MATLAB TUTORIAL

MATLAB is an abbreviation of MATRIX LABORATORY and was initially developed to do linear algebra, by providing an easy way to handle and manipulate matrices.

The idea behind this tutorial is that you view it in one window (or have a printout) while running MATLAB in another window. You should be able to re-do all of the plots and calculations in the tutorial by typing the commands of the tutorial into MATLAB or an m-file. `I will use this font to denote text that has to be typed in` MATLAB.

Index

1. Getting Started

To start MATLAB from Windows:

1) Click on Start
2) Select Programs and then MATLAB

Once the programme starts, you will see three separate windows; the large window on the right hand side is the **Command Window** of MATLAB. This is the main space you will be using. On top of it you will see:

```
To get started, select "MATLAB Help" from the Help menu.
>>
```

This window allows a user to enter simple commands. To perform a simple computations type a command and next press the **Enter**/**Return** key.[1] For instance

[1] From now on I will be saying **Enter** key and mean **Enter** or **Return** key.

```
» s = 1 + 2

s =

     3
```

Note that the results of these computations are saved in variables whose names are chosen by the user. When they are needed during your current MATLAB session, you can obtain their values by typing their names and pressing the **Enter** key. For instance type again

```
» s

s =

     3
```

The other windows you see (two windows on the left with tabs) are **Launch Pad, Workspace** (top left window) and **Command History, Current Directory** (bottom left window).

The **Command Window** is the place where you type commands to be executed. The **Launch Pad** contains the toolboxes available with your version (we will not be needing this for the moment). The **Workspace** shows you all the variables that are currently stored in the memory (notice the s that appears there). The **Command History** gives you a list of the commands you have executed in the Command Window. Finally, the **Current Directory** shows you where MATLAB will look for m-files or functions.

To close MATLAB type `exit` in the **Command Window** and next press **Enter** key. A second way to close your current MATLAB session is to select **File** in the MATLAB's toolbar and next click on **`Exit MATLAB`** option. All unsaved information residing in the MATLAB **Workspace** will be lost. You can also exit by typing `quit`

Here are some useful things you can do with the keyboard in the **Command Window**:

Keys	**Result**
Enter	The equivalent of double-clicking, it performs the default action for a selection. For example, pressing Enter while a line in the Command History window is selected runs that line in the Command Window.
Escape	Cancels the current action.
Ctrl+Tab or Ctrl+F6	Moves to the next tab in the desktop, where the tab is for a tool, or for a file in the Editor/Debugger. When used in the Editor/Debugger in tabbed mode outside of the desktop, moves to the next open file.
Ctrl+Shift +Tab	Moves to the previous tab in the desktop, where the tab is for a tool, or for a file in the Editor/Debugger. When used in the Editor/Debugger in tabbed mode outside of the desktop, moves to the previous open file.
Ctrl+Page Up	Moves to the next tab within a group of tools or files tabbed together.
Ctrl+Page Down	Moves to the previous tab within a window.
Alt+F4	Closes desktop or window outside of desktop.

Ctrl+C	Aborts the action that is being executed
Alt+Space	Displays the system menu.

IMPORTANT NOTE: MATLAB IS CASE SENSITIVE!!!

2. MATLAB Help

One of the nice features of MATLAB is its help system. A good place to start is with the command

```
» help help
```

that explains how the help systems works. In addition, the command **help** produces a list of topics for which help is available. In general, to get more information on any given command, type **help+name**, e.g.

```
» help sqrt
 SQRT   Square root.
    SQRT(X) is the square root of the elements of X. Complex
    results are produced if X is not positive.

    See also SQRTM.
```

The command **help** is good for fast reference for some command you already know something about. If want more details on how to do an operation (for example to see if there is an in built function that can do something specific) use the help browser. This has a search engine.

3. Numbers and Arithmetic Operations

There are three kinds of numbers used in MATLAB: integers, real numbers and complex numbers. In addition to these, MATLAB has three variables representing the non-numbers:

```
-Inf
Inf
NaN
```

The **–Inf** and **Inf** are the negative and positive infinity, respectively. Infinity is generated by overflow or by the operation of dividing by zero. The **NaN** stands for Not-A-Number and is obtained as a result of the mathematically undefined operations such as 0.0/0.0.

List of basic arithmetic operations in MATLAB include six operations:

```
+ : addition
```

```
- : substraction

* : multiplication

/ : right division

\ : left division

^ : power
```

4. Special Characters

There is a menu of special characters that have specific uses. For now, we will see only the main ones (you will be discovering the rest on the way). These are:

Relational operators

- Equal ==
- Not equal ~=
- Less than <
- Greater than >
- Less than or equal <=
- Greater than or equal >=

Logical operators

- Logical AND &
- Logical OR |
- Logical NOT ~

Special characters

- Colon :
- Parentheses and subscripting ( )
- Brackets [ ]
- Decimal point .
- Continuation ...
- Separator ,

- Semicolon ;

- Comment %

- Assignment =

- Quote ' *statement* '

- Transpose '

Some comments:

The colon (**:**) is mostly used to denote "up to", for example, if you want to repeat a step n times, you would want to declare it as "from 1 up to n" so you type `1:n`. It is also used in other ways, which we will see on the way, through examples.

