %  
% Find Determinant-- Any command in MATLAB which starts with  
% % sign is a comment statement  
Det= 6^2 - 4 * a * c;  
if (Det < 0),  
r1=- (-b + j* sqrt(-Det))/2/a;  
r2 = (-b -j* sqrt(-Det))/2/a;  
disp('The two roots are complex conjugates');  
elseif(Det == 0),  
r1 = -b/2/a;  
r2 = -b/2/a;  
disp('There are two repeated roots');  
else(Det > 0)  
r1 = (-6 + sqrt(Det))/2/a;  
r2 = (-b - sqrt(Det))/2/a;  
disp('The two roots are real');  
end
```

Some commands appearing in the above example will be discussed later. Once the *secroot.m* is created, we call that function as

```
>> [r1,r2]=secroot(3,4,5)
```

or

```
>> [p1,p2] =secroot(3,4,5)
```

One thing important about the function command is to set up the *m-file* pathname. The *m-file* should be in the directory which is set up by the MATLAB configuration set up stage. In the recent version of MATLAB, the set up procedure is relatively easy by simply adding a directory which we want to access in a MATLAB configuration file.

Another function subroutine *fct.m* is provided below.

```
function [f] =fct(x)

f= (1-x)^2;
```

The above function represents  $f(x) = (1-x)^2$ . In the MATLAB command prompt, we call the function as

```
>>y =fct(9);
```

The function subroutine utility of MATLAB allows users to write their own subroutines. It provides flexibility of developing programs using MATLAB.

## File Manipulation

Manipulating files is another attractive feature of MATLAB. We can save MATLAB workspace, that is, all variables used, in a binary file format and/or a text file format. The saved file can also be reloaded in case we need it later on. The list of file manipulation commands is presented in Table 5.

**Table 5** File Manipulation Commands

Symbol	Explanations
save	save current variables in a file
load	load a saved file into Matlab environment
diary	save screen display output in text format

**save** The **save** command is used to save variables when we are working in MATLAB. The synopsis is as follows

```
save filename var1 var2 ...
```

where *filename* is the filename and we want to save the variables, *var*<sub>1</sub> *var*<sub>2</sub> .... The filename generated by **save** command has extension of *.mat*, called a *mat-file*. If we do not include the variables name, then all current variables are saved automatically. In case we want to save the variables in a standard text format, we use

```
save filename var1 var2 .../ascii/double
```

**load** The **load** command is the counterpart of **save**. In other words, it reloads the variables in the file which was generated by **save** command. The synopsis is as follows

**load** *filename* *var<sub>1</sub>* *var<sub>2</sub>* ...

where *filename* is a *mat-file* saved by **save** command. Without the variables name specified, all variables are loaded.

For example,

```
>>a=[1 3 4];
>>6 = 3;
>>save test
>>clear all % clear all variables
>>who % display current variables being used
>>load test
>>who
```

**diary** Using **diary** command, we can capture all MATLAB texts including command and answer lines which are displayed on the screen. The texts will be saved in a file, so that we can edit the file later. For example,

```
>>diary on
>>a = 1; b = 4; c = 5;
>> [a b c]
>>d = a*b
>> e = g * h
>>diary off
```

Now we can use any text editor to modify the *diary* file. The **diary** command is useful displaying the past work procedures. Also, it can be used to save data in a text format.

### Basic Input-Output Functions

Input/output functions in MATLAB provide users with a friendly programming environment. Some input/output functions are listed in Table 6.

**Table 6** Input-Output Functions

Symbol	Explanations
input	save current variables in a file
disp	load a saved file into MATLAB
format	check the file status in the directory

**input** The **input** command is used to receive a user input from the keyboard. Both numerical and string inputs are available. For example,

```
>>age=input('How old are you?')
```

```
>>name=input('What is your name','s')
```

The 's' sign denotes the input type is string.

**disp** The **disp** command displays a string of text or numerical values on the screen. It is useful when we write a function subroutine in a user-friendly manner. For example,

```
>>disp('This is a MATLAB tutorial!')
```

```
>>c=3*4;
```

```
>> disp('The computed value of c turns out to be')
```

```
>> c
```

**format** The **format** command is used to display numbers in different formats. MATLAB calculates floating numbers in the double precision mode. We do not want to, in some situations, display the numbers in the double precision format on the screen. For a display purpose, MATLAB provides the following different formats

```
>> x = 1/9
```

```
x = 0.1111
```

```
>>format short e
```

```
z = 1.1111e-001
```

```
>>format long
```

```
1 = 0.1111111111111111
```

```
>>format long e
```

```
x = 1.1111111111111111e-001
```

```
>>format hex
```

```
x = 3/6c71c71c71c71c
```

## Plotting Tools

MATLAB supports some plotting tools, by which we can display the data in a desired format. The plotting in MATLAB is relatively easy with various options available. The collection of plotting commands is listed in Table 7.

A sample plotting command is shown below.

