Main layout of Matlab:

In the main layout of MATLAB, there are three windows, command window, workspace and command History.

Command window: All commands and functions are entered here.

Workspace: All created variables are listed here with their name, value, size and class. You can clear all variables in the workspace, with the `clear` command.

Command history: All commands entered in the command window are listed here. Commands can be dragged from here to the command window to execute again.

Matlab Help: Information and examples related to commands, functions, and other properties of Matlab can be found here. It is strongly recommended to take a look at this help service.

Editor (m-file editor) window: You can write several commands in an m-file and save this file. You can start the editor from the menu, by File, New, Script. You can open different m-files and they will appear as tabs in the editor window. There are line numbers on the left side of the editor. If there is an error about a command, Matlab shows the number of the line with the error. You can view a single m-file or more m-files by changing the window settings.

Variable editor: You can see the variables in this editor. You can start this editor by double clicking a variable from the workspace.

If some of the windows are missing in the main layout of Matlab, you can set using the desktop menu.

Basic Matlab commands:

You can use Matlab as a calculator:

```
>> (45 + 7)/3

ans =
   17.3333
```

Here, `ans` is temporary variable that stores the result of an instantaneous calculation. You can use the result in subsequent computations. Here are some examples:

```
>> 100^2-4*2*3
ans =
        9976

>> sqrt(ans)
ans =
   99.8799

>> (-100+ans)/4
ans =
   -0.0300
```

Here, `sqrt` is a built-in square root function. Matlab constitutes large number of predefined functions (`sin`, `cos`, `exp`, `log`, `tanh`, `abs`).

```
>> (cos(0.5))^2+(sin(0.5))^2
ans =
     1
>> exp(1)
ans =
    2.7183
>> log(ans)
ans =
     1
```

Like C language, the results of various arithmetic operations can be assigned to the variables that are defined. The following commands show how to enter numbers, vectors and matrices, and assign them to variables:

```
>> a = 2.12
a =
    2.1200
>> x = [1;2;3]
x =
     1
     2
     3
>> A = [1 2 3; 4 5 6; 7 8 9]
A =
     1     2     3
     4     5     6
     7     8     9
```

Notice that the rows of a matrix are separated by semicolons (;), while the entries on a row are separated by spaces (or commas). A useful command is `whos`, which displays the names of all defined variables and their types:

```
>> whos
  Name      Size            Bytes  Class     Attributes

  A         3x3                72  double
  a         1x1                 8  double
  ans       1x1                 8  double
  x         3x1                24  double
```

Here, the scalar `a` and `ans` are 1×1 arrays, the vector `x` is a 3×1 array and `A` is a 3×3 array. Note that the rules for picking up a valid name for an array (matrix/vector/scalar) are similar to the language C.

Matlab allows arrays to have complex entries. The complex unit $i = \sqrt{-1}$ is represented by either of the built-in variables `i` or `j`:

```
>> i
ans =
        0 + 1.0000i
>> -7.9+a*i
ans =
  -7.9000 + 2.1200i
```

These examples show complex numbers are displayed in Matlab. Most arithmetic operators as well as certain functions work for the matrices and vectors as well. For instance,

```
>> sin(x)
ans =
    0.8415
    0.9093
    0.1411
```

In this case, sine of each vector element is computed. A built-in variable that is often useful is π:

```
>> pi
ans =
    3.1416
```

The variable assignment is not limited with finite numbers in Matlab. One can assign infinity to variables:

```
>> a = inf
a =
   Inf

>> 1/a
ans =
     0
```

At this point, rather than providing a comprehensive list of functions and variables available in Matlab, it is explained how to get this information from Matlab itself. An extensive online help system can be accessed by commands of the form `help <command-name>`. For example:

```
>> help ans
 ANS Most recent answer.
    ANS is the variable created automatically when expressions
    are not assigned to anything else. ANSwer.
```

A good place to start is with the command `help help`, which explains how the help system works, as well as some related commands. Typing `help` by itself produces a list of topics for which help is available; looking at the list we find the entry “matlab\elfun - Elementary math functions.”. Typing `help elfun` produces a list of math functions available. You can also use `lookfor` command to search for unknown commands and functions.

It is often useful, when entering a vector/matrix/scalar, to suppress the display; this is done by ending the line with a semicolon (;). For instance type,

```
>> b = -9.75;
```

The scalar `b` is created in the workspace (i.e. in computer’s memory). However, the semicolon at the end of the statement suppresses the output.

At some point in time, it is possible to access certain elements of the matrices defined earlier. For instance, typing

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

will give the A matrix element (which is typed earlier!) at the second row and the third column. Similarly, if

```
>> A(:,2)
ans =
     2
     5
     8
```

is typed it is observed that Matlab has returned the entire second column of A. Here, colon (:) serves as a symbol to signal Matlab interpreter to omit the row numbers and to return the entire column. Likewise, when

```
>> A(3,:)
ans =
     7     8     9
```

is typed one gets the third row of elements of A. Matlab also enables the user to access submatrices:

```
>> A(1:2,2:3)
ans =
     2     3
     5     6
```

In this case, asking Matlab to consider the rows of A from 1 to 2, and the columns from 2 to 3.

Matlab enables the user to combine different matrices to form new ones. For instance, in this example:

```
>> B = [A; zeros(1,3); x']
B =
     1     2     3
     4     5     6
     7     8     9
     0     0     0
     1     2     3
```

A new matrix B (5×3) is created using A (3×3) and x (3×1) defined earlier. As can be noticed, some new features of Matlab is utilized in this example. The most important one is `zeros(n,m)` function which generates a n by m matrix full of zero element. The other one is (’) operator which takes the transpose of a given matrix (`transpose` is also possible). In this case, x (3×1) is transposed to form a new row vector (1×3). In the given example, a row vector full of zeros and the transposed x matrix is appended to our old matrix A. It should be noted how semi colons are utilized to combine these different Matlab variables. It is just like typing the elements of a vector. However, care must be exercised on the dimensions of these variables as they all must be conformable. This time, trying the following:

```
>> B = [x ones(3,1) A]
B =
     1     1     1     2     3
     2     1     4     5     6
     3     1     7     8     9
```

Once again, another important Matlab function: `ones(n,m)`. It is pretty much like `zeros` function but it creates an n by m matrix full of ones instead. Notice that this time; the semicolon is not used to separate out the elements. The thing done is very similar to typing a row vector. However, this time row vector elements are not scalars (1×1), but matrices/vectors having 3 rows.

