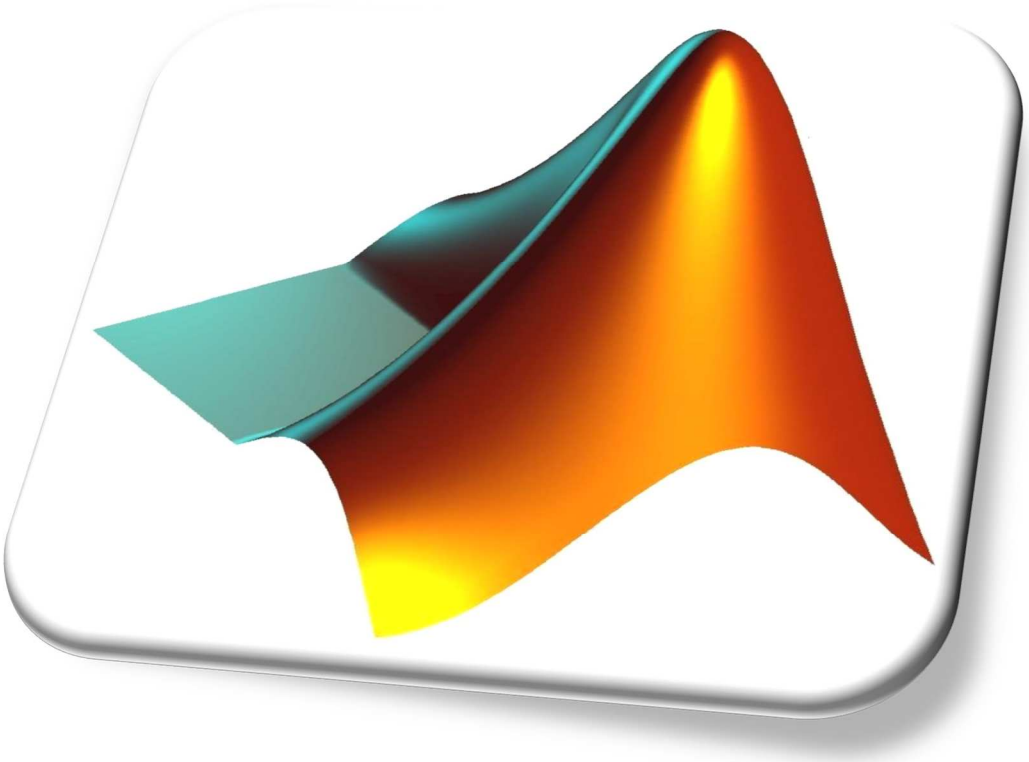
Programming



MATLAB

Mathematics and Computer Applications Dept.

2nd Stage / 1st Course

Chapter 1

Starting With MATLAB



MATLAB®(short for **MAT**rix **LAB**oratory) is a special-purpose computer program optimized to perform engineering and scientific calculations. It started life as a program designed to perform matrix mathematics, but over the years it has grown into a flexible computing system capable of solving essentially any technical problem.

The MATLAB program implements the MATLAB language and provides a very extensive library of pre-defined functions to make technical programming tasks easier and more efficient. This extremely wide variety of functions makes it much easier to solve technical problems in MATLAB than in other languages such as JAVA, FORTRAN, or C++.

1.1 The Advantages of MATLAB

- Ease of Use.
- Platform Independence.
- Pre-defined Functions
- Device-Independent Plotting.
- Graphical User Interface

1.2 Disadvantages of MATLAB

- The first is that it is an interpreted language and therefore can execute more slowly than compiled languages
- The second is cost: a full copy of MATLAB is five to ten times more expensive than a conventional C or FORTRAN compiler.

