
A Brief Introduction to the Matlab Programming Environment

Matlab is a program that features an interactive command window where a user can enter line commands to run programs or call specialized functions. The name Matlab came from "MAtrix LABoratory", and there is an extensive set of tools built in for conducting linear algebraic analysis. Matlab is a very valuable tool for doing numerical computations with matrices and vectors and displaying that information graphically. The basic data structure is the vector and matrix (reviewed below). These programs or functions may define or use numerical data that has been saved in the Matlab workspace. The ability to combine and call these programs and/or functions from within the Matlab environment allows for a very flexible computing environment. Once you are familiar with these examples, you can experiment with writing Matlab script files (called .m, "dot-m", files). These serve as Matlab programs and can be very powerful. There are many types of analysis functions that have been developed for scientific and engineering computing, and graphical input and output capabilities are continuously expanding. The best way to learn what Matlab can do is to work through some examples.

Getting Started with Matlab

Here is a sample session with Matlab. Text in bold is what you type, ordinary text is what the computer "types." You should read this example, then imitate it at the computer.

```
% matlab
```

```
>> a = [ 1 2; 2 1 ]
```

```
a =
```

```
 1  2  
 2  1
```

```
>> a*a
```

```
ans =
```

```
 5  4  
 4  5
```

```
>> quit
```

```
16 flops.
```

```
%
```

In this example you started Matlab by (you guessed it) typing **matlab**. Then you defined matrix **a** and computed its square ("a times a"). Finally (having done enough work for one day) you quit Matlab.

Matlab Tutorial Exercises

The tutorial below gives more examples of how to use Matlab. For best results, work them out using a computer: learn by doing!

Working with Vectors and Matrices

To enter the matrix

```
 1 2  
 3 4
```

and store it in a variable **a**, do this:

```
>> a = [ 1 2; 3 4 ]
```

To create the vector

```
1 2 3 4
```

and store it in variable **b**, do this:

```
>> b = [1 2 3 4]
```

note that there is no “;” between the square brackets (as in matrix **a**), which is what you use to define a matrix row separator.

Try this: Define the matrix a. Do the same with the examples below: work out each of them with matlab. Learn by doing!

To redisplay the matrix, just type its name:

```
>> a
```

Once you know how to enter and display matrices, it is easy to compute with them. First we will square the matrix *a* :

```
>> a * a
```

Wasn't that easy? Now we'll try something a little harder. First we define a matrix *b*:

```
>> b = [ 1 2; 0 1 ]
```

Then we compute the product *ab*:

```
>> a*b
```

Finally, we compute the product in the other order:

```
>> b*a
```

Notice that the two products are different: matrix multiplication is non-commutative.

Of course, we can also add matrices:

```
>> a + b
```

Now let's store the result of this addition so that we can use it later:

```
>> s = a + b
```

Matrices can sometimes be inverted:

```
>> inv(s)
```

To check that this is correct, we compute the product of s and its inverse:

```
>> s * inv(s)
```

The result is the unit, or identity matrix:

```
>> ans =  
    1.000 0.000  
    0.000 1.000
```

We can also write this as

```
>> s\s
```

which is the same as

```
>> inv(s) * s
```

To see that these operations, left and right division, are really different, we do the following:

```
>> a/b
```

Not all matrices can be inverted, or used as the denominator in matrix division:

```
>> c = [ 1 1; 1 1 ]  
>> inv(c)
```

A matrix can be inverted if and only if (iff) its determinant is nonzero:

```
>> det(a)  
>> det(c)
```

Here is another example: Enter each element of the vector (separated by a space) between brackets, and set it equal to a variable. For example, to create the vector *a*, enter into the Matlab command window (you can "copy" and "paste" from your browser into Matlab to make it easy):

```
>> a = [1 2 3 4 5 6 9 8 7]
```

Matlab should return:

```
a =
    1    2    3    4    5    6    9    8    7
```

Let's say you want to create a vector with elements between 0 and 20 evenly spaced in increments of 2 (this method is frequently used to create a time vector):

```
>> t = 0:2:20
```

```
t =
    0    2    4    6    8   10   12   14   16   18   20
```

Manipulating vectors is almost as easy as creating them. First, suppose you would like to add 2 to each of the elements in vector 'a'. The equation for that looks like:

```
>> b = a + 2
```

```
b =
    3    4    5    6    7    8   11   10    9
```

Now suppose, you would like to add two vectors together. If the two vectors are the same length, it is easy. Simply add the two as shown below:

```
>> c = a + b
```

```
c =
    4    6    8   10   12   14   20   18   16
```

Subtraction of vectors of the same length works exactly the same way.

