5. OCTAVE

- *% comment line*

Basic Operations:

- 5+6
- 3-2
- 5*8
- 4/2
- 2^6 *power*

➢ **Logical**:

1==2 *%**false** = 0*

1~=2(*not equal*) *%**true** = 1*

➢ To change the **prompt string**: **PS1('>> ');**

➢ Semicolon **supresses** output: put **;** at the end of code line: o/p won't print

➢ Semicolon is also used to write two statements on same line

➢ Chaining with **" , "** is also possible, but it will print the o/p

≫ **Variables**:

a=3

b='hi' *%**single quotes** are used for strings*

c = (3>=1) *%c=1* *%true*

a=**pi** *%pi is predefined*

⇨To **print** a:

○ **a**	○ **disp**(a);

⇨ To **display** strings:

- disp(**sprintf**(' *pi to 2 decimal places: **%0.2f** '*, **a**))

  pi to 2 decimal places: 3.14

⇨ Format Shortcut:

- format **long** *%3.14159265358979*
- format **short** *% 3.1416*

MATRICES AND VECTORS:

Matrix: ; to change rows

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

   1   2
   3   4
   5   6
```

Using **secondary prompt:**

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

   1   2
   3   4
   5   6
```

Vectors:

Row vector:

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

   1   2   3
```

Column vector:

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

   1
   2
   3
```

- Range Vectors:

```
>> v = 1:0.1:2
v =

 Columns 1 through 7:

    1.0000    1.1000    1.2000    1.3000    1.4000    1.5000    1.6000

 Columns 8 through 11:

    1.7000    1.8000    1.9000    2.0000
```

It **starts at 1** and **increments** in steps of 0.1 upto 2:   1 & 2 are inclusive

```
>> v = 1:6
v =

   1   2   3   4   5   6
```

Special vector functions:

- **ones :**

```
>> ones(2,3)
ans =

   1   1   1
   1   1   1
```

```
>> C = 2*ones(2,3)
C =

   2   2   2
   2   2   2
```

```
>> C = [2 2 2; 2 2 2]
C =

   2   2   2
   2   2   2
```

- **zeros**:

```
>> w = zeros(1,3)
w =

   0   0   0
```

- rand(row, col): gives random numbers bw 0 and 1 using Uniform Distribution

```
>> w = rand(1,3)
w =

   0.91477   0.14359   0.84860

>> rand(3,3)
ans =

   0.390426   0.264057   0.683559
   0.041555   0.314703   0.506769
   0.521893   0.739979   0.387001
```

- randn(row, col): gives random no from galsian distribution with mean=0 and variance =1

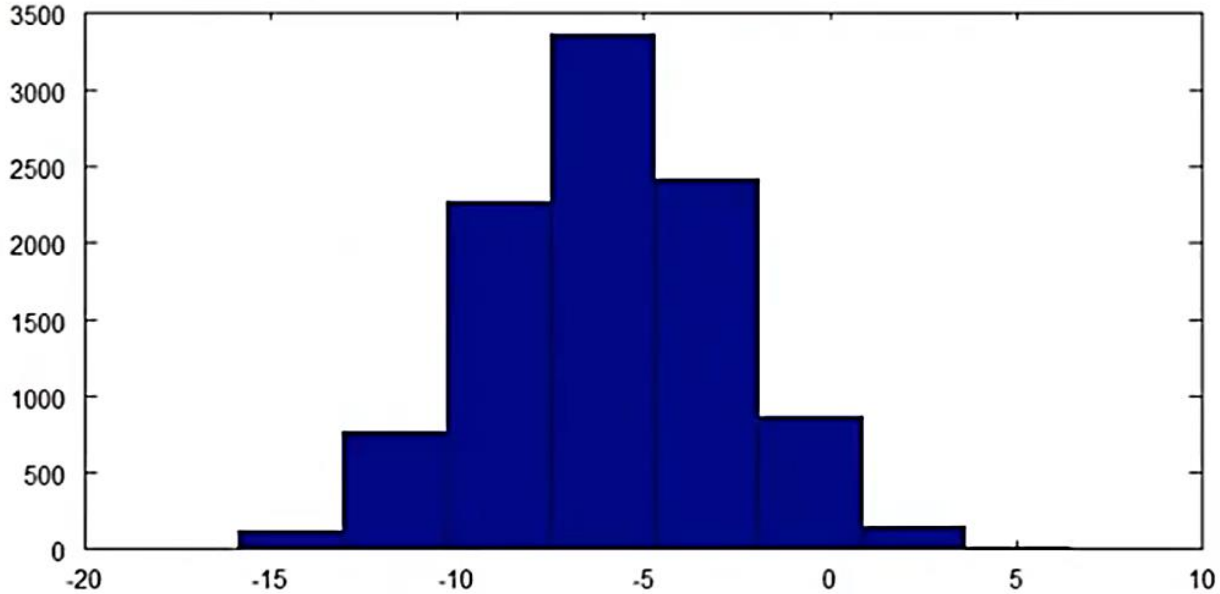
```
>> w = randn(1,3)
w =

  -1.44264  -1.27860  -0.69640

>> w = randn(1,3)
w =

  -0.33517   1.26847  -0.28211
```

- **Histograms:**

```
>> w = -6 + sqrt(10)*(randn(1,10000))
>> hist(w)
```

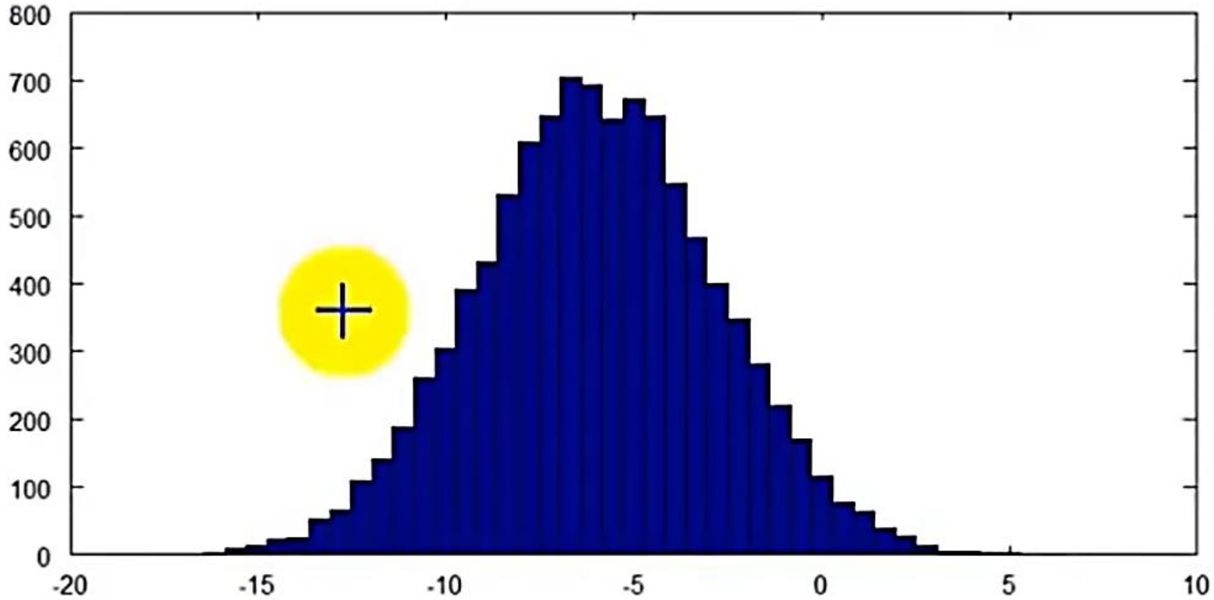
This plots the given matrix on a histogram.