The semicolon (**;**) is mainly used in two ways. First, to supress the output of a calculation and second to declare a new row or line. We will several examples for using this.

5. Vectors, Matrices and Three-Dimensional Arrays

All inputs in in MATLAB are taken to be arrays. In this way, a scalar is a 1x1 array, a row vector is a 1xn array, a column vector is an nx1 array and a matrix is a nxm array. MATLAB can also handle three dimensional arrays (that you can think of as matrices stacked one on top of the other). To define arrays, we use the brackets.

To create the row vector a type:

```
» a = [1 2 3 4 5 6 9 8 7]

a =

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

Note that separating the elements by a space you create a row vector. If you want to create a column vector, separate the elements by semicolons (in general the semicolon can be used to declare new row/line).

```
» b = [1; 2; 3; 4; 5; 6]

b =

     1
     2
     3
     4
     5
     6
```

Another way to create column vectors is by using the transpose, which is denoted with a **'** after the bracket:

```
» b = [1   2    3   4   5   6]'
b =
     1
     2
     3
     4
     5
     6
```

The command **length** returns the number of components of a vector:

```
» length(a)
ans =

     9
```

If you want to type a scalar just write:

```
» c = 2
c =
     2
```

(Of course, you can always use the bracket and type `c = [2]`)

Suppose you want to define a new vector, by adding 2 to each of the elements of vector a. Then, type

```
» e = a + 2

e =
```

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

Now suppose, you want to add two vectors together. If the two vectors are of the same length, it is easy. Simply add the two:

```
» f = a + e
f =
     4     6     8    10    12    14    20    18    16
```

Subtraction of vectors of the same length works exactly the same way. A useful command is **whos**, which displays the names of all defined variables and their types.

```
>> whos
Name      Size          Bytes  Class

  a         1x9             72  double array
  b         6x1             48  double array
```

```
  c          1x1                8  double array
  e          1x9               72  double array
  f          1x9               72  double array

Grand total is 34 elements using 272 bytes
```

You can also view information about the size of arrays, as well as their values in the **Workspace** window (top left).

Entering matrices into MATLAB is the same as entering a vector, except each row of elements is separated by a semicolon (**;**):

```
» B = [1 2 3 4; 5 6 7 8; 9 10 11 12]
  B =
```

```
     1     2     3     4
     5     6     7     8
     9    10    11    12
```

Alternatively, in m-files, you can enter the matrix as follows:

```
» B = [ 1  2  3  4
```

```
  5  6  7  8
9 10 11 12]
```

Note that this way of defining the matrix will not work in the Command Window, unless you copy and paste the whole thing.

Like with vectors, we have an in-built function that gives the dimensions of a matrix. This is **size**.

```
» size(B)
    ans =
```

```
3    4
```

This should be read as 'A has 3 rows and 4 columns'.

For your convenience, MATLAB can produce some commonly used matrices with in-built functions. The common ones are:

zeros(m,n) creates an m x n matrix of zeros

ones(m,n) creates an m x n matrix of ones

eye(n) creates the n x n identity matrix

diag(v) (assuming v is an n-vector) creates an n x n diagonal matrix with v on the diagonal.

For example, type

```
» zeros(2,1)
ans =
```

```
0
0
```

Matrix manipulation and operations

Matrices in MATLAB can be manipulated in many ways. For one, you can find the transpose of a matrix using the apostrophe key:

```
» C = B'
```

```
C =
          1    5    9
          2    6   10
          3    7   11
          4    8   12
```

Now you can multiply the two matrices B and C together. Remember that order matters when multiplying matrices.

```
» D = B * C
```

```
D =       30    70   110
          70   174   278
         110   278   446
```

```
» D  = C * B
```

```
 D =
     107   122   137   152
     122   140   158   176
     137   158   179   200
     152   176   200   224
```

Another option for matrix manipulation is that you can multiply the corresponding elements (element-by-element multiplication) of two matrices using the **.*** operator (the matrices must be the same size to do this).

```
» E = [1 2; 3 4]
        E =
             1     2
```

```
             3      4
» F = [2 3;4 5]
        F =
             2      3
             4      5
» G = E .* F
```

```
G =
     2      6
    12     20
```

If you have a square matrix, like E, you can also multiply it by itself as many times as you like by raising it to a given power.

```
» E^3
```

```
ans =
      37     54
      81    118
```

If wanted to cube each element in the matrix seperately, just use the element-by-element cubing (**.^**).

```
» E.^3
```

```
  ans =
         1      8
        27     64
```

In general:

- **a .* b** multiplies each element of a by the respective element of b
- **a ./ b** divides each element of a by the respective element of b
- **a .\ b** divides each element of b by the respective element of a
- **a .^ b** raise each element of a by the respective b element

You can also find the inverse of a matrix:

```
» X = inv(E)
```

```
  X =
       -2.0000    1.0000
        1.5000   -0.5000
```

or its eigenvalues:

```
» eig(E)
```

```
 ans =
```

```
         -0.3723
          5.3723
```

If A and B are matrices, then MATLAB can compute A+B and A-B when these operations are defined. For example, consider the following commands:

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

>> B = [1 1 1;2 2 2;3 3 3]
B =
     1     1     1
     2     2     2
     3     3     3

>> C = [1 2;3 4;5 6]
C =
     1     2
     3     4
     5     6

>> A+B
ans =
     2     3     4
     6     7     8
    10    11    12

>> A+C
??? Error using ==> +
Matrix dimensions must agree.
```

This happened because the operation we tried to do is not well-defined. You will get the same message if you try to multiply matrices that do not have compatible orders.