Systems of Equations

Now consider a linear equation

$$ax + by = p$$

$$cx + dy = q$$

We can write this more compactly as

$$AX = B$$

where the coefficient matrix *A* is

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix}$$

c d

the vector of unknowns is

x

y

and the vector on the right-hand side is

p

q

If A is invertible, $X = (1/A)B$, or, using Matlab notation, $X = A \setminus B$. Lets try this out by solving $ax = b$ with a as before and $b = [1; 0]$. Note that b is a column vector.

```
>> b = [ 1; 0 ]
```

```
>> a\b
```

Basic Matrix Operations

Once you are able to create and manipulate a matrix, you can perform many standard operations on it.

To see the list of matrix functions available in Matlab, type **help matfun**.

For example, you can find the inverse of a matrix.

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

Warning: Matrix is close to singular or badly scaled.

Results may be inaccurate. RCOND = 4.565062e-18

ans =

1.0e+15 *

-2.7022 4.5036 -1.8014

5.4043 -9.0072 3.6029

-2.7022 4.5036 -1.8014

Note, the operations are numerical manipulations so in this example a matrix results even though matrix A has a determinant equal to zero. A warning is given to indicate that the condition number is very large (run **help cond**).

There are many other operations (see **help matfun**). For example, you can find maximum or minimum values in a matrix.

A very useful set of matrix functions for system analysis consists of finding eigenvalues and eigenvectors of a matrix. You can either find just eigenvalues, or both eigenvalues and eigenvectors. Use **help eig** for an explanation. For example,

```
>> eig(A)           (just eigenvalues)
```

```
ans =
```

```
14.0664  
-1.0664  
0.0000
```

```
>> [U,S] = eig(A)           (return both eigenvalues and eigenvectors)
```

```
U =
```

```
-0.2656  0.7444 -0.4082  
-0.4912  0.1907  0.8165  
-0.8295 -0.6399 -0.4082
```

```
S =
```

```
14.0664    0    0  
    0 -1.0664    0  
    0    0  0.0000
```

```
>> diag(S)
```

```
ans =
```

```
14.0664  
-1.0664
```

0.0000

Using Loops in Matlab

Finally, we will do a little piece of programming. Let a be the matrix

```
0.8 0.1
0.2 0.9
```

and let x be the column vector

```
1
0
```

We regard x as representing (for example) the population state of an island. The first entry (1) gives the fraction of the population in the west half of the island, the second entry (0) give the fraction in the east half. The state of the population T units of time later is given by the rule $y = ax$. This expresses the fact that an individual in the west half stays put with probability 0.8 and moves east with probability 0.2 (note $0.8 + 0.2 = 1$), and the fact that in individual in the east stays put with probability 0.9 and moves west with probability 0.1. Thus, successive population states can be predicted/computed by repeated matrix multiplication. This can be done by the following Matlab program:

```
>> a = [ 0.8 0.1; 0.2 0.9 ]
>> x = [ 1; 0 ]
>> for i = 1:20, x = a*x, end
```

What do you notice? Is there an explanation? Is there a lesson to be learned? NOTE: you have just learned to write a kind of loop, a so-called *for loop*. This is an easy way to command the machine, in just a few words, to do much repetitive work.

