

Matlab Tutorial

Francesco Franco

Courtesy of
Francesco Franco.
Used with permission.

Matlab is a software package that makes it easier for you to enter matrices and vectors, and manipulate them. The interface follows a language that is designed to look like the notation used in linear algebra.

This tutorial guides you in the first steps for using Matlab. Start the program, The main window is subdivided in three windows. The Command Window is the one where you enter your commands (after a `>>` which is used to denote a command line).

I

Vectors : Almost all of Matlab Basic commands revolve around the use of vectors. To simplify the creation of vectors, you can define a vector by specifying the first entry, an increment, and the last entry. Matlab will automatically figure out how many entries you need and their values. For example, to create a vector whose entries are 1,3,5,7, type the following

```
>> 1:2:7

ans =

    1    3    5    7
```

Matlab keeps track of the last result. In the previous example a variable “ans” is created. To look at the transpose of the previous result, enter the following (‘ is the transpose operator):

```
>> ans'

ans =

     1
     3
     5
     7
```

To be able to keep track of the vectors you create, you can give them names. To create a row vector v:

```
>> v=[1:2:7]

v =

     1     3     5     7

>> v

v =

     1     3     5     7
```

If you add a semi-colon (;) at the end of the command line, the result is not displayed. Matlab will allow you to look at specific parts of the vector, thus to work with the first 3 elements of v:

```
>> v(1:3)
```

```
ans =
```

```
1 3 5
```

Once you master the notation you are free to perform other operations:

```
>> v(1:3)-v(2:4)
```

```
ans =
```

```
-2 -2 -2
```

Defining a matrix is similar to defining a vector. To define a matrix, you can treat it like a column of row vectors (note that the spaces are required!):

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

```
A =
```

```
1 2 3
```

```
3 4 5
```

```
6 7 8
```

You can also treat it like a row of column vectors:

```
>> B= [[1 2 3]' [2 4 7]' [3 5 8]']
```

```
B =
```

```
1 2 3
```

```
2 4 5
```

```
3 7 8
```

Now you have a lot of variables defined. If you lose track of what variables you have defined, the *whos* command will let you know all of the variables you have in your work space.

```
>> whos
```

```
Name      Size      Bytes Class
```

```
A         3x3        72 double array
```

```
B         3x3        72 double array
```

```
ans       1x3        24 double array
```

```
v         1x4        32 double array
```

```
Grand total is 25 elements using 200 bytes
```

You can of course multiply matrices and vectors (be careful that they must have the right size):

```
>> A*v(1:3)'  
  
ans =  
  
    22  
    40  
    67
```

You can work with different parts of a matrix, just as you can with vectors, thus to work with the submatrix $A_{(1:2,2:3)}$:

```
>> A(1:2,2:3)  
  
ans =  
  
     2     3  
     4     5
```

Once you are able to create and manipulate a matrix, you can perform many standard operations on it. For example, you can find the inverse of a matrix. You must be careful, however, since the operations are numerical manipulations done on digital computers, (calculate the determinant of A...maybe using matlab the command is `det(A)`), the software still gives you an answer :

```
Warning: Matrix is close to singular or badly scaled.  
Results may be inaccurate. RCOND = 3.469447e-018.  
  
ans =  
  
1.0e+015 *  
  
-2.7022  4.5036 -1.8014  
 5.4043 -9.0072  3.6029  
-2.7022  4.5036 -1.8014
```

One source of problem is that Matlab is case sensitive, thus A is not a. All the operations available are in the manual and in the help of Matlab. One easy way to remember what an operation does is to enter help “name of the operation” at the command line, for instance to find the eigen values of the matrix the command is `eig`, so let’s see how to use the command:

```
>> help eig  
  
EIG  Eigenvalues and eigenvectors.  
E = EIG(X) is a vector containing the eigenvalues of a square  
matrix X.  
  
[V,D] = EIG(X) produces a diagonal matrix D of eigenvalues and a  
full matrix V whose columns are the corresponding eigenvectors so  
that  $X*V = V*D$ .
```

`[V,D] = EIG(X,'nobalance')` performs the computation with balancing disabled, which sometimes gives more accurate results for certain problems with unusual scaling. If X is symmetric, `EIG(X,'nobalance')` is ignored since X is already balanced.

`E = EIG(A,B)` is a vector containing the generalized eigenvalues of square matrices A and B .

`[V,D] = EIG(A,B)` produces a diagonal matrix D of generalized eigenvalues and a full matrix V whose columns are the corresponding eigenvectors so that $A*V = B*V*D$.

`EIG(A,B,'chol')` is the same as `EIG(A,B)` for symmetric A and symmetric positive definite B . It computes the generalized eigenvalues of A and B using the Cholesky factorization of B .