Histograms mean value is = -6 in this case




```
>> hist(w,50)
```

: this incr the no of pins



- Identity matrix:

```
>> I = eye(4)
I =

Diagonal Matrix

   1   0   0   0
   0   1   0   0
   0   0   1   0
   0   0   0   1
```

⇨ help command:
- help eye
- help rand

⇨ help help

- size(matrix): return a 1x2 vector of size of another matrix

```
>> size(A)
ans =

   3   2
```

It can also be assigned as a vector:

```
>> sz = size(A)
sz =

   3   2
```

```
>> size(sz)
ans =

   1   2
```

➢ When only no of rows or cols of A are req:

```
>> size(A,1)
ans =  3
>> size(A,2)
ans =  2
```

length(matrix): returns the largest dimension; mostly used for vectors

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

   1   2   3   4

>> length(v)
ans =  4
```

```
>> length(A)
ans =  3
>> length([1;2;3;4;5])
ans =  5
```

⇨present working directory:

```
>> pwd
ans = C:\Octave\3.2.4_gcc-4.4.0\bin
```

⇨change dir:

```
>> cd 'C:\Users\ang\Desktop'
>> pwd
ans = C:\Users\ang\Desktop
```

listing files and folders:

```
>> ls
 Volume in drive C has no label.
 Volume Serial Number is 0C32-E0EC

 Directory of C:\Users\ang\Desktop

[.]                    [lectures-slides]    squareThisNumber.m
[..]                   matlab_session.m
costFunctionJ.m        [ml-class-ex1]
featuresX.dat          priceY.dat
               5 File(s)          8,071 bytes
               4 Dir(s)  406,465,044,480 bytes free
```

⇨loading files in octave:

- *featuresX.dat* and *priceY.dat* contains input and o/p data:
- this command will load the *file_name.extention* into a variable named file_name

```
>> load featuresX.dat
>> load priceY.dat
```

Or as string:

⇨
```
>> load('featuresX.dat')
```

⇨finding out variables that are currently assigned:

```
>> who
Variables in the current scope:

A           I           ans         c           priceY      v
C           a           b           featuresX   sz          w
```

Detailed view:

```
>> whos
Variables in the current scope:

  Attr Name           Size                     Bytes  Class
  ==== ====           ====                     =====  =====
       A              3x2                         48  double
       C              2x3                         48  double
       I              6x6                         48  double
       a              1x1                          8  double
       ans            1x2                         16  double
       b              1x2                          2  char
       c              1x1                          1  logical
       featuresX     47x2                        752  double
       priceY        47x1                        376  double
       sz             1x2                         16  double
       v              1x4                         32  double
       w              1x10000                  80000  double

Total is 10201 elements using 81347 bytes
```

⇨clear variable_name – deletes var from memory: type who to check

```
>> clear featuresX
```

⇨clear : clears every var

⇨Saving a variable into a file

```
>> save hello.mat v;
```

If we clear after this

Then if we load back the file “hello.mat”

It will load back the variable v… not hello

To save as human readable format:

```
>> save hello.txt v -ascii   % save as text (ASCII)
```

MATRIX DATA HANDLING:

Let:

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

     1     2
     3     4
     5     6
```

```
>> A(3,2)
ans =    6
```

returns element at row=3; col=2

Shorthand methods:

```
>> A(2,:)    % ":" means every element along that row/column
ans =

     3     4
```

Returns a vector

```
>> A(:,2)
ans =

     2
     4
     6
```

Getting multiple rows or columns:

```
>> A([1 3], :)
ans =

   1   2
   5   6
```

⇨It can also be used for assignment:

```
>> A(:,2) = [10; 11; 12]
A =

   1   10
   3   11
   5   12
```

It replaces the 2nd col of every row

in the matrix

⇨**Appending** a row or col:

```
>> A = [A, [100; 101; 102]];   % append another column vector to right
>> A
A =

     1    10   100
     3    11   101
     5    12   102
```

```
>> A(:)  % put all elements of A into a single vector
ans =

     1
     3
     5
    10
    11
    12
   100
   101
   102
```

Now let:

```
>> A = [1 2; 3 4; 5 6];
>> B = [11 12; 13 14; 15 16]
```

⇨joining 2 matrices column wise: == C = [A, B]

```
>> C = [A B]
C =

     1     2    11    12
     3     4    13    14
     5     6    15    16
```

⇨joining 2 matrices row wise:

```
>> C = [A; B]
C =

     1     2
     3     4
     5     6
    11    12
    13    14
    15    16
```

COMPUTATION ON DATA:

Matrix **multiplication**:

```
>> A*C
ans =

    5     5
   11    11
   17    17
```

Element-wise **multiplication**:

```
>> A .* B
ans =

   11    24
   39    56
   75    96
```

Element-wise **exponent**:

```
>> A .^ 2
ans =

    1     4
    9    16
   25    36
```

Elementwise **Reciprocal**:

```
>> 1 ./ A
ans =

   1.00000   0.50000
   0.33333   0.25000
   0.20000   0.16667
```

Reciprocation of a vector or matrices: element wise:

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

   1
   2
   3

>> 1 ./ v
ans =

   1.00000
   0.50000
   0.33333
```

Element-wise **logarithm**:

```
>> log(v)
ans =

   0.00000
   0.69315
   1.09861
```

Elementwise **exponent** with e %e^v

```
>> exp(v)
ans =

    2.7183
    7.3891
   20.0855
```

Element wise **absolute**:

```
>> abs([-1; 2;-3])
ans =

   1
   2
   3
```

Elementwise **negation**:

```
>> -v    % -1*v
ans =

  -1
  -2
  -3

>> v
v =

   1
   2
   3
```

Incrementing every element:

```
>> v
v =

   1
   2
   3

>> v + ones(length(v),1)
ans =

   2
   3
   4
```

Or

```
>> v + 1
ans =

   2
   3
   4
```

Transpose:

```
>> A'
ans =

   1   3   5
   2   4   6

>> (A')'
ans =

   1   2
   3   4
   5   6
```

Creating a float matrix:

```
>> a = [1 15 2 0.5]
a =

    1.00000   15.00000    2.00000    0.50000
```

Finding max element in a vector:

```
>> val = max(a)
val =  15
>> [val, ind] = max(a)
val =  15
ind =  2
```

val = value

ind = index -- starts from 1

Max element in a matrix:

```
A =

   1   2
   3   4
   5   6
```

```
>> max(A)
ans =

   5   6
```

Elemnt wise max

```
>> max(rand(3), rand(3))
ans =

   0.72763   0.78773   0.93872
   0.72363   0.83590   0.42763
   0.48315   0.41734   0.79961
```

Compares corresponding elements of both matrices

Column wise max:

```
A =

     8     1     6
     3     5     7
     4     9     2

>> max(A,[],1)
ans =

     8     9     7
```

1 is for column

Row wise max:

```
>> max(A,[],2)
ans =

     8
     7
     9
```

2 is for row wise

max(A) will do column wise max: by default

Absolute max in a matrix

```
>> max(max(A))
ans =   9
```

Or

```
>> max(A(:))
ans =   9
```

Element wise **comparison**:

```
>> a < 3
ans =

   1   0   1   1
```

Finding elements based on comparison:

```
>> find(a < 3)
ans =

   1   3   4
```

Magic matrix: every row/col/diagonal adds up to same no.

```
>> A = magic(3)
A =

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

Finding indices of elements based on conditions:

```
>> [r,c] = find(A >= 7)
r =

   1
   3
   2

c =

   1
   2
   3
```

Here r are the row indices and c are col indices

Corresponding to (1, 1) (3, 2) (2, 3)

Sum of all elements of a vector:

```
>> sum(a)
ans =  18.500
```

Column wise sum:

```
A =

   47   58   69   80    1   12   23   34   45
   57   68   79    9   11   22   33   44   46
   67   78    8   10   21   32   43   54   56
   77    7   18   20   31   42   53   55   66
    6   17   19   30   41   52   63   65   76
   16   27   29   40   51   62   64   75    5
   26   28   39   50   61   72   74    4   15
   36   38   49   60   71   73    3   14   25
   37   48   59   70   81    2   13   24   35

>> sum(A,1)
ans =

   369   369   369   369   369   369   369   369   369
```

1 is for column

Row wise sum:

```
>> sum(A,2)
ans =

   369
   369
   369
   369
   369
   369
   369
   369
   369
```

2 is for rows

Sum of **main diagonal**:

```
>> A .* eye(9)
ans =

    47     0     0     0     0     0     0     0     0
     0    68     0     0     0     0     0     0     0
     0     0     8     0     0     0     0     0     0
     0     0     0    20     0     0     0     0     0
     0     0     0     0    41     0     0     0     0
     0     0     0     0     0    62     0     0     0
     0     0     0     0     0     0    74     0     0
     0     0     0     0     0     0     0    14     0
     0     0     0     0     0     0     0     0    35

>> sum(sum(A.*eye(9)))
ans =   369
```

Sum of **secondary diagonal**:

```
>> flipud(eye(9))
ans =

Permutation Matrix

   0   0   0   0   0   0   0   0   1
   0   0   0   0   0   0   0   1   0
   0   0   0   0   0   0   1   0   0
   0   0   0   0   0   1   0   0   0
   0   0   0   0   1   0   0   0   0
   0   0   0   1   0   0   0   0   0
   0   0   1   0   0   0   0   0   0
   0   1   0   0   0   0   0   0   0
   1   0   0   0   0   0   0   0   0
```

```
>> sum(sum(A.*flipud(eye(9))))
ans =   369
```

Product of all elements of a vector:

```
>> prod(a)
ans =  15
```

Element wise **Floor** :

```
>> floor(a)
ans =

     1    15     2     0
```

Element wise **Ceil**:

```
>> ceil(a)
ans =

     1    15     2     1
```

Inverse of a matrix:

```
>> A = magic(3)
A =

   8   1   6
   3   5   7
   4   9   2

>> pinv(A)
ans =

   0.147222  -0.144444   0.063889
  -0.061111   0.022222   0.105556
  -0.019444   0.188889  -0.102778
```

pinv is a **pseudo inverse** function.. it gives just approximate inverse values:

```
>> temp = pinv(A)
temp =

   0.147222  -0.144444   0.063889
  -0.061111   0.022222   0.105556
  -0.019444   0.188889  -0.102778

>> temp * A
ans =

  1.0000e+000  1.5266e-016  -2.8588e-015
  -6.1236e-015  1.0000e+000  6.2277e-015
  3.1364e-015  -3.6429e-016  1.0000e+000
```

temp is inverse of A.. still temp * A is not **identity** matrix

its just nearly identical

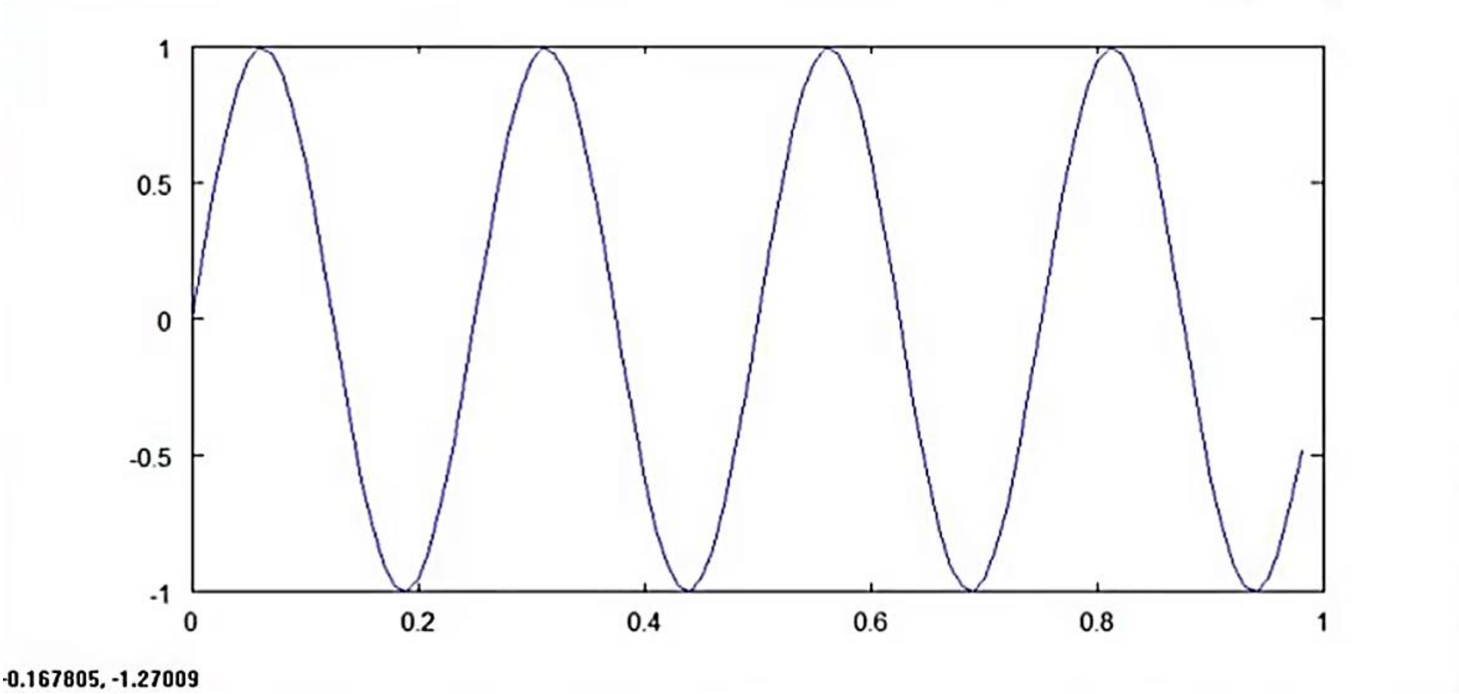
PLOTTING DATA:

Let:

```
>> t=[0:0.01:0.98];
```

Plotting a **sine** curve using the values in t

```
>> y1 = sin(2*pi*4*t);
>> plot(t,y1);
```



Similar for **cos**:

```
>> y2 = cos(2*pi*4*t);
>> plot(t,y2);
```

Hold on:

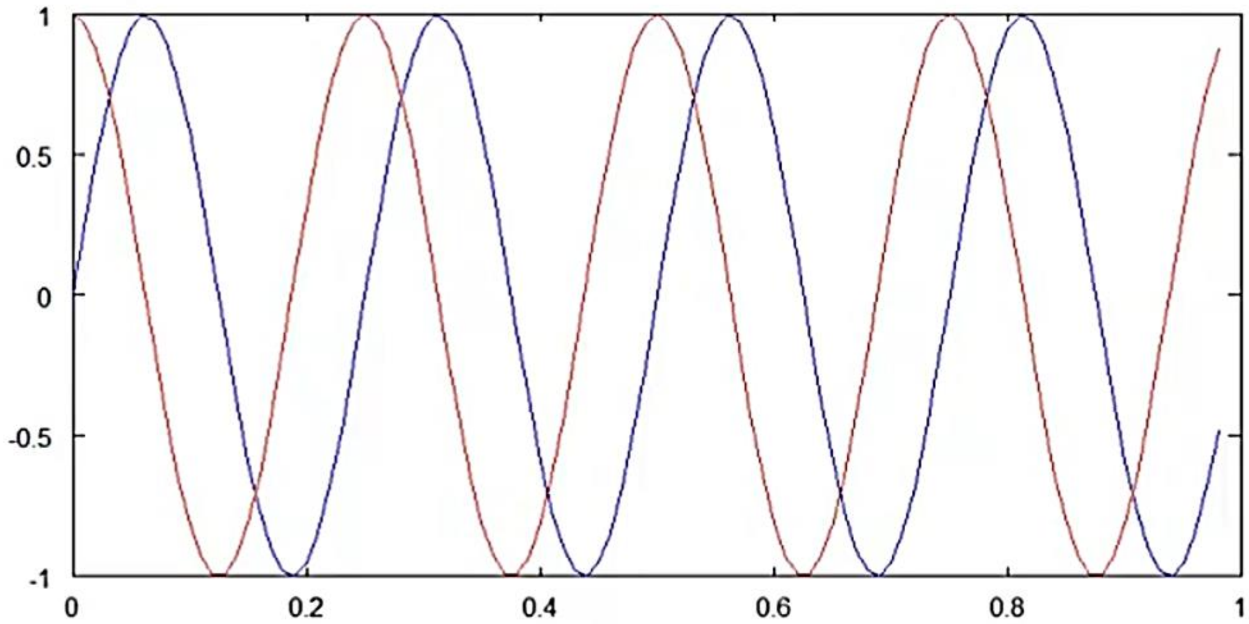
```
>> plot(t,y2);
>> plot(t,y1);
```

This will plot y2 first and then plot y1 but then y2 will disappear

So we use: hold on to plot another curve in the same window:

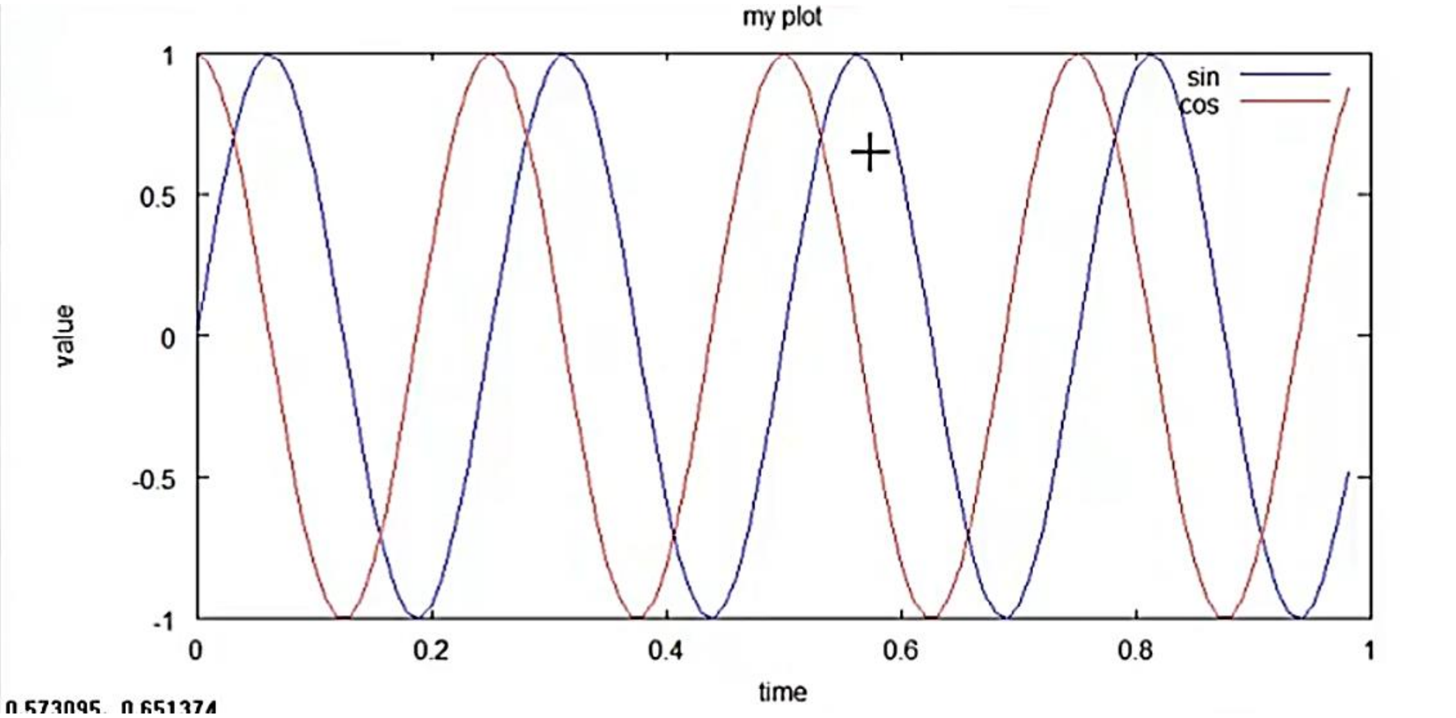
```
>> plot(t,y1);
>> hold on;
>> plot(t,y2,'r');
```