Another useful Matlab function is eye(n,m). This function creates an n×m matrix, which has ones for the elements with equal row and column numbers, and zeros for the other elements.

```
>> eye(3,3)
ans =
     1     0     0
     0     1     0
     0     0     1
```

In order to delete some of the rows or columns of a matrix, we can use the following:

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

Here, the colon (:) operator is used for row, so the entire second column is selected. This column is assigned to a matrix with zero size and this operation deletes this column. The resulting A matrix is of size 3×2.

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

This time, the colon operator is used for column, so the first and second rows are selected together. These rows are deleted by assigning to a matrix with zero size. The resulting A matrix is of size 1×3.

Standard matrix operations:

It can be started by typing the following matrices:

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

Matlab enables the user to perform arithmetic operations (+, -, *). For example, consider the following commands:

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

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

Matrix multiplication is also defined:

```
>> A*C
ans =
     22    28
     49    64
     76   100

>> C*A
??? Error using ==> mtimes
Inner matrix dimensions must agree.
```

Just like the concepts in linear algebra, the matrices must be conformable to conduct arithmetic operations on them. Nevertheless, there are some exceptions in Matlab:

```
>> A-3
ans =
     -2    -1     0
      1     2     3
      4     5     6
```

Here, the matrix dimensions do not agree. However, scalars (1×1) are considered as major exceptions. The result obtained is that the scalar 3 is subtracted from every element of the matrix A. In some cases, it is not desired to perform regular matrix multiplication. Instead, it is wished to multiply the corresponding elements of two matrices. Considering the following example:

```
>> A.*B
ans =
      1     2     3
      8    10    12
     21    24    27
```

(.*) is the right operator for the job. You can see more operations and information about these operations by typing `help ops`.

One another important issue about matrices is solving linear equations using Matlab. It is known that is A is a square, nonsingular matrix, then the solution of the equation Ax=b is $x=A^{-1}b$. Matlab implements this operation with the backslash operator.

```
>> A = [1 1 -4; 2 -3 2; 5 -8 10];
>> b = [8; 1; 2];
>> x = A\b
x =
    5.5000
    3.5000
    0.2500

>> x = inv(A)*b
x =
    5.5000
    3.5000
    0.2500
```

As an alternative way, matrix inversion command of Matlab (`inv`) can be employed.
Determinant of a square matrix can be calculated using `det` command:

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

Rank of a matrix can be obtained using `rank` command:

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

Eigenvalues and eigenvectors of a square matrix can be calculated using `eig` command:

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

>> eig(A)
ans =
   16.6504
   -1.2730
   -0.3774
>> [V,D] = eig(A)
V =
   -0.7504   -0.4257    0.1475
   -0.3141    0.9041   -0.9475
   -0.5816    0.0368    0.2838
D =
   16.6504         0         0
         0   -1.2730         0
         0         0   -0.3774
```

Here, `eig(A)` returns the eigenvalues of A in a vector, while `[V,D] = eig(A)` returns two matrices V and D. V is matrix whose columns are eigenvectors of A, while D is a diagonal entries are eigenvalues. Note that, the first column of matrix V is the eigenvector corresponding to the eigenvalue that is given in the first column of the matrix D. The same also holds for the other columns.

Typing `help matfun` will give more information about specialized functions for matrices.

Vectors and graphs in Matlab:

Matlab includes a number of functions (commands) to create special matrices; the following command creates a row vector whose components increase arithmetically:

```
>> t = 1:5
t =
     1     2     3     4     5
```

The components can change by non-unit steps:

```
>> x = 0:0.1:1

x =
  Columns 1 through 5
         0    0.1000    0.2000    0.3000    0.4000
  Columns 6 through 10
    0.5000    0.6000    0.7000    0.8000    0.9000
  Column 11
    1.0000
```

A negative step is also allowed:

```
x = 0:-0.5:-2
x =
         0   -0.5000   -1.0000   -1.5000   -2.0000
```

The command `linspace` has similar results; it creates a vector with linearly spaced entries. Specifically, `linspace(a,b,n)` creates a vector of length n with entries
{a, a+(b-a)/(n-1), a+2(b-a)/(n-1), …}:

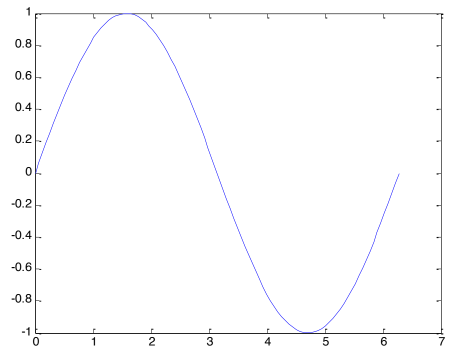
```
>> linspace(0,1,11)
ans =
  Columns 1 through 5
         0    0.1000    0.2000    0.3000    0.4000
  Columns 6 through 10
    0.5000    0.6000    0.7000    0.8000    0.9000
  Column 11
    1.0000
```

A vector with linearly spaced entries can be regarded as defining a one-dimensional grid, which is useful for graphing functions. To create a graph of y=f(x), one can create a grid in the vector x and then create a vector y with the corresponding function values.

It is easy to create the needed vectors to graph a built-in function since most Matlab functions are designed to return a vector (matrix). This means that if a built-in function such as sine is applied to an array, the effect of the function values of the entries of the original array. For example,