In several occasions you will need to do extract or manipulate some specific element(s) of a matrix. To see how this is done, you first need to understand how MATLAB understands and reads an **array**. Every array is indexed by a number of indexes that is equal to its dimensions. In other words, a vector has one index, a matrix has two indexes and a 3-dimensional array has

three indexes. For a matrix, the first index indicates rows, while the second indicates columns. Same applies to 3D arrays, where the last index indicates the 3rd dimension.

Now, define

```
>> C = [1 2 3 4; 5 6 7 8; 9 10 11 12]'
C =
     1     5     9
     2     6    10
     3     7    11
     4     8    12
```

Suppose you want to extract the element of C that is on the second row and first column. You should then type:

```
» C(2,1)
ans =
     2
```

Now suppose you want to extract a whole row or column of C. E.g. the following extracts the third row of C

```
» C(3,:)
ans =
     3     7    11
```

The colon is used here to denote that we want the elements in the 3rd row from all the columns. Similarly, if we wanted to extract the first column of C, we type

```
» C(:,1)
ans =
     1
     2
     3
     4
```

Last, suppose that you want to extract some block of the matrix. Try this

```
» C(1:3,2:3)
ans =
     5     9
     6    10
     7    11
```

This extracts the block that is defined by rows 1 to 3 and columns 2 to 3.

Using the same principle, you can alter/manipulate matrix elements. For example, suppose you want to redefine the (2,2) element of C, to be 10. Then type

```
» C(2,2) = 10

C =

     1     5     9
     2    10    10
     3     7    11
     4     8    12
```

This operation automatically erases the old value of the element from the computer memory and redefines it as 10. MATLAB will again give you as output the new matrix, where you have changed the (2,2) element.

Since you already know how the new C will look like, you might want to suppress the output (using the semicolon). To do this type

```
» C(2,2) = 10;

»
```

Although you do not see the new matrix, it is stored in the memory.

Last, a common operation you might want to do is to solve a system of linear equations Ax = b. The solution for this is $x = A^{-1} b$. To work an example, first, define a matrix A. Recall that we do not need to define b, as it is already stored in the memory. If you have forgotten the size of b, use **length(b)**

```
>> A = [1 1 0 1 2 0; 2 1 1 0 4 -1; 3 2 1 0 0 1; 1 1 1 0 1 1; -1 2 1 0 1 1; 4 1
2 3 1 0]

A =

     1     1     0     1     2     0
     2     1     1     0     4    -1
     3     2     1     0     0     1
     1     1     1     0     1     1
    -1     2     1     0     1     1
     4     1     2     3     1     0
```

In order to find the solution, we need the inverse of A. There are two ways to get the inverse. First, is an in-built function **inv** so that you do

```
»  x = inv(A)*b

x =

   -0.4200
    0.1600
    2.6700
    0.6200
    0.3200
    1.2700
```

or you can use the left division ****

```
»  x = A\b
x =
   -0.4200
    0.1600
    2.6700
    0.6200
    0.3200
    1.2700
```

Note that the left division **** is more efficient than **inv** in terms of computing time (when you invert big matrices you will notice the difference), but its use can be sometimes confusing, especially if you have many matrices to multiply!

Finally, note that the colon (**:**) can be used to generate easily arrays with a certain pattern. For example, suppose you want to create a row vector that has elements that start from 1 and go progressively to 6, by adding 1 to the previous element (i.e. a vector [1 2 3 4 5 6]). Instead of typing all the elements, you can use the colon as follows:

```
» y = 1:6
y =
     1     2     3     4     5     6
```

To create a similar column vector type

```
» y = (1:6)'
y =
     1
     2
     3
     4
     5
     6
```

If you want to make the step with which the elements increase different from 1 (default value), e.g. 0.5, type

```
» y = 1:0.5:3
y =
    1.0000    1.5000    2.0000    2.5000    3.0000
```

Notice that by doing this, the length of the vector is determined by the end value 3.

6. Clearing the Workspace

Before we go on with the next section, we will see how erase objects that are already stored in the memory. The relevant command is **clear**. First, click the tab **Workspace** at the top left window. Inspect the variables that are stored in the memory.