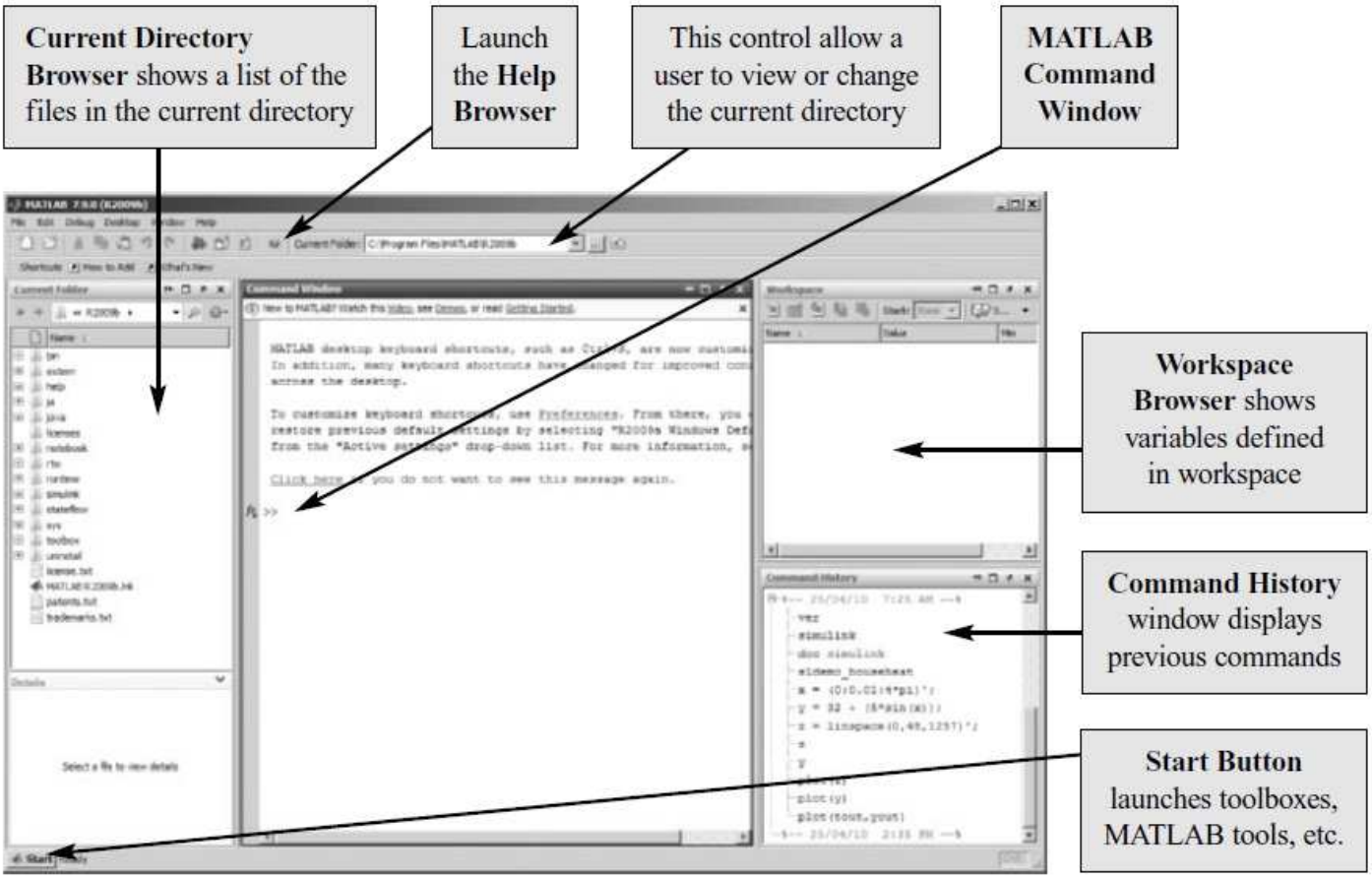
1.3 Uses of MATLAB

MATLAB is widely used as a computational tool in science and engineering encompassing the fields of physics, chemistry, math and all engineering streams. It is used in a range of applications including:

- Signal Processing and Communications
- Image and Video Processing
- Control Systems
- Test and Measurement
- Computational Finance
- Computational Biology

1.4 The MATLAB Environment

When MATLAB executes, it can display several types of windows that accept commands or display information. The *three* most important types of windows are ***Command Windows***, where commands may be entered; ***Figure Windows***, which display plots and graphs; and ***Edit Windows***, which permit a user to create and modify MATLAB programs. In addition, MATLAB can display other windows



Tool	Description
Command Window	A window where the user can type commands and see immediate results
Command History Window	A window that displays recently used commands
Start Button	The starting point for accessing MATLAB tools and resources
Current Directory Window	A window the displays MATLAB files, and allows the user to edit or debug them
Workspace Browser	A window that displays the names and values of variable stored in the MATLAB workspace
Variable Editor	Double-clicking on any variable listed in the workspace window to edit or add new variable
Figure Window	A window that displays a MATLAB plot
Edit Window	This window allows to type and save a series of commands without executing them.
Help Browser	A tool to get help for MATLAB functions

1.5 Commands in MATLAB

By clicking on the ”Command Window”, after the MATLAB prompt **>>,** a command can be typed. After having pushed the ‘***Enter***’ key, the command is executed, and possibly the result appears on the screen. A command and its result looks like this:

```
>> 1 + 2 + 3
ans =
     6
```

The result of 1 + 2 + 3 is automatically assigned to the variable ***ans***. You can then use ***ans*** further.

```
>> ans*ans
ans =
       36
```

Arithmetic operations		
Operation	**Symbol**	**Example**
Addition	+	**9 + 4**
Subtraction	-	**9 – 4**
Multiplication	*	**9 * 4**
Right division	/	**9 / 4**
Left division	\	**9 \ 4 = 4 / 9**
Exponentiation	^	**9 ^ 4**

In an expression with several operations, higher-precedence operations are executed ***before*** lower-precedence operations, for the ***same*** precedence operations the execution will be from left to right.