‘r’ is to use diff color -- red



Giving it **labels, legend** and **title**:

```
>> xlabel('time')
>> ylabel('value')
>> legend('sin', 'cos')
>> title('my plot')
```



Saving the plot into an **image file**:

```
>> cd 'C:\Users\ang\Desktop'; print -dpng 'myPlot.png'
```

Closing a figure:

$close

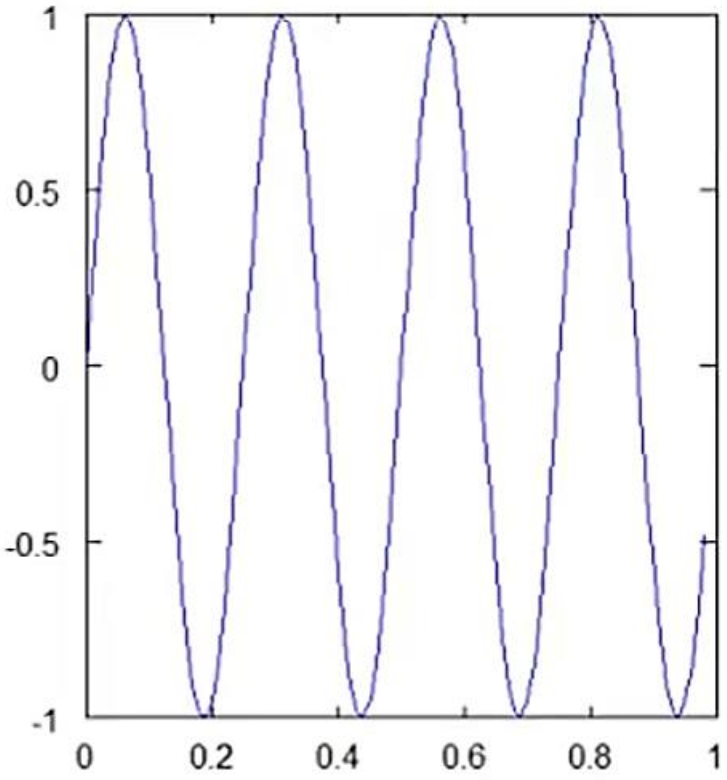
To plot more than one plots simultaneously but in diff windows:

Specify the figure number:

```
>> figure(1); plot(t,y1);
>> figure(2); plot(t,y2);
```

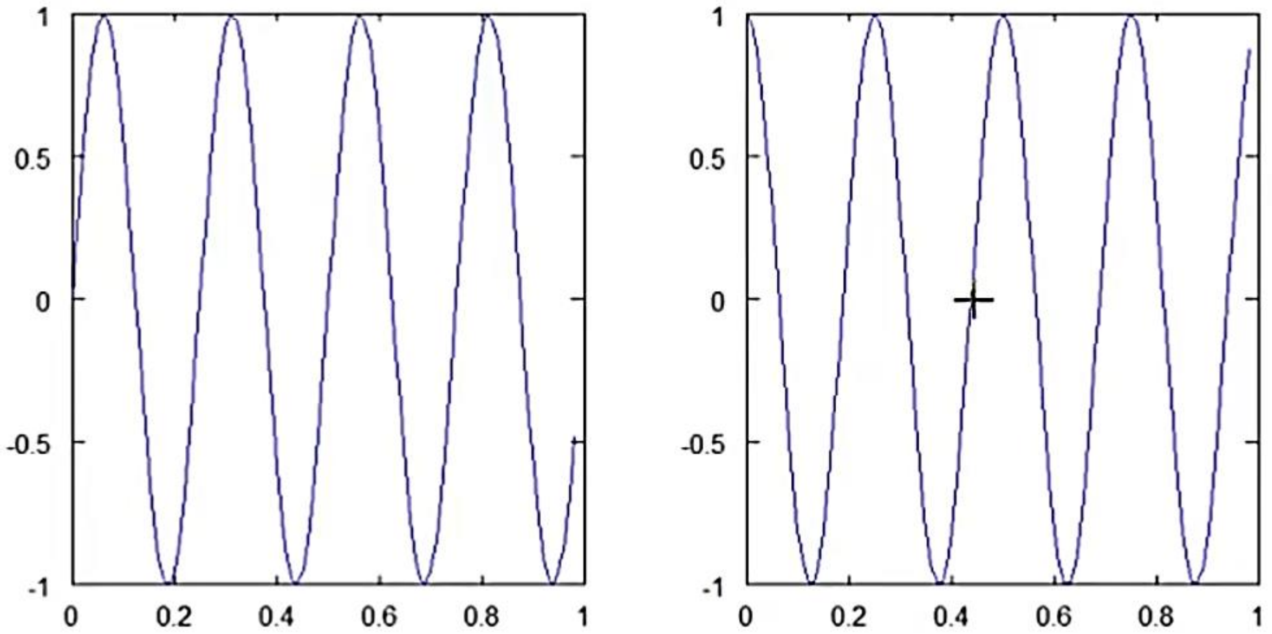
Subplots:

```
>> subplot(1,2,1); % Divides plot a 1x2 grid, access first element
>> plot(t,y1);
```