If you want to erase only one object from the memory, e.g. C, type

```
» clear C
```

To convince yourselves that the variable is gone, call C

```
» C
??? Undefined function or variable 'C'.
```

You can also check that by inspecting the **Workspace**. If you want to clear all objects, type

```
» clear all
```

7. Random Number Generation

We can generate random numbers or arrays from two distributions, the U(0,1) and N(0,1). The relevant commands are **rand** (for uniform) and **randn** (for normal). Try the following

```
>> rand
```

returns a random number between 0 and 1.

```
>> randn
```

returns a random number selected from a normal distribution with a mean of 0 and variance of 1. **rand(r,c)** returns a matrix of dimensions rxc, of random numbers from U(0,1). E.g

```
>> A = rand(3,3)
A =
    0.9501    0.4860    0.4565
    0.2311    0.8913    0.0185
    0.6068    0.7621    0.8214
>> b = rand(3,1)
b =
    0.4447
    0.6154
    0.7919
```

8. Conditionals and loops

MATLAB has a standard **if-elseif-else** conditional. For example:

```
>> t = rand(1);
>> if t > 0.75
      s = 0;
```

```
   elseif t < 0.25
      s = 1;
   else
      s = 1-2*(t-0.25);
   end
>> t
t =
    0.7622
>> s
s =
     0
```

Thus the general form of the **if** statement is

```
if expr1
   statements
elseif expr2
   statements
   .
   .
   .
else
   statements
end
```

MATLAB provides two types of loops, a **for** loop and a **while** loop. A **for** repeats the statements in the loop as the loop index takes on the values in a given row vector:

```
>> for i=[1 2 3 4]
      disp(i^2) %(the built-in function disp, displays its argument.)
   end

     1
     4
     9
    16
```

The loop must also be terminated by **end**. This loop would more commonly be written as:

```
>> for i=1:4           %(recall that 1:4 is the same as [1 2 3 4])
      disp(i^2)
   end
```

```
     1
     4
     9
    16
```

The **while** loop repeats as long as the given expression is true:

```
>> x=1;
>> while 1+x > 1
      x = x/2;
   end
>> x
x =
   1.1102e-16
```

Thus, the general form of a while loop is

```
while   expression
    statements
end
```

The statements will be repeatedly executed as long as the relation remains true. Another example: for a given number a, the following will compute and display the smallest nonnegative integer n such that $2^n >= a$. Let, for example a = 4.

```
>> a = 4;
>> n = 0;
>> while  2^n < a
      n = n + 1;
   end
>> n
n =
     2
```

9. Scripts and Functions

MATLAB includes many standard functions. Indeed, all of the standard functions such as sin, cos, log, exp, sqrt, as well as many others. Commonly used constants such as pi are also incorporated into MATLAB. Some examples are (a basic list can be found at the end of the tutorial):

```
» sin(pi/4)
ans =
```

```
    0.7071
```

(You must have noticed by now that as long as you don't assign a variable a specific operation or result, MATLAB will store it in a temporary variable called `ans`)

```
» cos(.5)^2+sin(.5)^2
 ans =
       1

» exp(1)
  ans =
      2.7183

» log(ans)+1
  ans =
       2
```

Note that, when entering a command such as **sqrt** into MATLAB what you are really doing is running an m-file or a script.

What is an m-file or a script?

An m-file is a simple text file where you can place MATLAB commands, so, it is simply a collection of MATLAB commands in an m-file (a text file whose name ends in the extension ".m", e.g. **sqrt.m**). When the code is being run, MATLAB reads the commands and executes them exactly as it would if you had typed each command sequentially at the MATLAB command window. To make life easier, choose a name for your m-file that doesn't already exist. To see if a filename exists, type **help filename** at the MATLAB workspace.

```
» help chryssi

chryssi.m not found.
```

The m-file must be located in one of the directories in which MATLAB automatically looks for m-files. One of the directories in which MATLAB always looks is the **current working directory**; the command **cd** identifies the current working directory, and **cd newdir** changes from the working directory to **newdir**.

For simple problems, entering your requests at the workspace is fast and efficient. However, as the number of commands increases typing the commands over and over at the workspace becomes tedious. M-files will be helpful and almost necessary in these cases. You actually use m-files to create your own programs.

How to create, save, open and run an m-file?

To create an m-file, choose **New** from the **File** menu and select **m-file**. This procedure brings up a text editor window in which you can enter MATLAB commands.

To save the m-file when you have typed your commands, simply go to the **File** menu and choose **Save as** (remember to save it with the '.m' extension). To open an existing m-file, go to the **File** menu and choose **Open**.

After the m-file is saved with the name **filename.m** in the MATLAB folder or directory, you can execute the commands in the m-file by simply typing filename at the MATLAB command window. You can also use m-files to create your own functions.

You can either type commands directly into MATLAB, or put all of the commands that you will need together in an m-file, and just run the file. If you put all of your m-files in the same directory that you run MATLAB from, then MATLAB will always find them.

How to create your own function?

The new function must be given a filename with a '.m' extension. For example, create a function (recall that functions are also m-files) that is called **addition.m**, which will add two numbers. The first line of the file should contain the syntax for this function in the form:

```
function [output1,output2,…outputn] = filename(input1,input2,…,inputn)
```

The inputs are what you have to give to the function, in this case the two variables you want to add, and the output will be the sum of the two variables. So, you open a new m-file, as explained above, and type the following:

```
Function  [var3] = addition(var1,var2)
```

The next few lines contain the text that will appear when the help addition command is evoked. For example, you can write:

```
% ADDITION is a function that adds two numbers
```

These lines are optional, but must be entered using % in front of each line in the same way that you include comments in an ordinary m-file. Finally, below the help text, the actual function with all of the commands is included. In this case, we would then have:

```
function [var3] = addition(var1,var2)
%addition is a function that adds two numbers
var3 = var1+var2;
```

If you save these three lines in a file called "addition.m" in the MATLAB directory, then you can use it always by typing at the command line back in the command window:

```
» z = addition(3,8)
z =
    11
```

Obviously, most functions will be more complex than the one demonstrated here. This example just shows what the basic form looks like. Try **help function** for more information.