Precedence	Operation
1st.	**( )**
2nd.	^
3rd.	*** , /**
4th.	**+ , -**

```
 >> 4+8/2
ans =
    8
 >> (4+8)/2
ans =
    6
 >> 6*2/3
ans =
    4
```

```
>> 10/2^4
ans =
   0.6250
>> 27^ (1/3)
ans =
    3
>> 27^1/3
ans =
    9
```

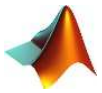
EXERCISE 1 Type the following expressions into MATLAB at the command prompt, and observe the results:

1. 5-2
2. 5 * 2
3. 5/2
4. 3 - 2 * 14 – 32
5. 2.54 * 8\2.6
6. 6.3 - 2.1045
7. 3.6^2
8. 1-2^2
9. $\sqrt[3]{6^2}$
10. $12.4 - \frac{5+2.5}{6-4.5}$
11. $\frac{\sqrt{7+\frac{5}{3}}}{3^{\sqrt{2}}}$

MATLAB supports the following special variables and constants:

Special Variables and Constants	
ans	**Most recent answer**
i,j	**The imaginary unit $\sqrt{-1}$**
Inf	**Infinity**
NaN	**Undefined numerical result (not a number)**
eps	**Accuracy of floating-point precision=(2^{-52})**
pi	**The number π=3.14159…**

```
>> i
ans =
      0 + 1.0000i
>> j
ans =
      0 + 1.0000i
>> pi
```



```
ans =
    3.1416
 >> eps
ans =
  2.2204e-016

>> 1/0
ans =
   Inf
  >> inf/inf
ans =
   NaN
```

MATLAB provides various **commands for managing a session**. The following table provides such commands:

Command	Description	Example
clc	**Clears** Command window	>> clc
clear	**Removes** variables from memory.	>> clear >> clear **x y**
help	Searches for a help topic	>> help **log**
lookfor	Searches help entries for a keyword	>> lookfor **log**
who	Lists current variables	>> who
whos	Lists current variables with details	>> whos
quit	Exit from MATLAB	>> quit

1.6 Elementary Math Built-In Functions

1.6.1 Exponential Functions

Function	Description	Example
exp(x)	e^x	**>> exp(2) ans = 7.3891**
log(x)	**ln**	**>> log(100) ans = 4.6052**
log10(x)	**log**	**>> log10(100) ans = 2**
sqrt(x)	$\sqrt{x}$	**>> sqrt(169) ans = 13**
nthroot(x,n)	$\sqrt[n]{x}$	**>> nthroot(96,4) ans = 3.1302**
factorial(n)	**n!**	**>> factorial(8) ans = 40320**



1.6.2 Rounding and Remainder Functions

Function	Description	Example
abs(x)	$\|x\|$	>> abs(-5.5) ans = 5.5000
mod(x , y)	Modulus after division	>> mod(13,7) ans = 6
round(x)	Round to nearest integer	>> round(5.5) ans = 6
fix(x)	Round toward zero	>> fix(5.5) ans = 5
ceil(x)	Round toward ∞	>> ceil(5.5) ans = 6
floor(x)	Round toward $-\infty$	>> floor(5.5) ans = 5

1.6.3 Trigonometric Functions

Function	Description	Example
sin(x) , cos(x) , tan(x) cot(x) , sec(x) , csc(x)	Sine ,cosine, tangent Cotangent, secant ,cosecant (In *radian*)	>> sin(pi/3) ans = 0.8660 >> tan(pi/4) ans = 1.0000 >> csc(pi/4) ans = 1.4142
sind(x) , cosd(x) , tand(x) cotd(x) , secd(x) , cscd(x)	Sine ,cosine, tangent Cotangent, secant ,cosecant (In *degrees*)	>> sind(60) ans = 0.8660 >> tand(45) ans = 1.0000 >> cscd(45) ans = 1.4142
asin(x) , acos(x) , atan(x) acot(x) , asec(x) , acsc(x)	sin^{-1}, cos^{-1} , tan^{-1} cot^{-1} , sec^{-1} , csc^{-1} *(in radian)*	>> asin(0.5) ans = 0.5236 >> atan(1) ans = 0.7854 >> acsc(-1) ans = -1.5708
asind(x) , acosd(x) , atand(x) acotd(x) , asecd(x) , acscd(x)	sin^{-1}, cos^{-1} , tan^{-1} cot^{-1} , sec^{-1} , csc^{-1} *(in degrees)*	>> asind(0.5) ans = 30 >> atand(1) ans = 45 >> acscd(-1) ans = -90



Notes:

- The hyperbolic trigonometric functions are **sinh(x) , cosh(x) , tanh(x) , coth(x) , sech(x) and csch(x)** for angle in ***radian*** .
- The Inverse hyperbolic trigonometric functions are **asinh(x) , acosh(x) , atanh(x) , acoth(x) , asech(x) and acsch(x)** for angle in ***radian*** .