```
>> x = linspace(0,2*pi,100);
>> y = sin(x);
>> plot(x,y)
```

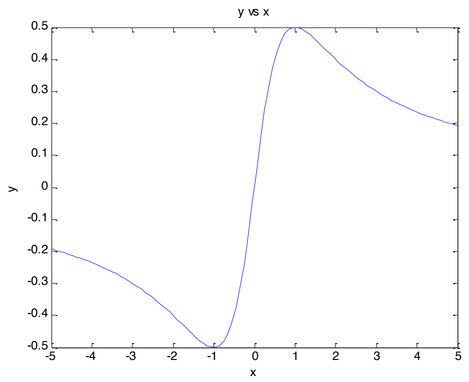
Matlab opens up a graphics window and displays the result:



As a second example, the following function is plotted

$y=x/(1+x^2)$ for $-5 \le x \le 5$. To graph, one should type the following:

```
>> x = linspace(-5,5,100);
>> y = x./(1+x.*x);
>> plot(x,y)
>> xlabel('x')
>> ylabel('y')
>> title('y vs x')
```



Here, arithmetic operator preceded by (.) is utilized. As discussed before, “dot” operators perform the arithmetic operation on the corresponding elements of two vectors (or matrices). In this case, firstly $1+x^2$ is calculated using array multiplication (.*). The result is a vector (1 by 100). Hence employing array division (./), the vector x is divided by this result in order to form the vector y. Typing `help plot` will give more information on this very useful command.

Note that, for axis and title labeling, `xlabel`, `ylabel` and `title` commands are used.

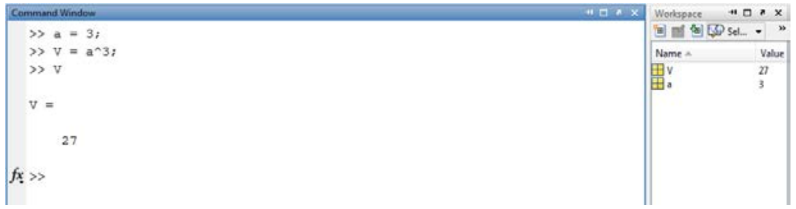
We can plot more than one plots on the same figure using `hold` command.

```
>> x = linspace(-5,5,100);
>> y = x./(1+x.*x);
>> z = x./(1+x.^4);
>> plot(x,y)
>> hold
Current plot held
>> plot(x,z,'--')
>> xlabel('x')
>> ylabel('y - z')
>> title('y vs x and z vs x')
>> legend('y','z')
```

Note that, Matlab uses the last open figure to plot. In order to make graph on different figures, you can use `figure` command, before each new figure.

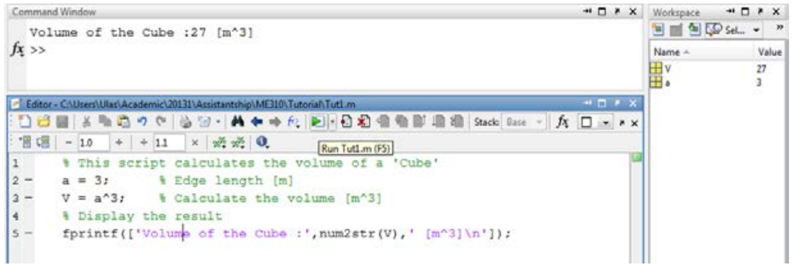
M-Files, or MATLAB Scripts

You can enter commands MATLAB command line one by one, or you can write a number of commands to a file so that you can execute them by calling that file.



```
>> a = 3;
>> V = a^3;
>> V

V =

    27

>>
```

Name	Value
V	27
a	3



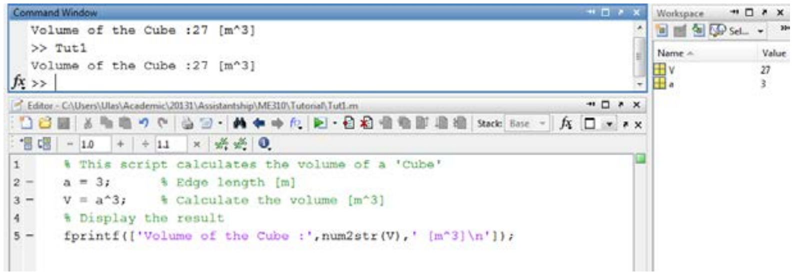
```
Volume of the Cube :27 [m^3]
>>
```

Editor - C:\Users\Ulas\Academic\20131\Assistantship\ME310\Tutorial\Tut1.m

```
% This script calculates the volume of a 'Cube'
a = 3;        % Edge length [m]
V = a^3;      % Calculate the volume [m^3]
% Display the result
fprintf(['Volume of the Cube :',num2str(V),' [m^3]\n']);
```

You may run your m-file in two ways:

i. Press F5, or press the run button on the toolbar of the Editor,
ii. Type the name of the m-file on your command window.

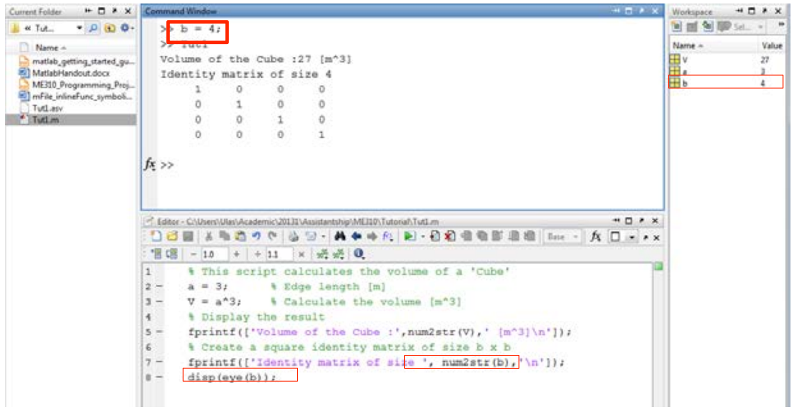