10. Solving nonlinear problems

Before starting the section, clear all the variables as explained earlier.

In addition to functions for numerical linear algebra, MATLAB provides functions for the solution of a number of common problems, such as numerical integration, initial value problems in ordinary differential equations, root-finding, and optimization.

Polynomials

In MATLAB, a polynomial is represented by a vector. To create a polynomial in MATLAB, simply enter each coefficient of the polynomial into the vector in descending order. For instance, let's say you have the following polynomial:

$$s^4 + 3s^3 - 15s^2 - 2s + 9$$

To enter this into MATLAB, just enter it as a vector in the following manner

```
» x = [1 3 -15 -2 9]
x =
    1 3  -15  -2  9
```

MATLAB can interpret a vector of length n+1 as an nth order polynomial. Thus, if your polynomial is missing any coefficients, you must enter zeros in the appropriate place in the vector. For example,

$$s^4 + 1$$

would be represented in MATLAB as:

```
» y = [1 0 0 0 1]
```

You can find the value of a polynomial using the **polyval** function. For example, to find the value of the above polynomial at s = 2,

```
» z = polyval(y,2)
z =
    17
```

You can also extract the roots of a polynomial. This is useful when you have a high-order polynomial such as

$$s^4 + 3s^3 - 15s^2 - 2s + 9$$

Finding the roots would be as easy as entering the following command

```
» roots([1 3 -15 -2 9])
ans =
                -5.5745
                 2.5836
                -0.7951
                 0.7860
```

Optimization commands

The main root-finding command is **fzero** (only for single variable).

Let now f be a function. MATLAB function **fzero** computes a zero of the function f using user supplied initial guess of a zero. In the following example let f(x) = cos(x) – x. First create a new function y = f1(x) (in an m-file) by typing

```
function y = f1(x)
y = cos(x) - x;
```

and saving it as **f1.m**. Then, suppose that our initial guess for the solution is 0.5. Type in the Command Window

```
» r = fzero('f1', 0.5)
r =
0.73908513321516
```

In the m-file **f1.m** we tell MATLAB that f1 is a function and the second line defines the function. To compute its zero we use MATLAB function **fzero** and we have to give an initial guess (type `help fzero` for more information). Other useful built-in functions are:

fsolve root-finding (several variables)

fmin nonlinear minimization (single variable)

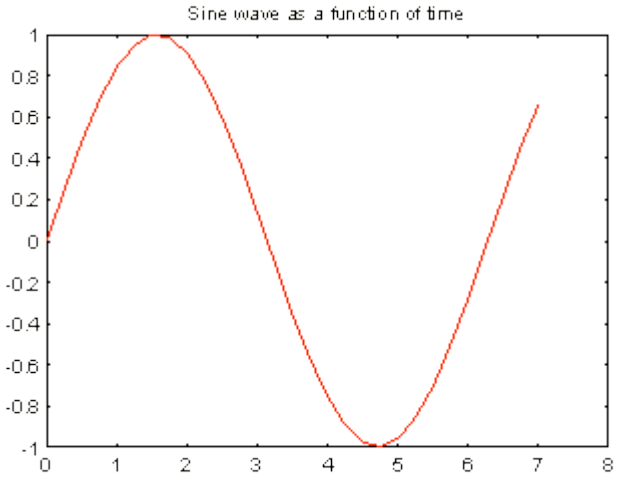
fmins nonlinear minimization (several variables)

11. Plotting

Before starting this part, clear all the variables from the Workspace, as explained earlier.

It is easy to create plots in MATLAB. Suppose you wanted to plot a sine wave as a function of time. First make a time vector and then compute the sin value at each time.

```
» t=0:0.25:7;
» y = sin(t);
» plot(t,y)
```



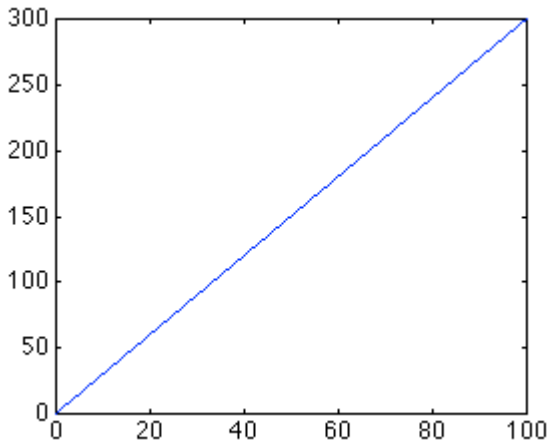
One of the most important functions in MATLAB is the **plot** function. Plot also happens to be one of the easiest functions to learn how to use. The basic format of the function is to enter the following command in the MATLAB command window or into a m-file.