1.7 Variables in MATLAB

Variable names consist of a **letter** followed by any *number* of letters, digits or underscore such as (**x** , **a2** , **result** , **total_price**). MATLAB is **case-sensitive.** For example (**X** and **x** are different variables)

Variable names can be of any length, however, MATLAB uses only first **N** characters, where **N** is given by the function ***namelengthmax***. For examples:

```
>> a=3          % defining a and initializing it with a value
a =
   3
>> a=a^2-1     % assigned new value to a and initializing it with an expression
a =
   8
```

- Once a variable is **entered** into the system, you can **refer** to it **later**.
- Variables **must** have **values before** they are **used**.
- When an expression returns a result that is not assigned to **any** variable, the system assigns it to a variable named **ans**, as mentioned before.
- It can follow a command with a **semicolon (;)** and the Command window output is **suppressed**, for example

```
>> % Use a semicolon (;) to hide the results after press Enter key
>> a=10;
>> b=4;
>> c=a+b^2;
>> % display the value of any variable by typing its name
>> c
c =
   26
```

- You can have **multiple assignments** on the same line. For example

```
>> x=3;  y=2;  z=(x-y)/(x+y)
z =
   0.2000
```



- **Long assignments** can be extended to another line by using an **ellipses** (**...**). For example

```
>> initial_velocity =0;  acceleration =9.8;  time =20;
>> final_velocity = initial_velocity ...
+ acceleration * time
final_velocity =
  196
```

1.8 The Format Command

By default, MATLAB displays numbers with four decimal place values. This is known as ***short format***. If you want more precision, you need to use the **format** command.

Command	Description
format short	**Four** decimal digits **(default).**
format long	**16** decimal digits.
format short e	**Five** digits plus exponent.
format long e	**16** digits plus exponents.
format bank	**Two** decimal digits
format rat	**Rational** approximation
format compact	Eliminates **empty** lines
Format loose	Adds **empty** lines

```
>> format short
>> pi
ans =
   3.1416

>> format long
>> pi
ans =
   3.141592653589793

>> format bank
>> pi
ans =
        3.14

>> format short e
>> pi^17
ans =
  2.8284e+008
```

```
>> format long e
>> pi^17
ans =
   2.828445635865329e+008

>> format rat
>> pi
ans =
    355/113
```

Example 1.1: prove the trigonometric equation below when $\boldsymbol{\theta = 2\pi/3}$

$$\boldsymbol{sin^2\theta = \frac{1 - cos2\theta}{2}}$$

Sol:

```
>> theta =2*pi/3;
>> Lhs= sin(theta)^2
Lhs =
   0.7500

>> Rhs=(1-cos(2*theta))/2
Rhs =
   0.7500
```

EXERCISE 2: with **angle = π/6** verify the equations:

1. $\boldsymbol{cos^2\frac{x}{2} = \frac{\tan x + \sin x}{2\tan x}}$

2. $\boldsymbol{tanh^{-1}x = \frac{1}{2}ln\left(\frac{1+x}{1-x}\right)}$

Chapter 2

Vectors And Matrices



2.1 Defining Vectors

A vector is a one-dimensional **array** of numbers. MATLAB allows creating two types of vectors:

- Row vectors
- Column vectors

Row vectors are created by enclosing the set of elements in **square brackets**, using space or comma (**,**) to delimit the elements. For example

```
>> v1=[1 2 3 4 5]          % using space between elements
v1 =
    1    2    3    4    5

>> v2=[2.1,3.8,4,10.9,7,1.3]      % using comma ( , ) between elements
v2 =
   2.1000    3.8000    4.0000   10.9000    7.0000    1.3000
```

Column vectors are created by enclosing the set of elements in **square brackets [ ]**, using semicolon **(;)** to delimit the elements. For example:

```
>> c=[2;3;4;7;9]
c =
    2
    3
    4
    7
    9
```

2.2 Creating Vectors

There are numerous ways to actually create a vector, each one having advantages in particular situations:

2.2.1 Colon Notation

MATLAB colon notation is a compact way to refer to ranges of matrix elements. It is often used in copy operations and in the creation of vectors and matrices. Colon notation can be used to create a vector as follows:

Variable = firstValue : step : lastValue

Or

Variable = firstValue : lastValue

Where ***firstValue*** and ***lastValue*** are the range of values covered by elements of the vector, and ***step*** is the (optional) increment. If ***step*** is **omitted** a value of ***1*** (unit increment) is used. The numbers ***firstValue***, ***step***, and ***lastValue*** need not be integers. For examples:



```
>> v =1:10    % creates vector with 10 elements range from 1 to 10 with step=1
v =
    1    2    3    4    5    6    7    8    9   10

>> u =1:2:10    % creates vector with 5 elements range from 1 to 10 with step=2
u =
    1    3    5    7    9

>> r =1.5:0.5:5    % creates vector with 8 elements range from 1.5 to 5 with step=0.5
r =
   1.5000   2.0000   2.5000   3.0000   3.5000   4.0000   4.5000   5.0000

>> r = [1.5:0.5:5]    % another way to create same vector r using [ ]
r =
   1.5000   2.0000   2.5000   3.0000   3.5000   4.0000   4.5000   5.0000

>> a=[-1:2:-12]          % generating error
a =
   Empty matrix: 1-by-0

>> a=[-1:-2:-12]
a =
   -1   -3   -5   -7   -9  -11
```

2.2.2 Built-in Function

To create a vector with one of these functions you must (at least initially) decide how long you want the vector to be. You must also decide whether the vector is a row or column vector. The ***linspace*** and ***logspace*** functions create vectors with linearly spaced or logarithmically spaced elements, respectively.

Variable=linspace (firstValue , lastValue , no. of elements)
Variable=logspace (firstValue , lastValue , no. of elements)

The third argument of both ***linspace*** and ***logspace*** is optional. The third argument is the number of elements to use between the range specified with the first and second arguments.

```
>> x=linspace(2,10,5)    % generates 5 elements range from 2 to 10 of x vector
x =
     2     4     6     8    10

>> y=logspace(1,4,4)      % generates 4 elements between 10^1 to 10^4
y =
        10       100      1000     10000
```



Note: when the no. of elements is omitted, ***linspace*** function generates ***100*** elements and ***logspace*** function generate ***50*** elements.

2.3 Vector Addressing

Individual elements of a vector can be addressed with an integer index enclosed in parentheses, for examples:

```
>> x=[2:2:20]
x =
    2    4    6    8   10   12   14   16   18   20

>> y = x(3)          % assigning 3rd element of vector x to variable y
y =
    6

>> z=x(9)/y^x(1)   % assigning 1st and 9th elements in expression to compute z
z =
   0.5000
```

Note :

- **Adding** element(s) to **increase** the **size** of a vector simply by assigning a value to an element(s) that has not been previously used. For examples:

```
>> r=linspace(5,20,4)       % creates vector r with 4 elements
r =
    5   10   15   20

>> x=r(6)          % generating error
??? Attempted to access r(6); index out of bounds because numel(r)=4.

>> r(6)=30      % increases the size of r (4 elements) by assigning value to its 6th
                elements through passing the 5th element by assigning 0 to it
r =
    5   10   15   20    0   30

>> r(7:9) = [50 60 70]          % adding elements (from 7th to 9th) to vector r
r =
    5   10   15   20    0   30   50   60   70
```

- **Deleting** element(s) from vector through **equality** empty square parentheses **[ ]** to its element(s). For example



```
>> r(5) =[ ]                % deleting the 5th element from vector r
r =
    5    10    15    20    30    50    60    70

>> r(5:8) =[ ]              % deleting elements ( 5th to 8th ) from vector r
r =
    5    10    15    20
```

- Once a vector has been created, it may be assigned to **another vector** or assigned subset of it and also create **transpose** vector. For examples:

```
>> x=[2,1,-3,5,8,-7,-11]          % creates vector x with 7 elements
x =
    2     1    -3     5     8    -7   -11

>> y = x                          % assigning vector x to vector y
y =
    2     1    -3     5     8    -7   -11

>> z = x(2:6)                     % assigning subset of vector x to z
z =
    1    -3     5     8    -7

>> w = x(4:end)          % assigning subset (from 4th to last element) of vector x to w
w =
    5     8    -7   -11

>> v = x'                % creates transpose vector v from vector x using single quote ( ' )
v =
     2
     1
    -3
     5
     8
    -7
   -11

>> v = x(3:6)'           % reassigns subset of vector x to transpose vector v
v =
    -3
     5
     8
    -7
```



2.4 Vector Operations

2.4.1 Addition and Subtraction

Addition and subtraction of vectors is performed with the + and – operators, such as (a+b, b-a , -x+a , …). The vectors must have the **same length**. For examples:

```
>> a = [2,5,9,-1,4];      % creates vector a
>> b = [5,-7,-1,0,8];     % creates vector b
>> c = a+b                % vector addition
c =
    7    -2     8    -1    12

>> c = -b-a               % vector subtraction
c =
   -7     2    -8     1   -12

>> c = 20+b-a             % adding 20 to each elements of b and then minus a
c =
    23     8    10    21    24
```