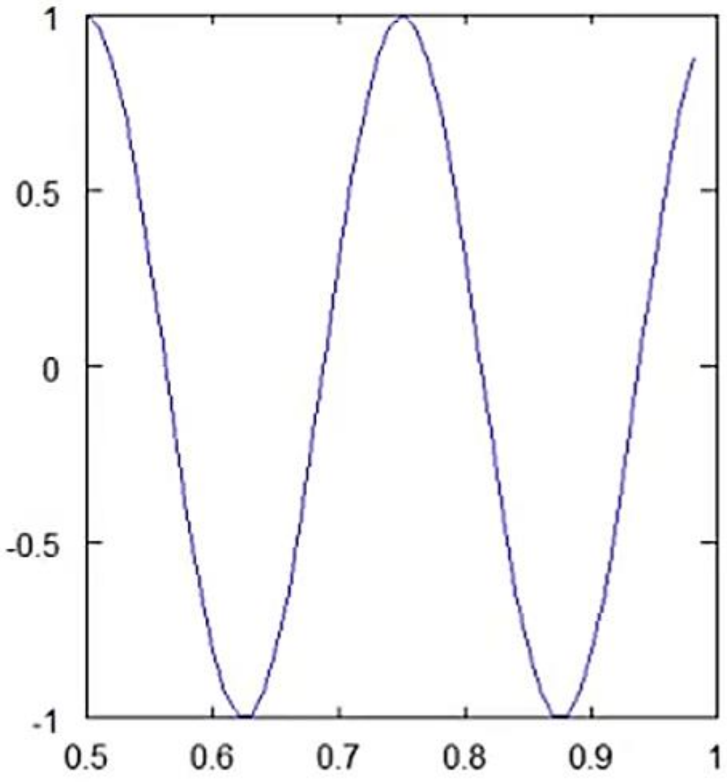
```
>> subplot(1,2,2);
>> plot(t,y2);
```

access the 2nd element



Setting axis ranges for plots:

```
>> axis([0.5 1 -1 1])
```

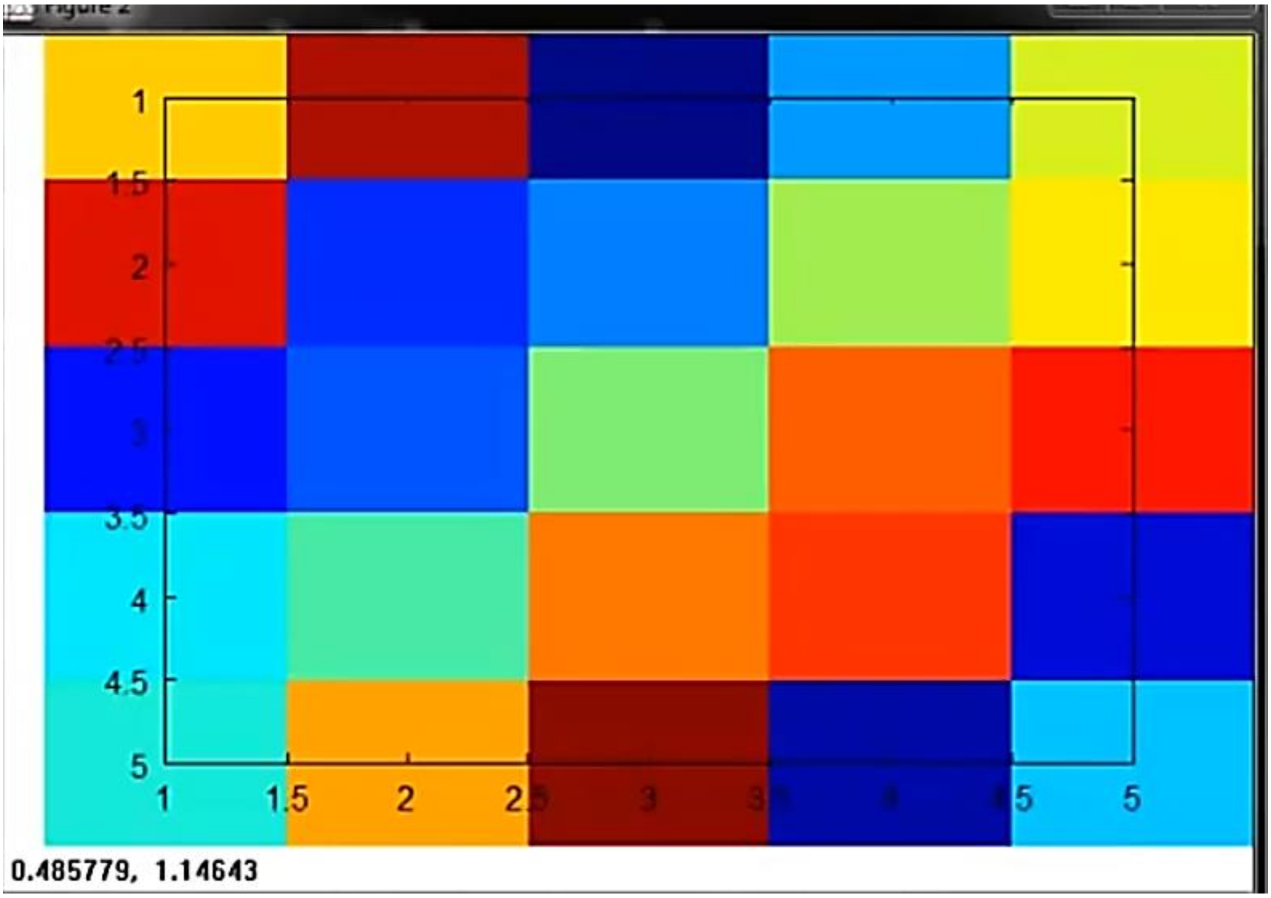


To clear all figures: **$clf**

To plot a matrix as a set of colours: each colour denote a value specified by range:

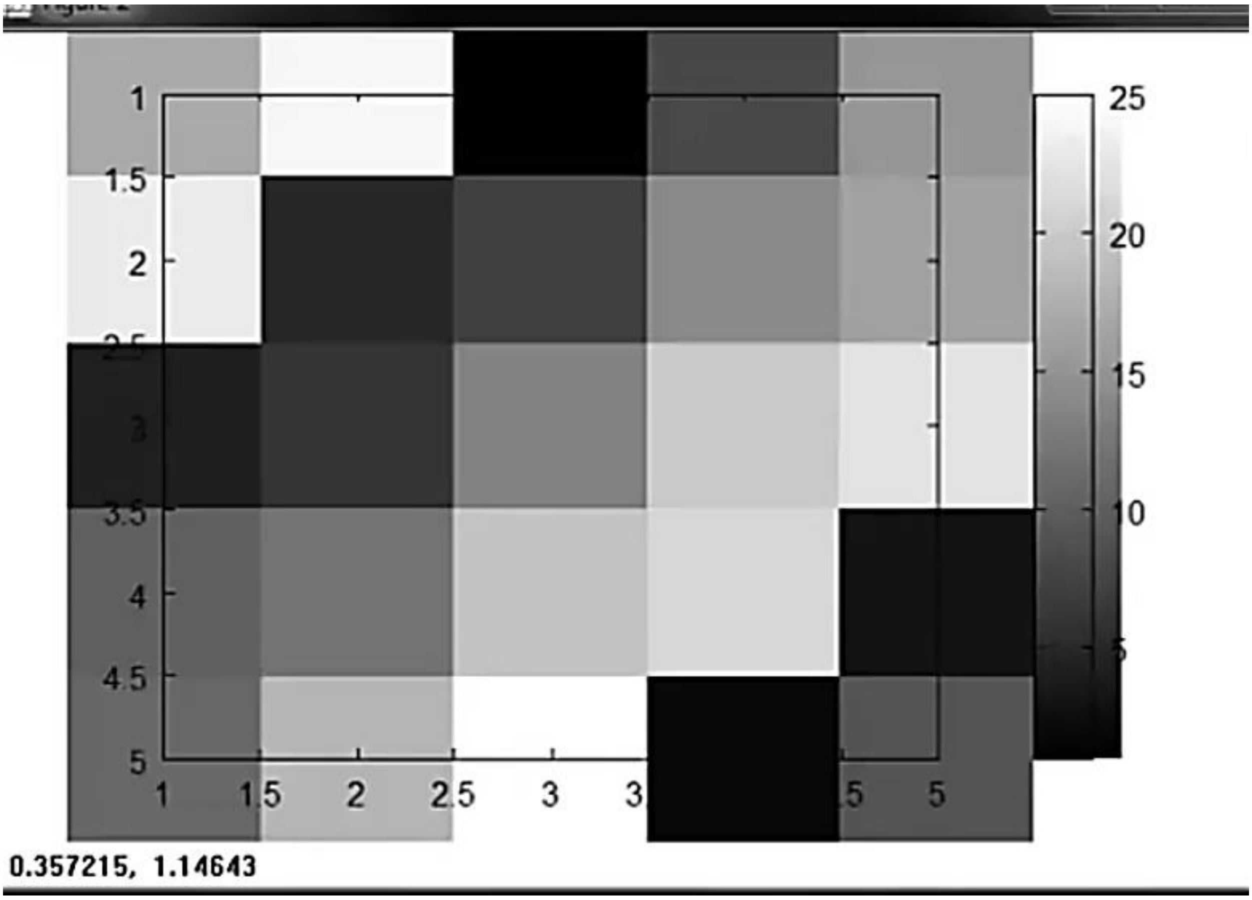
The range is given in colour bar

```
A =

    17    24     1     8    15
    23     5     7    14    16
     4     6    13    20    22
    10    12    19    21     3
    11    18    25     2     9

>> imagesc(A)
```



To bring it to greyscale: and to also bring out the colour bar:

```
>> imagesc(A), colorbar, colormap gray;
```



CONTROL STRUCTURES:

For loop: Let:

```
V =

     0
     0
     0
     0
     0
     0
     0
     0
     0
     0
```

```
>> for i=1:10,
>     v(i) = 2^i;
>   end;
>> v
v =

      2
      4
      8
     16
     32
     64
    128
    256
    512
   1024
```

Using list control:

```
>> indices=1:10;
>> indices
indices =

    1    2    3    4    5    6    7    8    9   10

>> for i=indices,
>     disp(i);
>   end;
 1
 2
 3
 4
 5
 6
 7
 8
 9
 10
```

While loop:

```
>> i = 1;
>> while i <= 5,
>     v(i) = 100;
>     i = i+1;
>   end;
>> v
v =

    100
    100
    100
    100
    100
     64
    128
    256
    512
   1024
```

If statements:

```
>> i=1;
>> while true,
>     v(i) = 999;
>     i = i+1;
>     if i == 6,
>       break;
>     end;
>   end;
>> v
v =

    999
    999
    999
    999
    999
     64
    128
    256
    512
   1024
```

an **infinite loop** is used.

If-elseif-else:

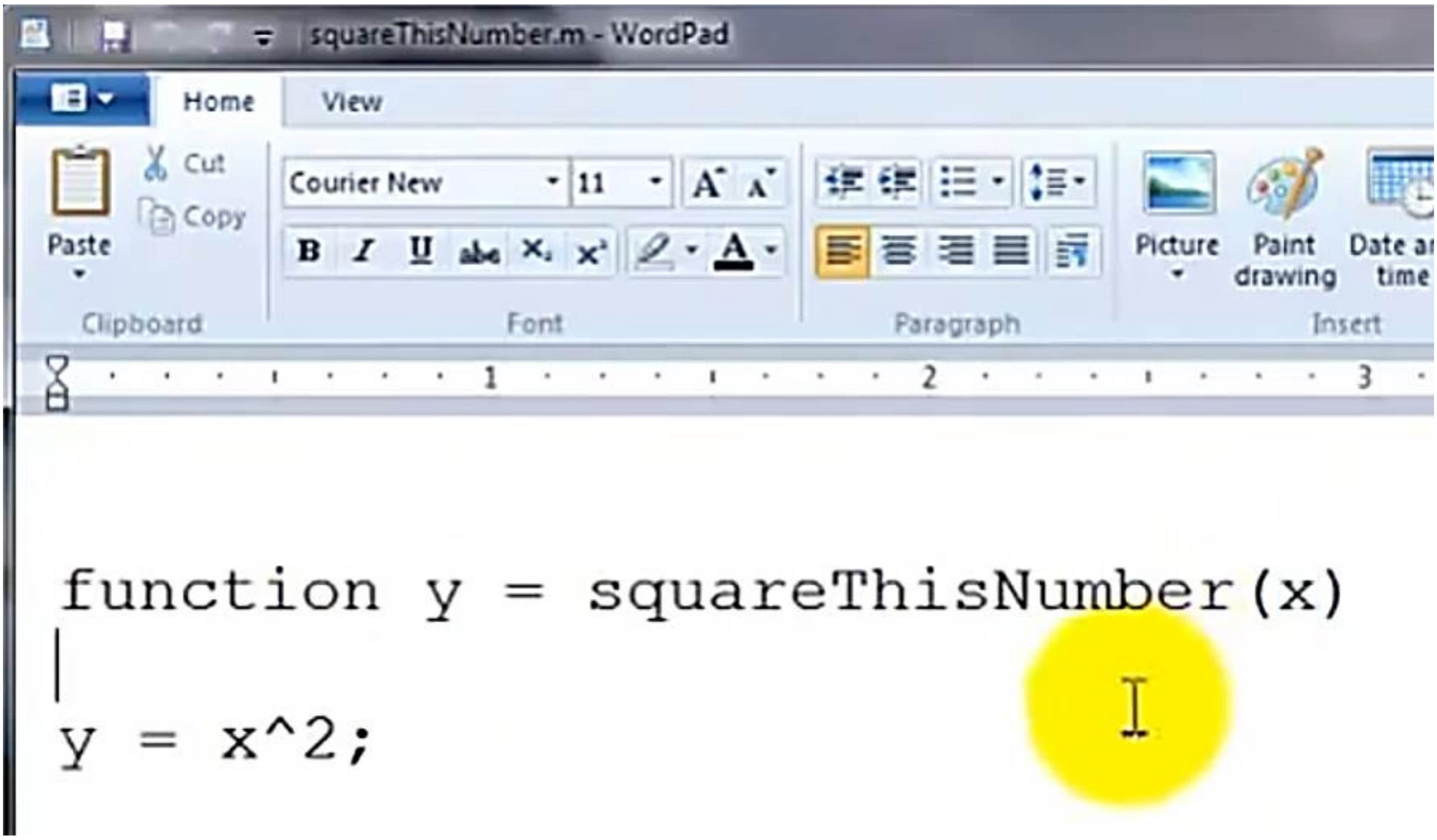
```
>> if v(1)==1,
>     disp('The value is one');
>   elseif v(1) == 2,
>     disp('The value is two');
>   else
>     disp('The value is not one or two.')
>   end;
The value is two
```