```
Volume of the Cube :27 [m^3]
>> Tut1
Volume of the Cube :27 [m^3]
>>
```

Editor - C:\Users\Ulas\Academic\20131\Assistantship\ME310\Tutorial\Tut1.m

```
% This script calculates the volume of a 'Cube'
a = 3;        % Edge length [m]
V = a^3;      % Calculate the volume [m^3]
% Display the result
fprintf(['Volume of the Cube :',num2str(V),' [m^3]\n']);
```

In an m-file you may create new data, or you may use the data already available in your workspace.



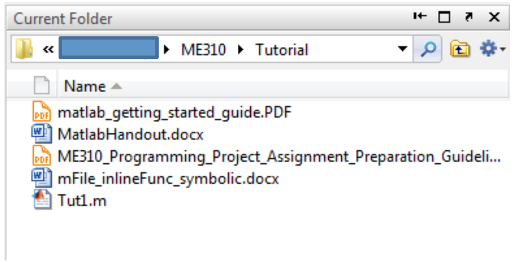
```
>> b = 4;
>> Tut1
Volume of the Cube :27 [m^3]
Identity matrix of size 4
     1     0     0     0
     0     1     0     0
     0     0     1     0
     0     0     0     1

>>
```

Editor - C:\Users\Ulas\Academic\20131\Assistantship\ME310\Tutorial\Tut1.m

```
% This script calculates the volume of a 'Cube'
a = 3;        % Edge length [m]
V = a^3;      % Calculate the volume [m^3]
% Display the result
fprintf(['Volume of the Cube :',num2str(V),' [m^3]\n']);
% Create a square identity matrix of size b x b
fprintf(['Identity matrix of size ', num2str(b),'\n']);
disp(eye(b));
```

Name	Value
V	27
a	3
b	4

You cannot run an m-file, which is not in the current directory (or folder). Either move your m-file to the current directory, or change the current directory of your MATLAB to the one containing your m-file. You can change your current directory by entering, or searching the address of the folder in the address bar in your *Current Folder Window.*

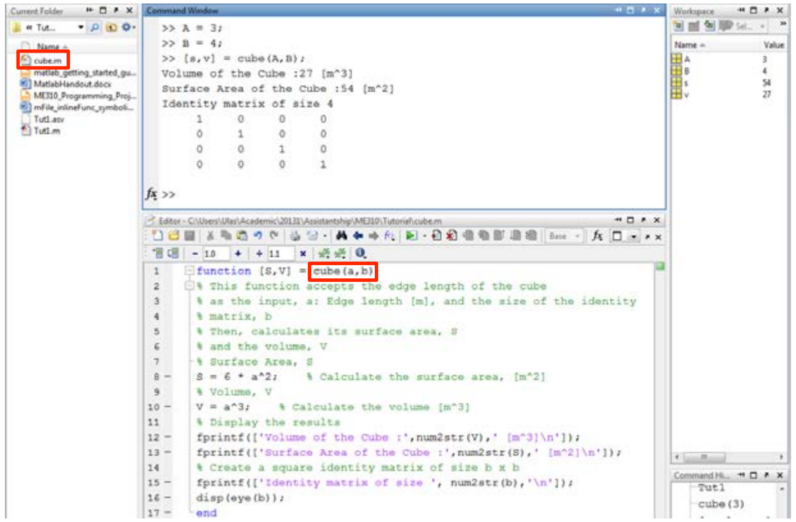


matlab_getting_started_guide.PDF
MatlabHandout.docx
ME310_Programming_Project_Assignment_Preparation_Guideli...
mFile_inlineFunc_symbolic.docx
Tut1.m

Functions

Functions accept inputs to be used in the calculations and return the resulting outputs.

- The names of the function and the file must be the same.
- `function [y1,...,yN] = myfun(x1,...,xM)` declares a function named myfun that accepts inputs `x1,...,xM` and returns outputs `y1,...,yN`. This declaration statement must be the first executable line of the function.



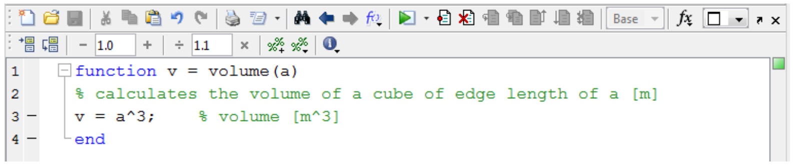
Current Folder
Tut...
Name
cube.m
matlab_getting_started_gu...
MatlabHandout.docx
ME310_Programming_Proj...
mFile_inlineFunc_symboli...
Tut1.asv
Tut1.m
Command Window
Workspace
Name
Value
A 3
B 4
s 54
v 27
Editor - C:\Users\Ulas\Academic\20131\Assistantship\ME310\Tutorial\cube.m
Command Hi...
Tut1
cube(3)

```
>> A = 3;
>> B = 4;
>> [s,v] = cube(A,B);
Volume of the Cube :27 [m^3]
Surface Area of the Cube :54 [m^2]
Identity matrix of size 4
     1     0     0     0
     0     1     0     0
     0     0     1     0
     0     0     0     1