2.4.2 Multiplication

Which includes scalar and vectors multiplication, such as (2*a , a*b , b*(-a/4) , …). The vectors must have the **same length** and one of them should be **transpose vector**. For examples;

```
>> a = [9 4 -1 -3 6 1];       % creates vector a
>> b = [2 7 5 -5 -2 0];       % creates vector b
>> c=3*a                      % scalar multiplication
c =
    27    12    -3    -9    18     3
>> c = a*b'                   % vector multiplication
c =
    44

>> c = b*-a'
c =
   -44

>> a/2^2                      % this expression is same equal to  (a*1/2^2)
ans =
    2.2500    1.0000   -0.2500   -0.7500    1.5000    0.2500
```



2.4.3 Element-wise Operations

Different operators to indicate element-by-element of vectors which include **multiplication**, **right division, left division** and **power** operators, vectors must have the same length. For examples:

```
>> a=[3 5 1 9];
>> b=[2,1,4,6];
>> c = a .* b        % using element by element multiplication ( .* )
c =
    6    5    4   54

>> c = a ./ b        % using element by element right division ( ./ )
c =
   1.5000   5.0000   0.2500   1.5000

>> c = a .\ b
c =
   0.6667   0.2000   4.0000   0.6667

>> c = a .^ 2        % using element by element power ( .^ )
c =
    9   25    1   81

>> c = a .^ b
c =
        9         5         1    531441
```

Note: The elementary math functions also their results are element by element when they are used with vectors. For examples:

```
>> x=[3,6,5,7];
>> y=[1.35  -2.68  8.72];
>> sqrt(x)
ans =
   1.7321   2.4495   2.2361   2.6458

>> sin(y)
ans =
   0.9757  -0.4454   0.6479

>> exp(x)
ans =
  1.0e+003 *
   0.0201   0.4034   0.1484   1.0966
```



```
>> fix(ans)
ans =
        20        403        148       1096

>> factorial(x)
ans =
         6        720        120       5040
```

2.5 Vector Built-In Functions

There are some useful vector built-in functions supported by MATLAB, such as:

Function	Description
Sum(x)	**sums** the vector elements
Length(x)	**Find** the no. of a vector's elements
max(x)	return the *largest* element in vector
min(x)	return the *smallest* element in vector
sort(x)	**Sorts** the vector
prod(x)	**Products** the vector n(n-1)(n-2)…
dot(a,b)	Computes the **dot** products (*must be 3-element vectors*)
cross(a,b)	Computes the **cross** products (*must be 3-element vectors*)

```
>> x=[-5 1 7 10 -2 4]
x =
   -5     1     7    10    -2     4

>> length(x)
ans =
     6

>> sum(x)
ans =
    15
>> max(x)
ans =
    10

>> min(x)
ans =
    -5

>> sort(x)                    % sorts ascending the vector
ans =
    -5    -2     1     4     7    10
```



```
>> sort(x,'descend')          % sorts descending the vector

ans =
   10     7     4     1    -2    -5

>> prod(x)
ans =
       2800

>> a=[1 -2 3];
>> b=[-3 2 4];
>> scalar = dot(a,b)
scalar =
    5
>> Vector = cross(a,b)
Vector =
  -14   -13    -4
```

Note:

- The ***min*** and ***max*** functions they also used to determine the element's ***location*** within the vector, the *syntax* is: ***[a b] = min/max (v)***. Where "***a***" is the value and ***b*** is the elements location. For examples:

```
>> V=[2 8 0 1 9 3 0 4];        % vector V with two minimum values
>> [a b]=min(V)
a =
  0                            % the smallest value in vector V
b =
  3                            % the location of 1st smallest element (regardless other value)

>> R=[7 5 3 2 1 9 8 6 4 0];          %  Creates vector R
>> [largest  position]=max(R)
largest =
    9                          % the largest value in vector R
position =
    6                          % the location of the element
```

- These built-in functions can also use with **matrices** with little different in arguments and which will be mentioned later**.**

Example 2.1: Vectors $\boldsymbol{u = 2i + 5j - k}$ and $\boldsymbol{v = -4i + 3j - 2k}$, find:

1) The *angle* between them
2) The *perpendicular* vector on their plane



Sol:

[1] To find the *angle* between vector use the rule:

$$\boldsymbol{u}.\boldsymbol{v} = \cos\theta \, \|\boldsymbol{u}\|\|\boldsymbol{v}\| \quad \rightarrow \quad \theta = \cos^{-1}\frac{\boldsymbol{u}.\boldsymbol{v}}{\|\boldsymbol{u}\|\|\boldsymbol{v}\|}$$ Where

$\boldsymbol{u}.\boldsymbol{v}$ is the *dot product*

$\|\boldsymbol{u}\|$ is the *magnitude* of the vector which: $\|\boldsymbol{u}\| = \sqrt{\boldsymbol{i}^2 + \boldsymbol{j}^2 + \boldsymbol{k}^2}$

```
>> u = [2 5 -1];
>> v = [-4 3 -2];
>> uu = sqrt (sum(u.^2));          % computes the u vector magnitude
>> vv = sqrt (sum(v.^2));          % computes the v vector magnitude
>> theta = acosd (dot(u,v)/(uu*vv))  % finds the angle(in degree) using acosd function
theta =
   72.2341
```