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

This command will plot the elements of vector x on the horizontal axis of a figure, and the elements of the vector y on the vertical axis of the figure. The default is that each time the plot command is issued, the current figure will be erased; we will discuss how to override this below. If we wanted to plot the simple, linear formula: y=3x, we could type the following (or create a m-file with the following lines of code):

```
» x = 0:0.1:100;
» y = 3*x;
» plot(x,y)
```

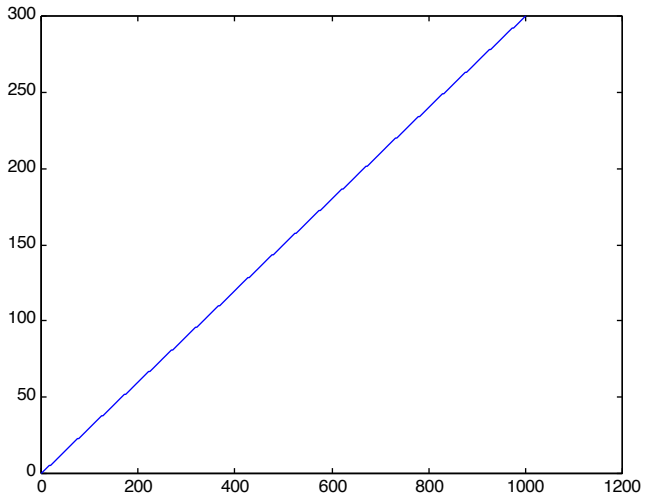
which will generate the following plot,



One thing to keep in mind when using the plot command is that the vectors x and y must be the same length. The plot command can also be used with just one input vector. In this case, the vector 1:1:n will be used for the horizontal axis, where n is the length of y. So, if you type:

```
» plot(y)
```

you will get the following picture:



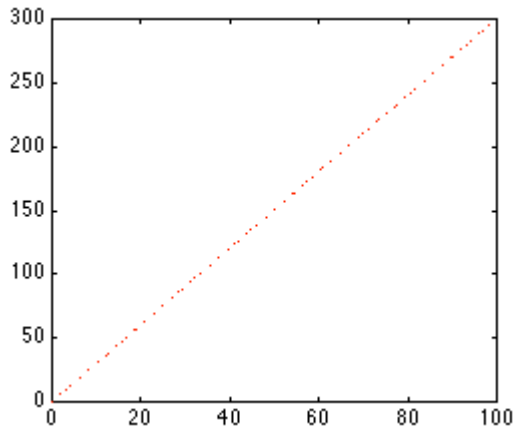
Plot aesthetics

With the latest versions of MATLAB you can manipulate the looks of a figure using the graph editor. The graph editor is an interface between you and the in-built function plot (to activate it, in the **figure** window click the arrow icon, then double-click anywhere in the graph). With older versions, the user had to manipulate the aesthetics manually (you can still do this, but it's a waste of time). Just to get a flavour of how this works, let's see an example.

The color and point marker can be changed on a plot by adding a third parameter (in single quotes) to the plot command. For example, to plot the above function as a red, dotted line, the m-file should be changed to:

```
» x = 0:0.1:100;
» y = 3*x;
» plot(x,y,'r:')
```

The plot now looks like:



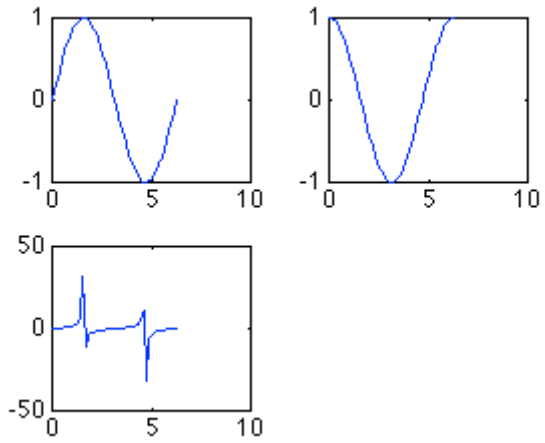
Subplotting

More than one plot can be put on the same figure using the subplot command. The **subplot** command allows you to separate the figure into as many plots as desired, and put them all in one figure. To use this command, the following line of code is entered into the MATLAB command window or an m-file:

```
» subplot(m,n,p)
```

This command splits the figure into a matrix of m rows and n columns, thereby creating m*n plots on one figure. The p'th plot is selected as the currently active plot. For instance, suppose you want to see a sine wave, cosine wave, and tangent wave plotted on the same figure, but not on the same axis. The following m-file will accomplish this:

```
» x = linspace(0,2*pi,50); % LINSPACE(x1, x2, N) generates N points between x1 and x2
» y = sin(x);
» z = cos(x);
» w = tan(x);
» subplot(2,2,1)
» plot(x,y)
» subplot(2,2,2)
» plot(x,z)
» subplot(2,2,3)
» plot(x,w)
```



There are only three plots, even though I created a 2 x 2 matrix of 4 subplots. Thus, you do not have to fill all of the subplots you have created, but MATLAB will leave a spot for every position in the matrix. The subplots are arranged in the same manner, as you would read a book. The first subplot is in the top left corner; the next is to its right.