Plotting Data Using Matlab Graphics

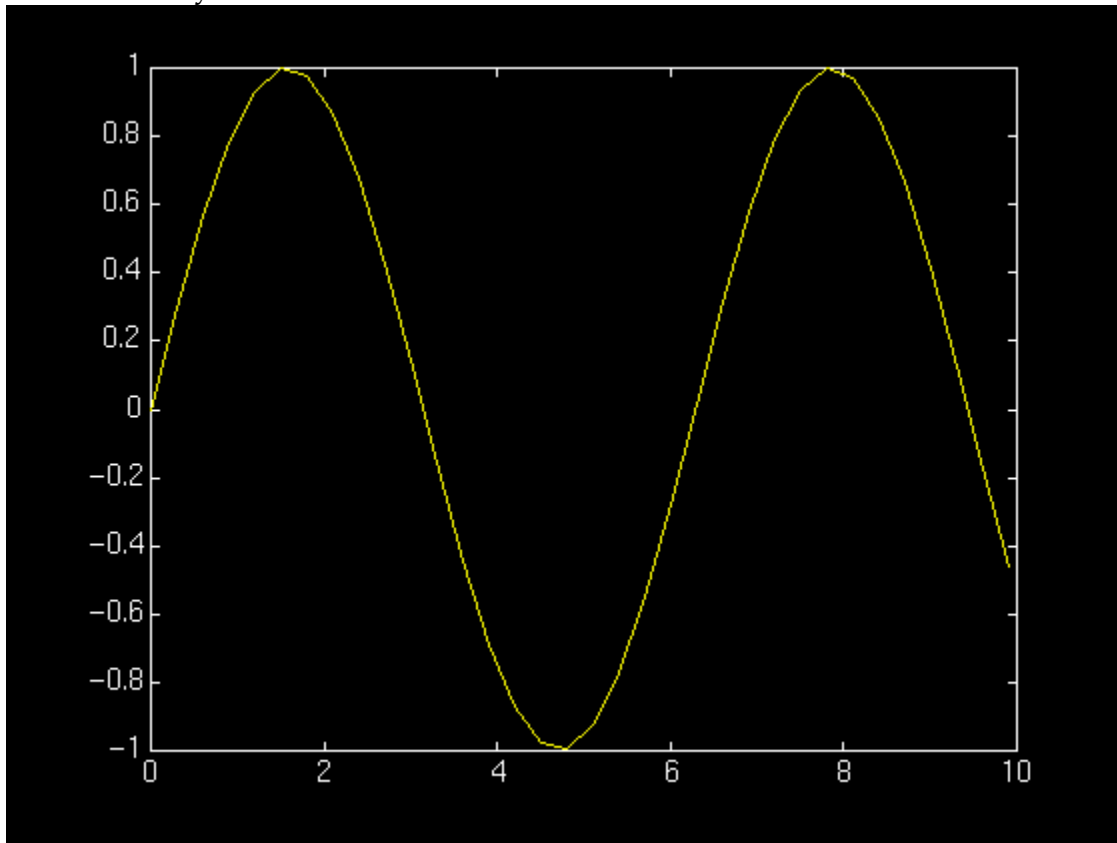
Functions of one variable:

To produce a graph of $y = \sin(t)$ on the interval $t = 0$ to $t = 10$ we do the following:

```
>> t = 0:3:10;
>> y = sin(t);
```

```
>> plot(t,y)
```

Here is what you should see on the screen:



The command `t = 0:0.3:10;` defines a vector with components ranging from 0 to 10 in steps of 0.3. The `y = sin(t);` defines a vector whose components are $\sin(0)$, $\sin(0.3)$, $\sin(0.6)$, etc. Finally, `plot(t,y)` use the vector of `t` and `y` values to construct the graph.

Functions of two variables:

Matlab plotting is well documented by **help plot**. Try the simple plot generation,

```
>> x = 1:10
```

```
x =
```

```
1 2 3 4 5 6 7 8 9 10
```

```
>> y = 2*x
```