```
>> t = 0:0.1: 10;
>> y = sin(t);
>> plot(y)
>> title('plot(y)')
```

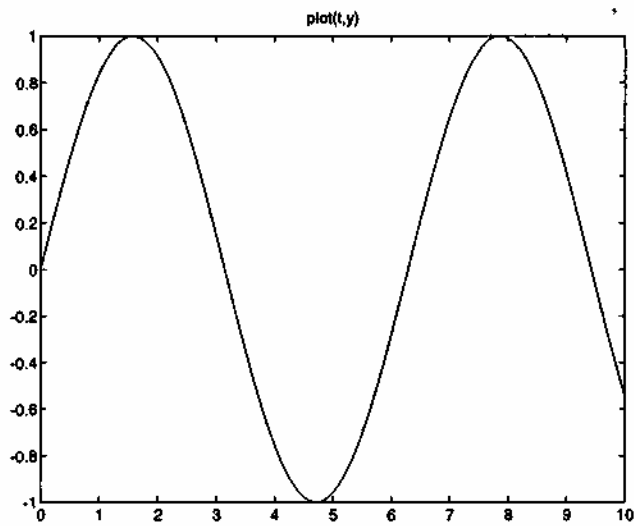
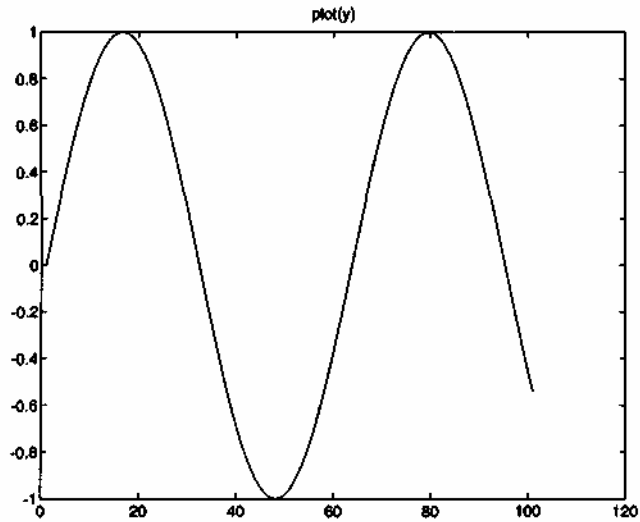
The resultant plot is presented at the top of Fig. 1.

```
>> t = 0 :0.1 : 10;
>> y = sin(t);
>> plot(t,y)
title('plot(t,y)')
```

**Table 7** Plotting Commands

Symbol	Explanations
<b>plot</b>	basic plot command
<b>xlabel(ylabel)</b>	attach label to x(y) axis
<b>axis</b>	manually scale x and y axes
<b>text</b>	place a text on the specific position of graphic
<b>title</b>	place a graphic title on top of the graphic
<b>ginput</b>	produce a coordinate of a point on the graphic
<b>gtext</b>	receives a text from mouse input
<b>grid</b>	add a grid mark to the graphic window
<b>pause</b>	hold graphic screen until keyboard is hit
<b>subplot</b>	breaks a graphic window into multiple windows

The resultant plot is presented at the bottom of Fig. 1. In the above example,  $t = 0 : 0.1 : 10$  represents a vector  $t$  which starts from 0 and ends at 10 with an interval of 0.1. We can use just  $y$  or both  $y$  and  $t$  together. In the first case, the horizontal axis represents number of data, from 0 to 101. In the second case, the horizontal axis is the actual time scale  $t$  hi the **plot(t,y)** command.



**Figure 1** A Sample Plot

**Plotting multiple data** We plot multiple data sets as shown below.

```
>> t = 0 : 1 : 100;
>> y1 = sin(t).*t;
>> y2 = cos(t).*t;
>> plot(t,y1,'-',t,y2,'-')
```

where '-' and '-' represent line styles. The line styles, line marks, and colors are listed in Table 8.

For example, if we want to plot data in a *dashed blue* line, the command becomes

```
>> plot(y,'-b');
```

**xlabel, ylabel** The `xlabel('text')` and `ylabel('text')` are used to label the  $x$  and  $y$  axes.

**axis** The `axis` command sets up the limits of axes. The synopsis is

```
axis[Xmin, Xmax, Ymin, Ymax]
```

**text** The `text` command is used to write a text on the graphic window at a designated point. The synopsis is