One thing to note about the subplot command is that every plot command issued later will place the plot in whichever subplot position was last used, erasing the plot that was previously in it. For example, in the m-file above, if a plot command was issued later in the m-file, it would be plotted in the third position in the subplot, erasing the tangent plot. To solve this problem, a new figure should be specified (using figure). For example, two plots will be opened if you type (first close the old figure):

```
» figure(1)
» plot(x,y)
» figure(2)
» plot(x,z)
```

The best way to understand how to manipulate plots is by trial and error and by looking through the relevant help files. In general, keep in mind that the graph editor of the latest versions is very powerful and can do manipulations such as changing the axes, the looks of the lines (colors etc.), adding text of various sizes and fonts, adding gridlines, etc.

Furthermore, you can use the figure menu to export your figures in two different formats, namely bitmap (.bmp) and windows metafile (.wmf).

12. Basic Commands and Functions

- **max(x)** returns the largest entry of x, if x is a vector; see help max for the result when x is a *k*-dimensional array
- **min(x)** analogous to max
- **abs(x)** returns an array of the same size as x whose entries are the magnitudes of the entries of x

- **mean(x)** returns the mean value of the elements of a vector or if x is a matrix, returns a row vector whose elements are the mean value of the elements from each column of the matrix.
- **median(x)** same as mean(x), only returns the median value.
- **sum(x)** returns the sum of the elements of a vector or if x is a matrix, returns the sum of the elements from each respective column of the matrix
- **prod(x)** same as sum(x), only returns the product of elements.
- **std(x)** returns the standard deviation of the elements of a vector or if x is a matrix, a row vector whose elements are the standard deviations of each column of the matrix.
- **sort(x)** sorts the values in the vector x or the columns of a matrix and places them in ascending order. Note that this command will destroy any association that may exist between the elements in a row of matrix x.
- **size(A)** returns a **1x k** vector with the number of rows, columns, etc. of the *k*-dimensional array A
- **save fname** saves the current variables to the file named fname.mat
- **load fname** load the variables from the file named fname.mat
- **clear x** erases the matrix 'x' from your workspace
- **clear or clear all** erases ALL matrices from your workspace

10. Suggestions for Computing Efficiency

Before discussing efficiency, a useful set of commands that will help you detect the computing time you need to do an operation is **clock** and **etime**. Here's the general command of how to use them

```
t0 = clock;

      operation

etime(clock,t0)
```

Using this, let's see some things to avoid in order to reduce computing time.

Try to avoid loops in your programs. MATLAB is optimized to run the built-in functions. The following two command sequences have the same effect (in order to run this properly type or copy+paste this into an m-file then run it):

```
% Direct way - inbuilt function
t = (0:0.000005:1)';
t0 = clock;
y=sin(t);
timer1 = etime(clock,t0)

% Indirect way - loop
t = (0:0.000005:1)';
t2 = clock;
for i=1:length(t)
    y(i) = sin(t(i));
```

```
end
timer2 = etime(clock,t2)
```

Save the m-file under a name of your choice (e.g. compare.m) then run it in the workspace. The result will be

```
» compare

timer1 =

    0.0600

timer2 =

    3.3500
```

(The results of course can differ from computer to computer). These mean that the first way of computing took 0.05 seconds, while the second way took 3.35 seconds. The second way is clearly inefficient!!!

Of course there will be occasions where you cannot avoid using the **for** loop especially when you are working with recursive problems. In this case, you should always assign an initial array of e.g. zeros that will be filled up as the loop progresses. We'll see examples of this in later stages of the course.

Next, always supress any unecessary outputs with the semicolon (;). If you want to see the outputs as the programme runs, beware that the speed of execution will be significantly higher. When you write your m-file start it with a command **clear all;** this will clear the memory and improve the performance of the processor. Last, although it's good to write comments (using %) in your m-file, these increase the execution time, as the computer actually "reads" the line but does not execute it. So, here you face the trade-off of computing time versus readability of your code. A good idea is to make two identical versions of your code one with and one without the comments, so that you can run the fast one and consult the more readable one!

14. Useful Sites on the Web

-The MathWorks Web site: http://www.mathworks.com/

-MATLAB Educational Sites: http://www.eece.maine.edu/mm/matweb.html

-Some MATLAB Links: http://math.uc.edu/~kingjt/MATLAB_lnk.html

-MATLAB resources on the web:

http://www.eeng.brad.ac.uk/help/.packlangtool/.maths/.MATLAB/.resource.html

-Online MATLAB Tutorials

http://mechanical.poly.edu/faculty/vkapila/MATLABtutor.htm

-One of the best tutorials I found on the web:

http://www.math.siu.edu/MATLAB/tutorials.html

15. Practice Exercises

1. Determine the size for the following vectors and matrices. Enter them in MATLAB and check your results using the 'whos' statement.

```
a = [1,0,0,0,0,1]
b = [2;4;6;10]
c = [5  3  5; 6  2  -3]
e = [3  5  10  0;  0  0 ...
     0  3;  3  9  9  8  ]
t = [4  24  9]
q = [t 0 t]
```

2. Define the 5 x 4 matrix, g.

```
g = [ 0.6  1.5  2.3 -0.5
      8.2  0.5 -0.1 -2.0
      5.7  8.2  9.0  1.5
      0.5  0.5  2.4  0.5
      1.2 -2.3 -4.5  0.5 ]
```

Determine the content and size of the following matrices and check your results for content and size using MATLAB.

```
a = g(:,2)
b = g(4,:)
c = [10:15]
d = [4:9;1:6]
e = [-5,5]
f= [1.0:-.2:0.0]
t1 = g(4:5,1:3)
```

3. Find the solution to

```
2x1 +  x2 - 4x3 + 6x4 + 3x5 - 2x6 = 16
-x1 + 2x2 + 3x3 + 5x4 - 2x5       = -7
 x1 - 2x2 - 5x3 + 3x4 + 2x5 +  x6 =  1
4x1 + 3x2 - 2x3 + 2x4       +  x6 = -1
3x1 +  x2 -  x3 + 4x4 + 3x5 + 6x6 = -11
5x1 + 2x2 - 2x3 + 3x4 +  x5 +  x6 =  5
```

using the matrix inverse and matrix division.