>>
```

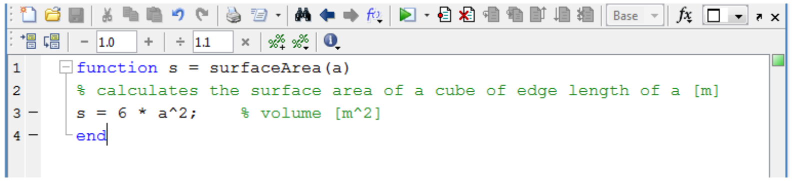
```
function [S,V] = cube(a,b)
% This function accepts the edge length of the cube
% as the input, a: Edge length [m], and the size of the identity
% matrix, b
% Then, calculates its surface area, S
% and the volume, V
% Surface Area, S
S = 6 * a^2;     % Calculate the surface area, [m^2]
% Volume, V
V = a^3;     % Calculate the volume [m^3]
% Display the results
fprintf(['Volume of the Cube :',num2str(V),' [m^3]\n']);
fprintf(['Surface Area of the Cube :',num2str(S),' [m^2]\n']);
% Create a square identity matrix of size b x b
fprintf(['Identity matrix of size ', num2str(b),'\n']);
disp(eye(b));
end
```

- You may call other functions located in the same current folder in a function.

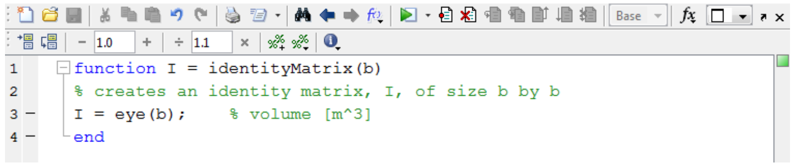
The function cube will call three different functions defined in the same folder, namely, *volume(), surfaceArea(), identityMatrix();*



```
function v = volume(a)
% calculates the volume of a cube of edge length of a [m]
v = a^3;     % volume [m^3]
end
```

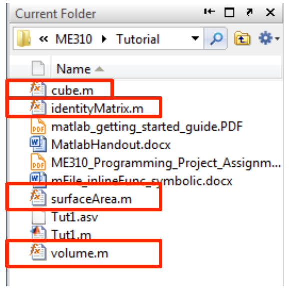


```
function s = surfaceArea(a)
% calculates the surface area of a cube of edge length of a [m]
s = 6 * a^2;     % volume [m^2]
end
```

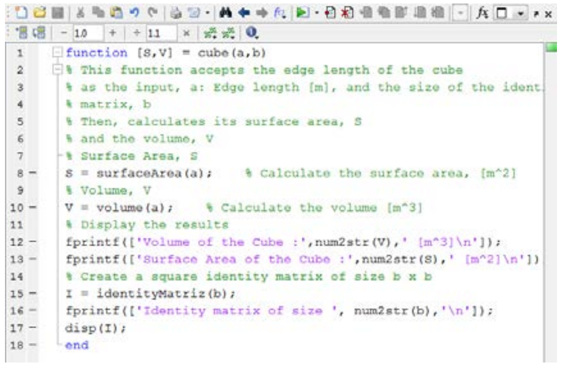


```
function I = identityMatrix(b)
% creates an identity matrix, I, of size b by b
I = eye(b);     % volume [m^3]
end
```

Those functions will be used by the function *cube()*. Again note that all these functions should be in the same folder.

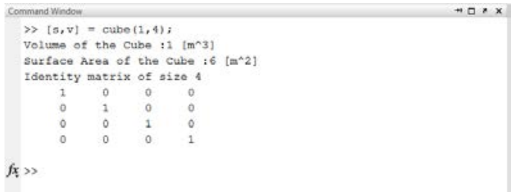


The functions *volume(), surfaceArea(), identityMatrix()* are called by *cube()* as follows:



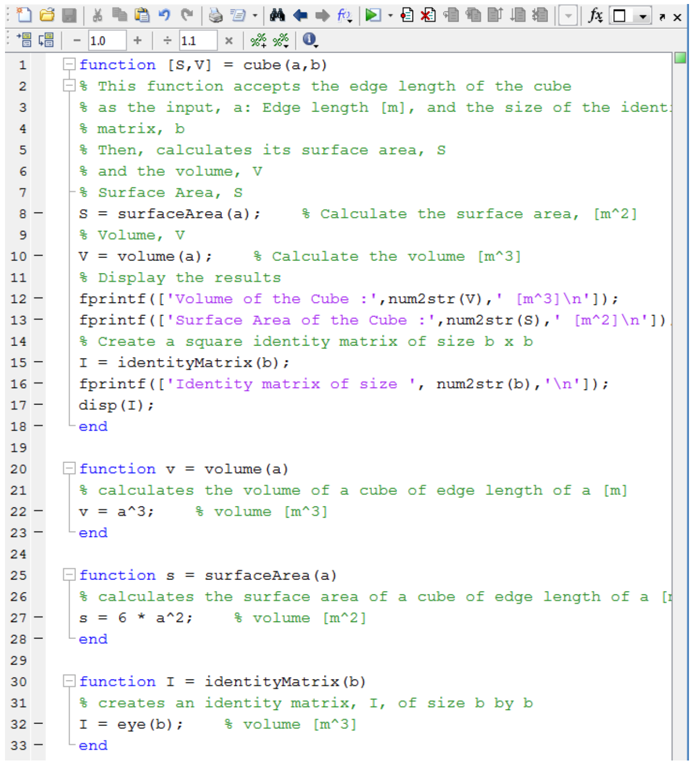
```
function [S,V] = cube(a,b)
% This function accepts the edge length of the cube
% as the input, a: Edge length [m], and the size of the ident
% matrix, b
% Then, calculates its surface area, S
% and the volume, V
% Surface Area, S
S = surfaceArea(a);     % Calculate the surface area, [m^2]
% Volume, V
V = volume(a);     % Calculate the volume [m^3]
% Display the results
fprintf(['Volume of the Cube :',num2str(V),' [m^3]\n']);
fprintf(['Surface Area of the Cube :',num2str(S),' [m^2]\n'])
% Create a square identity matrix of size b x b
I = identityMatriz(b);
fprintf(['Identity matrix of size ', num2str(b),'\n']);
disp(I);
end
```

The result is displayed in the ‘Command Window’:



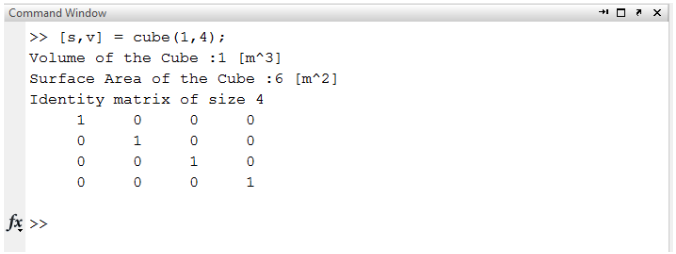
```
>> [s,v] = cube(1,4);
Volume of the Cube :1 [m^3]
Surface Area of the Cube :6 [m^2]
Identity matrix of size 4
     1     0     0     0
     0     1     0     0
     0     0     1     0
     0     0     0     1