[2] The *perpendicular* vector is found by using the ***cross*** function

```
>> p = cross (u,v)     % finds the perpendicular vector using cross function
p =
    -7     8    26
```

2.6 Creating Matrices (2D array)

Creating (***m x n***) matrix (where ***m*** is rows and ***n*** is columns) by assigning the elements of the matrix to a variable and its done by entering elements **row by row** inside **square brackets [ ]**. Each elements in row should separated with **space or comma (,)** and for each row is separated by **semicolon(;) or Enter key**.

Variable =[1st row elements ;2nd row elements ;3rd row elements ; …]

```
>> a = [2 3 5; 1 0 -2; 7 8 9]   % using semicolon (;) at the end of the row
a =
     2     3     5
     1     0    -2
     7     8     9

>> a=[1 2 3 4                   % pressing Enter key before entering new row
5 7 8 9
3 0 1 7]
a =
     1     2     3     4
     5     7     8     9
     3     0     1     7
```



```
>> b = [ 1:5 ; linspace(2,10,5) ; 5:5:25 ]  % creates matrix using vector creation methods
b =
     1     2     3     4     5
     2     4     6     8    10
     5    10    15    20    25

>>mat=[pi/2, log(10) , 3.2*4 ; sqrt(12) , sin(pi/6) , 1/exp(2)]   Creates matrix using (functions, expressions and constants)
mat =
   1.5708    2.3026   12.8000
   3.4641    0.5000    0.1353
```

Note: when entering elements of matrix, it must considered that each row has the same number of elements.

2.7 Matrix Addressing

The address of an element in a matrix is defined by the row number and the column number enclosed in parentheses, for examples:

```
>> x=[2 11 1 -6;0 7 14 3;9 19 -8 -1]  % creates matrix x (3 x 4)
x =
    2    11     1    -6
    0     7    14     3
    9    19    -8    -1

>> x(3,2)                              % displays the value of (3,2) element
ans =
   19

>> x(1,1)=5                            % assigns a new value to (1,1) element

x =
     5    11     1    -6
     0     7    14     3
     9    19    -8    -1

>> x(1,4)/x(2,2)                       % using elements in mathematical expression
ans =
   -0.8571
```

Adding new element(s) to increase the size of matrix by adding **entire** row(s) or column(s). For examples:

```
>> mat=[1:5 ; 2:2:10 ; linspace(-10,-2,5) ; 11 12 13 14 15]  % creates matrix (4x5)
```



```
mat =
 1   2   3   4   5
 2   4   6   8   10
-10  -8  -6  -4  -2
 11  12  13  14  15
```

```
>> mat(5,5)=100          % adding new element located at 5th row and 5th column
mat =
1    2    3    4    5
2    4    6    8    10
-10  -8   -6   -4   -2
11   12   13   14   15
0    0    0    0    100
```

Adding entire row which is not created previously (with zero value) to assign the new element

```
>> mat(5,1:4)=[50 60 70 90]  % assigns values to elements (1st to 4th column) of the 5th row
mat =
1    2    3    4    5
2    4    6    8    10
-10  -8   -6   -4   -2
11   12   13   14   15
50   60   70   90   100
```

```
>> mat( : , 6)=111       % adds the 6th column with the same value for all its elements
mat =
1    2    3    4    5    111
2    4    6    8    10   111
-10  -8   -6   -4   -2   111
11   12   13   14   15   111
50   60   70   90   100  111
```

```
>> mat (end , : ) = 200      % modifies all elements value of the last row (the 6th row)
mat =
1    2    3    4    5    111
2    4    6    8    10   111
-10  -8   -6   -4   -2   111
11   12   13   14   15   111
200  200  200  200  200  200
```

```
>> mat( 2:4 , end)=50      % modifies the elements of (2nd to 4th row) of the last column
mat =
1    2    3    4    5    111
2    4    6    8    10   50
-10  -8   -6   -4   -2   50
11   12   13   14   15   50
200  200  200  200  200  200
```

```
>> mat(2:4 , 4:end)=99      % modifies subset of matrix located at (2nd to 4th row)
                            and (4th to last column) with same value
```



```
mat =
1     2     3     4     5     111
2     4     6    99    99    99
-10   -8    -6    99    99    99
11    12    13    99    99    99
200   200   200   200   200   200
```