4.Create a ten-dimensional row vector whose all components are equal 2.

5. Let x = [2 5 1 6].

a. Add 16 to each element

b. Add 3 to just the odd elements

c. Compute the square root of each element

d. Compute the square of each element

6. Let x = [3 2 6 8]' and y = [4 1 3 5]'

a. Add the sum of the elements in x to y

b. Raise each element of x to the power specified by the corresponding element in y.

c. Multiply each element in x by the corresponding element in y, calling the result "z".

d. Add up the elements in z

7. Evaluate the following MATLAB expressions by hand and use MATLAB to check the answers

a. 2 / 2 * 3

b. 6 - 2 / 5 + 7 ^ 2 - 1

c. 10 / 2 \ 5 - 3 + 2 * 4

d. 3 ^ 2 / 4

e. 3 ^ 2 ^ 2

8. Create a vector x with the elements ...

a. 2, 4, 6, 8, ...

b. 10, 8, 6, 4, 2, 0, -2, -4

c. 1, 1/2, 1/3, 1/4, 1/5, ...

d. 0, 1/2, 2/3, 3/4, 4/5, ...

9. Plot the functions x, x^3, e^x and e^{x2} over the interval $0 < x < 4$. Put a title to each function and add some text inside the graphs.

10. Make a good plot (i.e., a non-choppy plot) of the function

f(x) = sin(1/x) for $0.01 < x < 0.1$.

11. Given x = [3 1 5 7 9 2 6], explain what the following commands "mean" by summarizing the net result of the command.

a. x(3)

b. x(1:7)

c. x(1:end)

d. x(1:end-1)

e. x(6:-2:1)

g. sum(x)

12. Given the array A = [2 4 1 ; 6 7 2 ; 3 5 9], provide the commands needed to

a. assign the first row of A to a vector called x1

b. assign the last 2 rows of A to an array called y

c. compute the sum over the columns of A

d. compute the sum over the rows of A

13. Given the arrays x = [1 4 8], y = [2 1 5] and A = [3 1 6 ; 5 2 7], determine which of the following statements will correctly execute and provide the result. If the command will not correctly execute, state why it will not. Using the command **whos** may be helpful here.

a. x + y

b. x + A

c. x' + y

d. A - [x' y']

e. [x ; y']

f. [x ; y]

g. A - 3

14. Given the array A = [2 7 9 7 ; 3 1 5 6 ; 8 1 2 5], explain the results of the following commands:

a. A'

b. A(:,[1 4])

e. A(:)

h. [A A(end,:)]

i. A(1:3,:)

j. [A ; A(1:2,:)]

m. sum(A,2)

15.Given the array A from problem 4, above, provide the command that will

a. assign the even-numbered columns of A to an array called B

b. assign the odd-numbered rows to an array called C

c. convert A into a 4-by-3 array

e. compute the square-root of each element of A

16. Provide the right answers and use MATLAB to check them.

a.
```
if n > 1
    m = n+1
else
    m = n - 1
end
```

a. n = 7 m = ?

b. n = 0 m = ?

c. n = -10 m = ?

b.
```
if z < 5
    w = 2*z
elseif z < 10
    w = 9 - z
elseif z < 100
    w = sqrt(z)
```

a. z = 1 w = ?

b. z = 9 w = ?

c. z = 60 w = ?

d. z = 200 w = ?

```
    else
       w = z
    end
```

c.
```
if T < 30
   h = 2*T + 1
elseif T < 10
   h = T - 2
else
   h = 0
end
```

a. T = 50 h = ?

b. T = 15 h = ?

c. T = 0 h = ?

d.
```
if 0 < x < 10
   y = 4*x
elseif 10 < x < 40
   y = 10*x
else
   y = 500
end
```

a. x = -1 y = ?

b. x = 5 y = ?

c. x = 30 y = ?

d. x = 100 y = ?

17. Given the vector x = [1 8 3 9 0 1], create a short set of commands that will (use loops for this)

a. Add up the values of the elements (Check the result with **sum**.)

b. computes the sine of the given x-values

18. Create an M-by-N array of random numbers (use **rand**). Move through the array, element by element, and set any value that is less than 0.2 to 0 and any value that is greater than (or equal to) 0.2 to 1. (use loops for this)

19. Given x = [4 1 6] and y = [6 2 7], compute the following arrays

a. $a_{ij} = x_i y_j$ (i.e, the element a(1,1) will be x1*y1)

b. $b_{ij} = x_i / y_j$

c. $c_i = x_i y_i$, then add up the elements of c.