`EIG(A,B,'qz')` ignores the symmetry of A and B and uses the QZ algorithm.

In general, the two algorithms return the same result, however using the QZ algorithm may be more stable for certain problems.

The flag is ignored when A and B are not symmetric.

See also `CONDEIG`, `EIGS`.

Thus typing `eig(A)` gives us the eigen values, while typing `[v,e]=eig(A)` gives us both the eigen values and the eigen vectors:

```
>> eig(A)

ans =

    14.0664
    -1.0664
    -0.0000

>> [v,e]=eig(A)

v =

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

e =

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

Now if you want to find x , which is the solution to $v=Bx$:

```
>> v=[1 3 5]'  
  
v =  
  
    1  
    3  
    5  
>> x=inv(B)*v  
  
x =  
  
    2.0000  
    1.0000  
   -1.0000
```

Or using the operator \ (left matrix divide):

```
>> x=B\v  
  
x =  
  
    2  
    1  
   -1
```

If it was $v=xB$, then

```
>> x=v'*inv(B)  
  
x =  
  
    4.0000   -3.0000    1.0000  
>> x=v'/B  
  
x =  
  
    4.0000   -3.0000    1.0000
```

To clear all your data in memory, use `clear`. (check using `whos` afterwards). To have a complete list of the operators type `help +`. You of course know the rules to add, subtract, multiply and divide matrices (they must be conformable).

```
>> v=[1 2 3]'  
  
v =  
  
    1  
    2  
    3
```

```
>> b=[2 4 6]'
```

```
b =
```

```
2  
4  
6
```

```
>> v*b'
```

```
ans =
```

```
2  4  6  
4  8 12  
6 12 18
```

There are many times where we want to do an operation to every entry in a vector or matrix. Matlab will allow you to do this with "element-wise" operations. For example, suppose you want to multiply each entry in vector v with its corresponding entry in vector b . In other words, suppose you want to find $v(1)*b(1)$, $v(2)*b(2)$, and $v(3)*b(3)$. It would be nice to use the "*" symbol since you are doing some sort of multiplication, but since it already has a definition, we have to come up with something else. The programmers who came up with Matlab decided to use the symbols ".*" to do this. In fact, you can put a period in front of any math symbol to tell Matlab that you want the operation to take place on each entry of the vector.

```
>> v.*b
```

```
ans =
```

```
2  
8  
18
```

```
>> v./b
```

```
ans =
```

```
0.5000  
0.5000  
0.5000
```

Now let's work with a large vector, and let's use more fancy functions (If you pass a vector to a predefined math function, it will return a vector of the same size, and each entry is found by performing the specified operation on the corresponding entry of the original vector). Remember to use the colon if you do not want the output to be printed on the screen)

```
>> x=[0:0.1:100]