y =

2 4 6 8 10 12 14 16 18 20

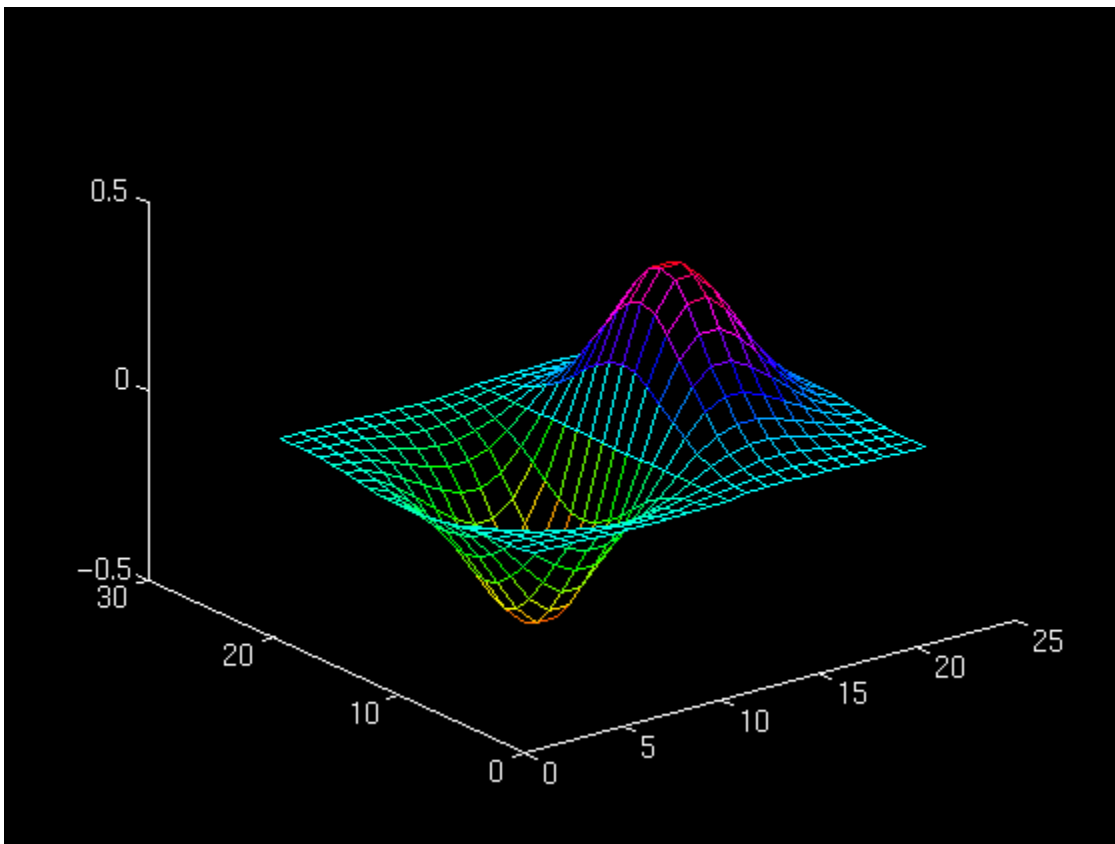
```
>> plot(x,y)
```

The last command generates a plot. You can label the plot, each axes, export the plot, use graphical tools in the plot window to add lines, text, etc. Features can change with upgrades to Matlab, so you will have to review those to take advantage of new capabilities.

Review the help on **plot** to see how you can change the line type, color, and legend symbol.

Here is how we graph the function $z(x,y) = x \exp(-x^2 - y^2)$:

```
>> [x,y] = meshdom(-2:.2:2, -2:.2:2);  
>> z = x .* exp(-x.^2 - y.^2);  
>> mesh(z)
```



The first command creates a matrix whose entries are the points of a grid in the square $-2 \leq x \leq 2$, $-2 \leq y \leq 2$. The small squares which make up the grid are 0.2 units wide and 0.2 unit tall. The second command creates a matrix whose entries are the values of the function $z(x,y)$ at the grid points. The third command uses this information to construct the graph.

What happens now when you enter the following?

```
>> surf(z)
```

You can now make the plot very stylish by entering

```
>> shading interp, lighting phong
```

This should give you the idea that you can make very impressive graphics using Matlab's graphics functions. For more graphics and plotting functions type

```
>> help graph3d.
```

Reading Data Files

One easy way to read in a data file is to use MATLAB's built in function `load`. Create a file in emacs that contains a matrix, e.g.

```
2 3 5  
12 33 1  
9 45 2  
2 4 4
```

If this file is called `matrix1.data` it can be loaded into MATLAB in the following way :

```
>> load matrix1.data -ASCII
```

You can then type `whos` to check the file was loaded ok.

You can also use `fscanf` to read in data from a file. Essentially, `fscanf` uses the same notation as it uses in C (more or less, anyway). One quick example is, suppose we had the following file :

```
64  
12.23 321.9
```

4 32
23.32 1

called 'my_file.txt'. You could load it into MATLAB like this :

```
>> fid = fopen('my_file.txt','r')
>> num1 = fscanf(fid, '%d', 1);
>> mat1 = fscanf(fid, '%f',[2,3]);
>> fclose(fid);
```

The 'r' in the fopen() command lets you open the file for reading. In the quotes, '%d' means integer, and '%f' means float. In the second call to fscanf() the [2,3] means ``read a matrix of 2 *columns* and 3 *rows*'' - note that it's column,row instead of the usual row,column! See the help pages for more info.

Writing Data to Files

The easiest way of printing things to a file is to use fprintf. Here is a very quick example of how to use this command :

```
>> var1 = 67;
>> mat1 = [1.223 45; 3 9; 12.2 6];
>> fid = fopen('save1.txt','w');
>> fprintf(fid, 'Some text \n');
>> fprintf(fid, '%d \n', var1);
>> fprintf(fid, '%f %f \n', mat1);
>> fclose(fid);
```

The 'w' in the call to the fopen() function means ``open the file for writing''. The '\n' in the quotes in the fprintf command is standard ANSI C format notion meaning ``new line''.

Creating and Using Matlab .m Files

You can write functions in MATLAB by creating a what is called a .m file **which MUST have the same name as the function**. The first line must be of the form function [<return var1>, <return var2>, ...] = name(arg1, arg2, ...). You then type in MATLAB commands as normal, and they are executed sequentially when the function is called. E.g., create a file ``row_col.m'' which contains the following.

```
function [rows, columns] = row_col(MAT)
```

```
rows = size(MAT,1);
```

```
columns = size(MAT,2);
```

You can then call this function from MATLAB by :

```
>> A = [1 2; 3 4; 5 6; 7 8];
```

```
>> [r, c] = row_col(A);
```

```
>> r
```

```
ans = 4
```

```
>> c
```

```
ans = 2
```