```
text(x,y,'text contents')
```

where  $x$  and  $y$  locate the  $(x,y)$  position of the `'text contents'`. A text can be also added in 3-D coordinates as shown below:

```
text(x, y,z,'text contents')
```

Table 8 Line, Mark, and Color Styles

Style	Line marks	Color
Solid	Point .	red r
dashed	star *	green g
dotted	circle o	blue b
dashdot	plus +	white w
	x-mark x	invisible i

**ginput** This command allows us to pick up any point on a graphic window. The synopsis is

```
[x,y] = ginput
```

We can pick as many points as we want on the graphic screen. The vector `[x,y]` then contains all the points.

**gtext** The `gtext` command is used to place text on the graphic window using the mouse input. The synopsis is

```
gtext('text')
```

Once the above command is entered or read in a function subroutine, the cursor on the graphic window is activated waiting for the mouse input, so that the `'text'` is located at the point selected by the mouse.

**grid** The grid command adds grids to the graphic window. It is useful when we want to clarify axis scales.

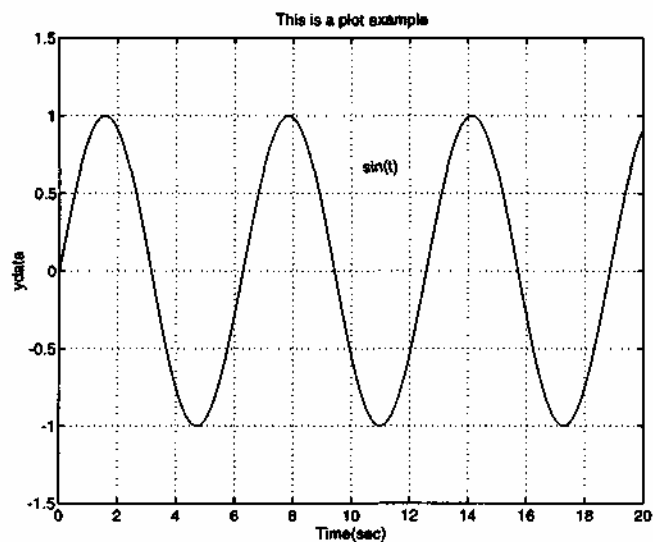
An example plot constructed using some of the commands described above is presented in Fig. 2. The following commands are used for the plot output.

```
>> t=0:0.1:20;  
>> plot(t,sin(t))  
>> xlabel('Time(sec)')  
>> ylabel('ydata')  
>> title('This is a plot example')  
>> grid  
>> gtext('sin(t)')  
>> axis([0 20 - 1.5 1.5])
```

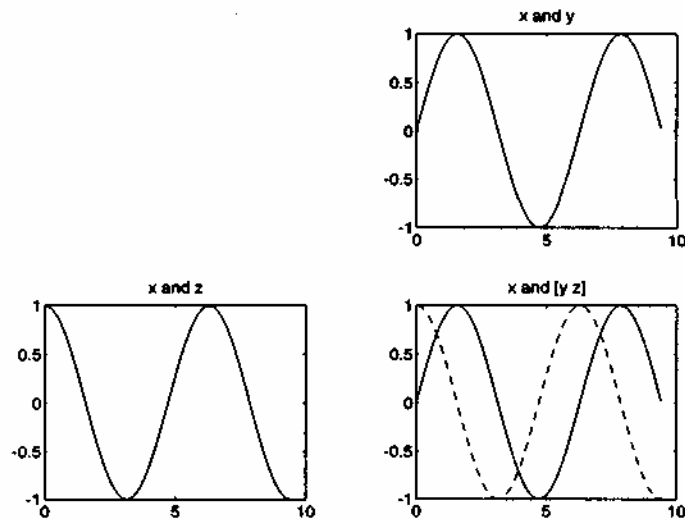
**pause** This command is useful when we display multiple graphic windows sequentially. It allows us to display one at a time with the keyboard interrupt.

**subplot** The subplot is used to put multiple plots on the same MATLAB figure window. The command is

```
>> subplot(pqr)
```



**Figure 2** A Plot Example With Some Commands



**Figure 3** A Subplot Example

The plot size is adjusted by a  $p$  by  $q$  matrix on the whole size of the graphic window. Then the third index  $r$  picks one frame out of the  $p$  by  $q$  plot frames. An example subplot is presented in Fig. 3 with the following commands entered.

```
>> x = 0 : 0.1 : 3 * pi; y = sin(x); z = cos(x);
>> subplot(222)
>> plot(x,y)
>> title('x and y')
>> subplot(223)
>> title('x and z')
>> subplot(224)
>> plot(x,y,'-',x,z,'--')
>> title('x and [y z]')
```

where  $pi$  is an internally defined variable equivalent to  $\pi$ .