x =

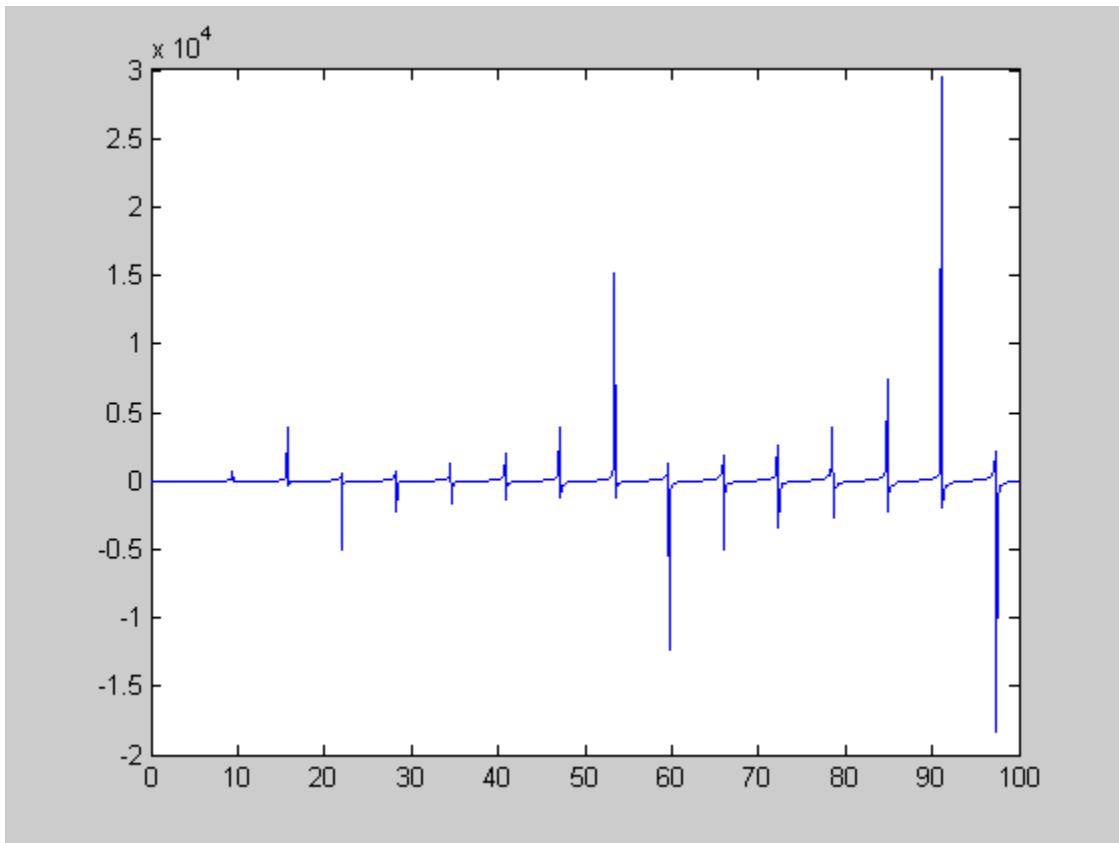
Columns 1 through 11

    0    0.1000    0.2000    0.3000    0.4000    0.5000    0.6000    0.7000    0.8000    0.9000
 1.0000.....

>> y=sin(x).*x./(1+cos(x));
```

Now let us graph the result:

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



To see all the graph options type help plot.

II

Once we will write programs Loops will become very important. A llop allows us to repeat certain commands. If you want to repeat some action in a predetermined way, you can use the "for" loop. All of the loop structures in matlab are started with a keyword such as "for", or

"while" and they all end with the word "end". The "for" loop will loop around some statement, and you must tell Matlab where to start and where to end. Basically, you give a vector in the "for" statement, and Matlab will loop through for each value in the vector:

For example, a simple loop will go around four times:

```
>> for j=1:4,  
j  
end  
j=  
1  
j=  
2  
j=  
3  
j=  
4
```

Another example, is one in which we want to perform operations on the rows of a matrix. If you want to start at the second row of a matrix and subtract the previous row of the matrix and then repeat this operation on the following rows, a "for" loop can do this in short order:

```
>> A= [[1 2 3]' [3 2 1]' [2 1 3]']  
A =  
1 3 2  
2 2 1  
3 1 3  
>> for j=2:3,  
A(j,:)=A(j,:)-A(j-1,:)   
end
```

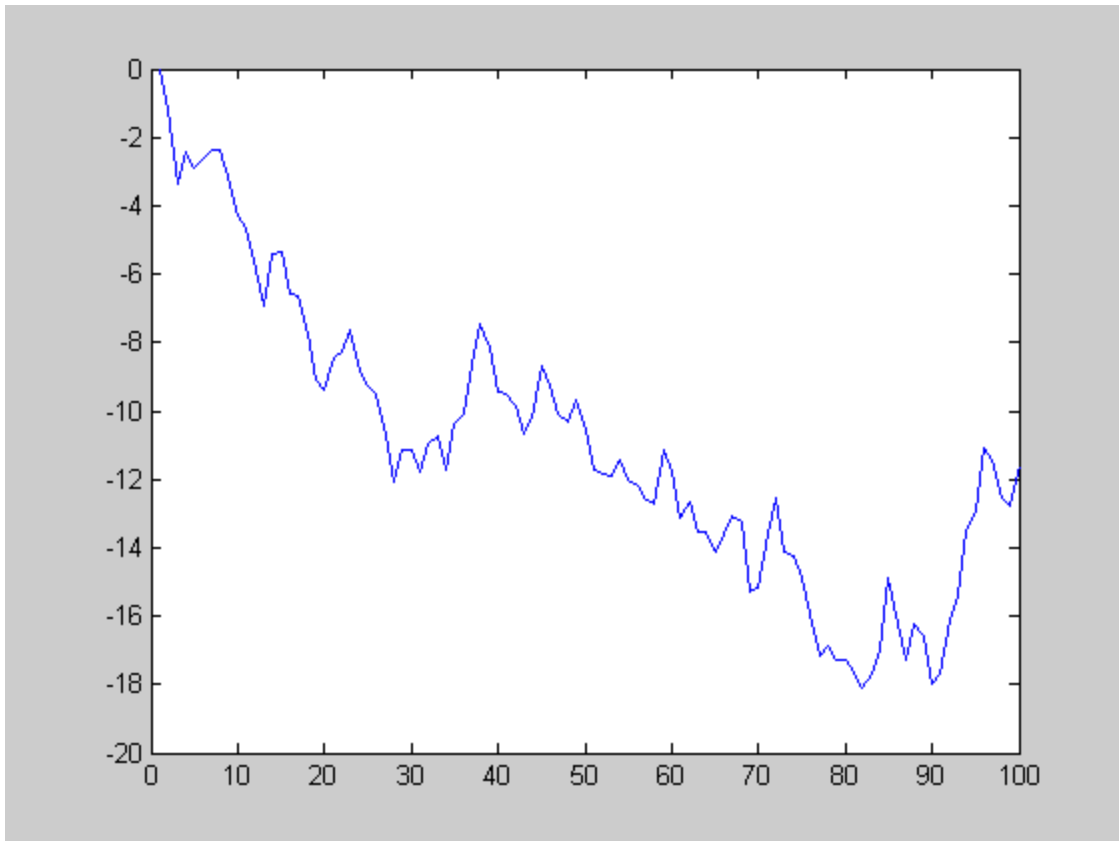