fx >>
```

- You may define auxiliary functions in a main function, all written in the same file.



```
function [S,V] = cube(a,b)
% This function accepts the edge length of the cube
% as the input, a: Edge length [m], and the size of the ident
% matrix, b
% Then, calculates its surface area, S
% and the volume, V
% Surface Area, S
S = surfaceArea(a);     % Calculate the surface area, [m^2]
% Volume, V
V = volume(a);     % Calculate the volume [m^3]
% Display the results
fprintf(['Volume of the Cube :',num2str(V),' [m^3]\n']);
fprintf(['Surface Area of the Cube :',num2str(S),' [m^2]\n'])
% Create a square identity matrix of size b x b
I = identityMatrix(b);
fprintf(['Identity matrix of size ', num2str(b),'\n']);
disp(I);
end

function v = volume(a)
% calculates the volume of a cube of edge length of a [m]
v = a^3;     % volume [m^3]
end

function s = surfaceArea(a)
% calculates the surface area of a cube of edge length of a [
s = 6 * a^2;     % volume [m^2]
end

function I = identityMatrix(b)
% creates an identity matrix, I, of size b by b
I = eye(b);     % volume [m^3]
end
```

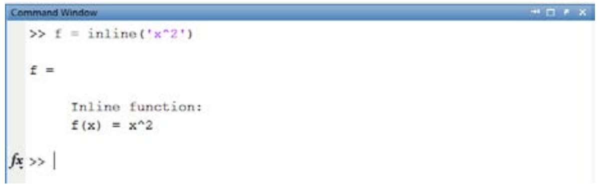
The function is called in the *Command Window* and the result can be seen in the following figure.



```
>> [s,v] = cube(1,4);
Volume of the Cube :1 [m^3]
Surface Area of the Cube :6 [m^2]
Identity matrix of size 4
     1     0     0     0
     0     1     0     0
     0     0     1     0
     0     0     0     1

fx >>
```

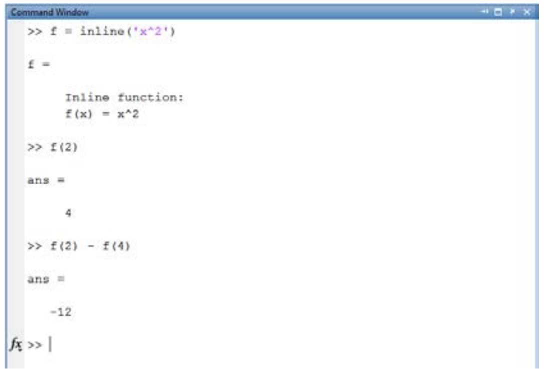
Inline Functions

You create objects representing analytical expressions by using *inline()* function of MATLAB. An example of an inline function is given in the following figure.



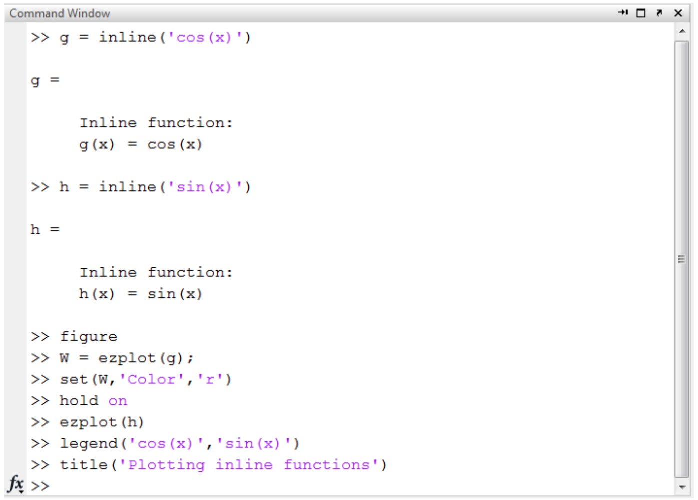
```
>> f = inline('x^2')

f =

     Inline function:
     f(x) = x^2

fx >> |
```

- As seen, the input argument is automatically determined, which is x in the above example.
- You may evaluate the value of the function at a specified point easily as follows;



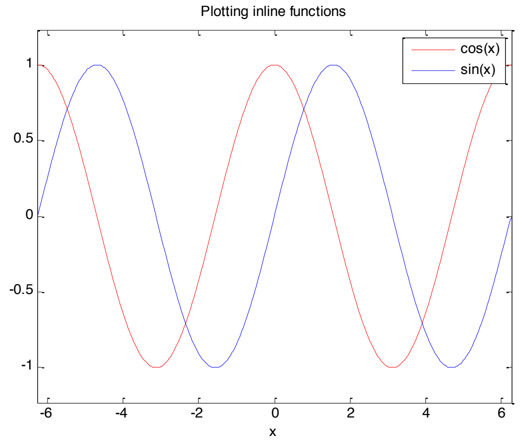
```
>> f = inline('x^2')

f =

     Inline function:
     f(x) = x^2

>> f(2)

ans =

     4

>> f(2) - f(4)