Exit: **$exit or $quit**

FUNCTIONS:

Functions are saved in different file in **pwd**: name of file = name of function

Extension= .m



Here, function y denotes that it’s a fxn and “y” is the return parameter

(x) = argument

Then the statements through which return parameters are calculated

---

To **execute** a fxn: cd to the location of file; and **call()**

```
>> squareThisNumber(5)
ans =   25
```

Or: we can just change the octave search path for fxns:

```
>> % Octave search path (advanced/optional)
>> addpath('C:\Users\ang\Desktop')
>> cd 'C:\'
>> squareThisNumber(5)
ans =   25
>> pwd
ans = C:\
```

Fxns can return multiple values:

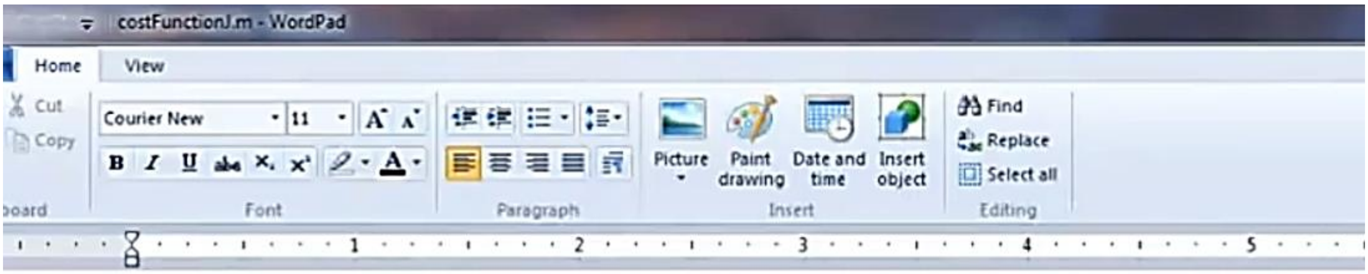

```
function [y1,y2] = squareAndCubeThisNumber(x)

y1 = x^2;
y2 = x^3;
```

```
>> [a,b] = squareAndCubeThisNumber(5);
>> a
a =  25
>> b
b =  125
```

Calculating the minimum cost error

Let:

```
>> X = [1 1; 1 2; 1 3]
X =

   1   1
   1   2
   1   3

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

   1
   2
   3

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




```
function J = costFunctionJ(X, y, theta)

% X is the "design matrix" containing our training examples.
% y is the class labels

m = size(X,1);             % number of training examples
predictions = X*theta;     % predictions of hypothesis on all m
examples
sqrErrors = (predictions-y).^2;   % squared errors


J = 1/(2*m) * sum(sqrErrors);
```

Here,

X → m x (n+1)   n=1

y → m x 1   o/p of all examples

theta = (n+1) x 1

predictions = X * Θ = m x 1 – here, hypothesis value of each example is calculated separately and stored in diff rows.

(prediction – y) – this id the term of diff bw predicted value and actual value

Both are m x 1 vectors: so it subtracts from corresponding elements.

sqrErrors -- elementwise sqr is used then all values are summed

and J is calculated

```
>> j = costFunctionJ(X,y,theta)
j = 0
```

J=0 means the chosen values of Θ perfectly fits the given data

```
>> theta = [0;0];
>> j = costFunctionJ(X,y,theta)
j =  2.3333
```

Here 2.333 = sum of sqr of all values of y divided by 2m

VECTORIZATION:

$$\underline{h_\theta(x)} = \sum_{\underline{j=0}}^{n} \theta_j x_j$$

$$= \underline{\theta^T x}$$

$$\underline{\Theta} = \begin{array}{l} \begin{bmatrix} \theta_0 \\ \theta_1 \\ \theta_2 \end{bmatrix} \end{array} \begin{array}{l} \text{theta(1)} \\ \text{theta(2)} \\ \text{theta(3)} \end{array} \qquad \underline{X} = \begin{bmatrix} x_0 \\ x_1 \\ x_2 \end{bmatrix}$$

Comparing implementations:

<u>Unvectorized implementation</u>

```
prediction = 0.0;
for j = 1:n+1,
   prediction = prediction +
                theta(j) * x(j)
end;
```

<u>Vectorized implementation</u>

```
prediction = theta' * x;
```

Gradient descent:

$$\theta_0 := \theta_0 - \alpha \frac{1}{m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)}) x_0^{(i)}$$

$$\theta_1 := \theta_1 - \alpha \frac{1}{m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)}) x_1^{(i)}$$

$$\theta_2 := \theta_2 - \alpha \frac{1}{m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)}) x_2^{(i)}$$

$$\underline{(n = 2)}$$

Vectorized implementation:

$$\Theta := \Theta - \alpha \delta$$

Here **Θ and δ are vectors**

$$\delta = \begin{bmatrix}\delta_0\\ \delta_1\\ \delta_2\end{bmatrix}$$

Where

$$\delta_0 = \boxed{\frac{1}{m}\sum (h_\theta(x^{(i)}) - y^{(i)}) \cdot x_0^{(i)}}$$

Dimensions:

$$\underset{\mathbb{R}^{n+1}}{\theta} := \overset{\mathbb{R}^{n+1}}{\theta} - \underset{\mathbb{R}}{\alpha}\, \overset{\mathbb{R}^{n+1}}{\delta}$$

$$\text{where} \quad \delta = \boxed{\frac{1}{m}\sum_{i=1}^{m} \underbrace{\boxed{(h_\theta(x^{(i)}) - y^{(i)})}}_{\mathbb{R}}\ \underbrace{\boxed{x^{(i)}}}_{\mathbb{R}^{n+1}}}$$

Where

$$\underbrace{(h_\theta(x^{(1)}) - y^{(1)}) \cdot x^{(1)} + (h_\theta(x^{(2)}) - y^{(2)}) \cdot x^{(2)} + \cdots}_{n+1}$$

$$\sum_{i=1}^{m} \boxed{(h_\theta(x^{(i)}) - y^{(i)})}\ \boxed{x^{(i)}}$$

Where

$$x^{(i)} = \begin{bmatrix} x_0^{(i)} \\ x_1^{(i)} \\ x_2^{(i)} \end{bmatrix}$$

The above multiplication of vectors is like :

For loop implementation:

$$u(j) = 2v(j) + 5w(j) \quad \text{(for all } j\text{)}$$

Vectorized implementation:

$$u = 2v + 5w$$