```
A =  
 1  3  2  
 1 -1 -1  
 3  1  3
```

```
A =  
 1  3  2  
 1 -1 -1  
 2  2  4
```

The while loop repeats a sequence of commands as long some condition is met. For instance let us create a Random Walk $y(t)=y(t-1) + \varepsilon(t)$, where ε is iid $N(0,1)$ and $y(0)=0$.

```
>> e=randn(1,100);  
>> i=1;  
>> y=0*e;  
>> size(y)  
  
ans =  
  
1 100  
  
>> max(size(y))  
  
ans =  
  
100  
  
>> while(i<max(size(y)))  
y(i+1)=y(i)+e(i+1);  
i=i+1;  
end  
  
>> t=[1:1:100];  
>> plot(t,y)
```



III

Now we have enough instruments to create an executable file. Once you have a general routine in a matlab file, it allows you to perform more complex operations, and it is easier to repeat these operations. For example, you might have a set of instructions to create a Random Walk, but you want to be able to use those instructions for initial conditions, and the number of observations. First, you will need to create the file. Go on File, choose new and then M.file. This opens the Matlab editor. Type the following :

```
% file: Random Walk.m
% This file will create a Random Walk  $y(t)=y(t-1)+e(t)$ 
% where e is a  $N(0,1)$ 
% To run this file you will first need to specify
%   T   : number of observations
%   starty : initial value
% The routine will generate a column vector
% and plot the vector

% Switches
T=100;
starty=0;

%Routine
y=starty*ones(1,T);
```

```

e=randn(1,T);
for i=1:(T-1),
    y(i+1)=y(i)+e(i+1);
end

time=[1:1:T];
plot(time,y);

```

Once the commands are in place, save the file. Go back to your original window and start up matlab. The file is called up by simply typing in the base name : Randomwalk (be sure to be in the right directory, you can use chdir path).

Sometimes you want to repeat a sequence of commands, but you want to be able to do so with different vectors and matrices. One way to make this easier is through the use of subroutines. Subroutines are just like executable files, but you can pass it different vectors and matrices to use. For instance you want to calculate the utility of consumption using a power utility function, we then want to create the routine poweru that calculate this utility when we call it. The only difference with the executable file is that in the first line we will have function[x] = power(c) (it needs c as an input):

```

% Power utility function u(c)=c^(1-gamma)/(1-gamma) if gamma ~= 1;
% u(c)=ln(c) if gamma=1.
% Utility is only defined if c>=0 for gamma<1 and c>0 if gamma>1

function u=poweru(c)

gamma=1;

[m,n]=size(c);
u=zeros(m,n);

if (gamma<0)
    u=-inf*abs(c);
% display('invalid relative risk aversion coefficient');
else
    for i=1:m
        for j=1:n
            if (gamma<1 & c(i,j) >= 0) | (gamma>1 & c(i,j)>0)
                u(i,j)=c(i,j)^(1-gamma)/(1-gamma);
            else if (gamma==1 & c(i,j)>0)
                u(i,j)=log(c(i,j));
            else
                u(i,j)=-inf;
% display('consumption must be nonnegative if RRA coefficient is less than 1 and
positive if greater than and equal to 1')
            end
        end
    end
end
end
end

```

As you have noticed we have used the if statement. There are times when you want your code to make a decision. In the previous case you want the routine to use different functional forms depending on the value of the RRA. Thus if $\gamma=1$ we use the log, or if γ is less than zero, the program will tell you that this value is invalid. Each if statement must be terminated with an end command, as you notice you can create else or elseif block inside an if statement. (type Help if).