ans =

   -12

fx >> |
```

- Also you can easily obtain the plot of an inline function by using the built-in command of MATLAB, *ezplot():*



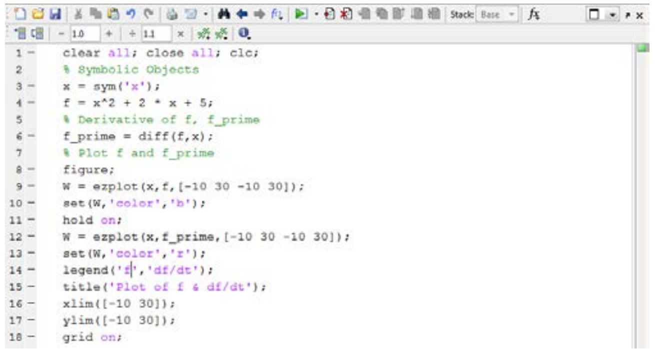
```
>> g = inline('cos(x)')

g =

     Inline function:
     g(x) = cos(x)

>> h = inline('sin(x)')

h =

     Inline function:
     h(x) = sin(x)

>> figure
>> W = ezplot(g);
>> set(W,'Color','r')
>> hold on
>> ezplot(h)
>> legend('cos(x)','sin(x)')
>> title('Plotting inline functions')
>>
```

The resulting plot is as follows,

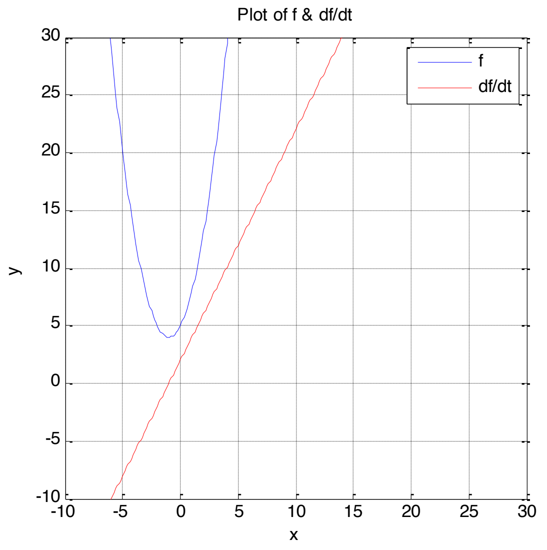


Symbolic Functions

You can also obtain analytic expressions by creating symbolic objects. A symbolic object is constructed using the function *sym().* The following example illustrates the use of the symbolic objects.



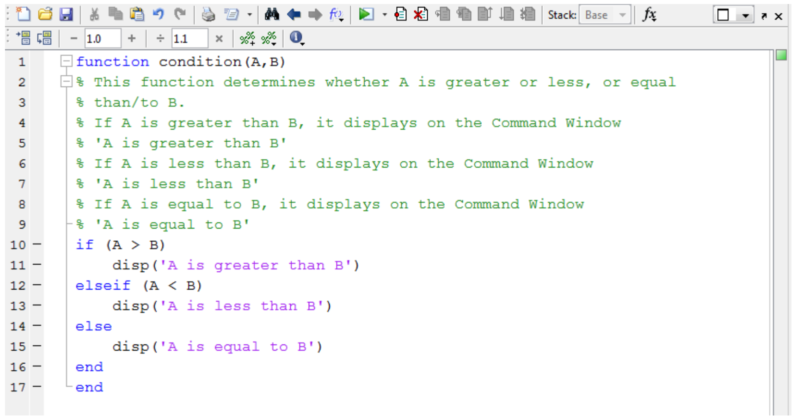
```
clear all; close all; clc;
% Symbolic Objects
x = sym('x');
f = x^2 + 2 * x + 5;
% Derivative of f, f_prime
f_prime = diff(f,x);
% Plot f and f_prime
figure;
W = ezplot(x,f,[-10 30 -10 30]);
set(W,'color','b');
hold on;
W = ezplot(x,f_prime,[-10 30 -10 30]);
set(W,'color','r');
legend('f','df/dt');
title('Plot of f & df/dt');
xlim([-10 30]);
ylim([-10 30]);
grid on;
```



Some Basics of Programming – Flow Control

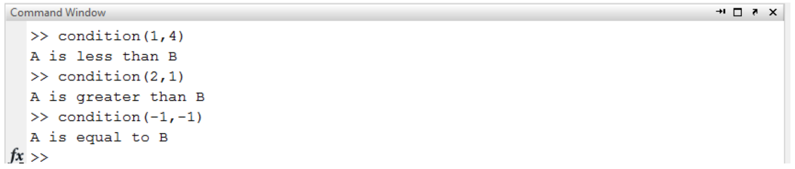
- **Conditional Control - If/elseif/else**

The following simple example illustrates the use of *if/elseif/else* statements in a function.



```
function condition(A,B)
% This function determines whether A is greater or less, or equal
% than/to B.
% If A is greater than B, it displays on the Command Window
% 'A is greater than B'
% If A is less than B, it displays on the Command Window
% 'A is less than B'
% If A is equal to B, it displays on the Command Window
% 'A is equal to B'
if (A > B)
    disp('A is greater than B')
elseif (A < B)
    disp('A is less than B')
else
    disp('A is equal to B')
end
end
```

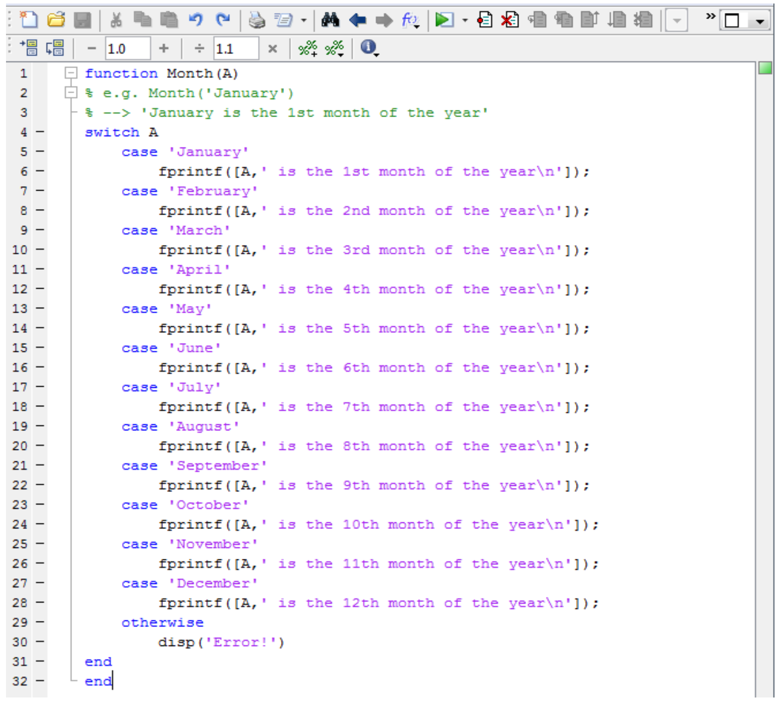
The above given function is tested as follows,



```
>> condition(1,4)
A is less than B
>> condition(2,1)
A is greater than B
>> condition(-1,-1)
A is equal to B
>>
```

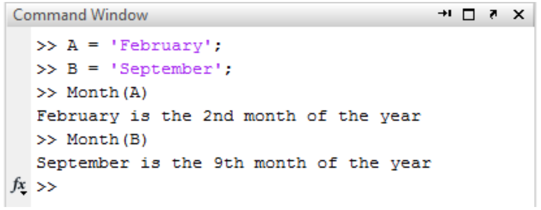
- **Conditional Control - switch/case/otherwise**

In *switch/case/otherwise* conditional control, a number of conditions are defined as the *case*s and what follows a *case* is the statement to be executed. The keyword *otherwise* contains whole the remaining conditions, that is the conditions which are not explicitly specified as a *case.* The statement *switch* directs the flow of the algorithm to the corresponding *case.* In the following example a function, named as *Month(A),* tells you the order of the month specified in argument *A* in a year.



```
function Month(A)
% e.g. Month('January')
% --> 'January is the 1st month of the year'
switch A
    case 'January'
        fprintf([A,' is the 1st month of the year\n']);
    case 'February'
        fprintf([A,' is the 2nd month of the year\n']);
    case 'March'
        fprintf([A,' is the 3rd month of the year\n']);
    case 'April'
        fprintf([A,' is the 4th month of the year\n']);
    case 'May'
        fprintf([A,' is the 5th month of the year\n']);
    case 'June'
        fprintf([A,' is the 6th month of the year\n']);
    case 'July'
        fprintf([A,' is the 7th month of the year\n']);
    case 'August'
        fprintf([A,' is the 8th month of the year\n']);
    case 'September'
        fprintf([A,' is the 9th month of the year\n']);
    case 'October'
        fprintf([A,' is the 10th month of the year\n']);
    case 'November'
        fprintf([A,' is the 11th month of the year\n']);
    case 'December'
        fprintf([A,' is the 12th month of the year\n']);
    otherwise
        disp('Error!')
end
end
```

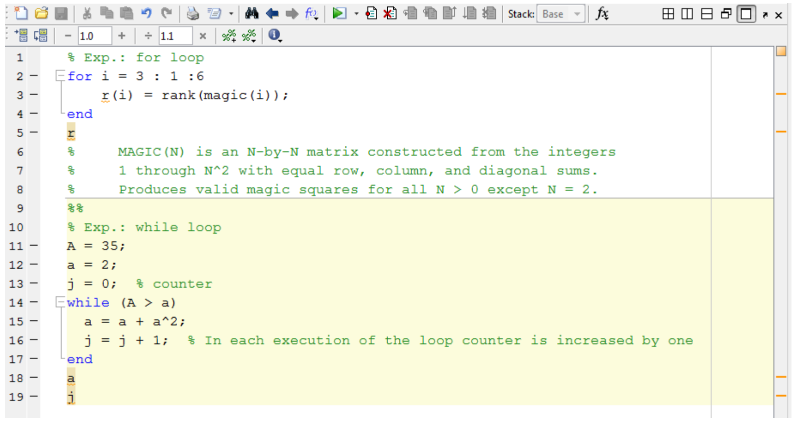
The given function is tested as follows:



```
>> A = 'February';
>> B = 'September';
>> Month(A)
February is the 2nd month of the year
>> Month(B)
September is the 9th month of the year
>>
```

- **<u>Loop Control – for/while</u>**

A group of statements can be executed as needed by using the *for/while loops.*The following two examples illustrate the use of *for* and *while loops.*



```
% Exp.: for loop
for i = 3 : 1 :6
    r(i) = rank(magic(i));
end
r
%     MAGIC(N) is an N-by-N matrix constructed from the integers
%     1 through N^2 with equal row, column, and diagonal sums.
%     Produces valid magic squares for all N > 0 except N = 2.
%%
% Exp.: while loop
A = 35;
a = 2;
j = 0;   % counter
while (A > a)
  a = a + a^2;
  j = j + 1;  % In each execution of the loop counter is increased by one
end
a
j
```

The *for loop* in the given example is executed 4 times for $i = 3,4,5,6$. On the otherhand, the *while loop* is executed until the condition $(A > a)$ holds. As you see unless the condition holds true in the given example, the program executes infinitely! Thus, the statement *break* allows you to exit early from *while and for loops.* For more information and an example about the *break* statement, type '*doc break'* in your command window.

<u>Reference</u>

MATLAB 7, Getting Started Guide,
http://www.mathworks.com/academia/student_version/learnmatlab.pdf , *last viewed on* 07.10.13.