Deleting element(s) from matrix by deleting entire row(s) and entire column(s). For examples:

```
>> mat( : , 4)=[ ]                % deleting the 4th column
mat =
1     2     3     5     111
2     4     6     10    50
-10   -8    -6    -2    50
11    12    13    15    50
200   200   200   200   200

>> mat(5, : )=[ ]                 % deleting the 5th row
mat =
1     2     3     5     111
2     4     6     10    50
-10   -8    -6    -2    50
11    12    13    15    50

>> mat(2:4, : )=[ ]               % deleting the rows from 2nd to 4th
mat =
     1     2     3     5    111
```

- It also can assign the matrix entirely or partially to another matrix or vector. For examples:

```
>> a=[12 2 4 5;9 -2 -5 4;-3 1 7 9]      % creates matrix a(3 x 4)
a =
   12     2     4     5
    9    -2    -5     4
   -3     1     7     9

>> Row_V=a(2 , :)                % assigns the 2nd row of matrix a to row vector
Row_V =
    9    -2    -5     4

>> col_V=a( : , end)             % assign the last column of matrix a to column vector
col_V =
     5
     4
     9
```



```
>> b=a(2:3 , 3:4)                % assigns subset of matrix a to matrix b
b =
     -5     4
      7     9

>> c = b(:)                      % converts matrix b to column vector c
c =
   -5
     7
     4
     9

>> c = b(2:4)                    % converts matrix b (2nd to 4th elements) to row vector c
c =
     7     4     9
```

2.8 Matrix Operations

Operator	MATLAB Command	Description
+	a+b , 100+a , -b +10	Add Matrices (same size) and scalar
-	a-b . –a-10, b -30	Subtract Matrices (same size) and scalar
*	a*b , 2*a,-b*3	Multiply matrices {(m,**k**)*(**k**,n)} and scalar
'	a', b'	Find the transpose (a^T) of the matrix
.*	a.*b	Multiply element by element of matrices(same size)
./	a./b , a./5 , 6./b	Right division element by element of matrices(same size)
.\	a.\b , 6.\b ,1.\a	Left division element by element of matrices(same size)
.^	a.^b , b.^3 , 2.^a	Exponential element by element of matrices(same size)

```
>> a=[1 2;3 4];                  % create matrix a (2 x 2)
>> b=[3 5;7 6];                  % create matrix b (2 x 2)
>> c=a+b                         % adding matrices
c =
    4     7
   10    10

>> c= -b +a+10                   % scalar with matrices addition
c =
     8     7
     6     8

>> c= b-a                        % subtracting matrices
c =
     2     3
     4     2
```



```
>> c= a*b                % multiplying matrices
c =
    17    17
    37    39

>> c= 2*(b-a)            % scalar multiplication
c =
     4     6
     8     4

>> c'                    % transpose matrix c^T
ans =
     8     6
     7     8

>> c=b*b'-a*a'
c =
    29    40
    40    60

>> b.*a                  % element by element multiplication
ans =
     3    10
    21    24

>> a./b                  % element by element right division
ans =
    0.3333    0.4000
    0.4286    0.6667

>> a.\ 2                 % element by element left division
ans =
    2.0000    1.0000
    0.6667    0.5000

>> 2.^b                  % element by element exponential
ans =
     8    32
   128    64
```



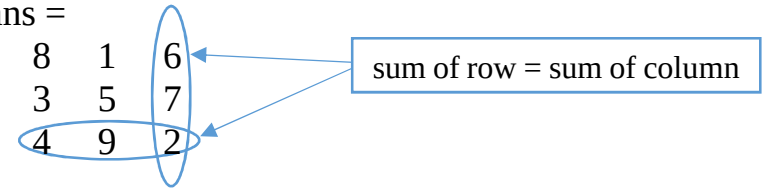
2.9 Special matrices

MATLAB provides functions that generating special matrices, such as

Function	Description
eye(n) *or* eye(n, m)	returns the ***(n x n)*** or ***(m x n)*** identity matrix
ones(n) *or* ones(n, m)	returns the ***(n x n)*** or ***(m x n)*** matrix of **1s**
zeros(n) *or* zeros(n, m)	returns the ***(n x n)*** or ***(m x n)*** matrix of **0s**
rand(n) *or* rand(n, m)	Generates ***(n x n)*** or ***(m x n)*** matrix of **real** random number between **0** to **1**
randi (imax, n) *or* randi (imax, n, m)	Generates ***(n x n)*** or ***(m x n)*** matrix of **integer** random number between **1** to **imax**
magic(n)	returns the ***(n x n)*** matrix constructed from the integers **1** through $\mathbf{n^2}$ with equal row and column sums, where ($\mathbf{n \geq 3}$)

```
>> Id = eye(4)          % generates identity matrix (4 x 4)
Id =
     1     0     0     0
     0     1     0     0
     0     0     1     0
     0     0     0     1

>> Id = eye(3,4)        % generates identity matrix (3 x 4)
Id =
     1     0     0     0
     0     1     0     0
     0     0     1     0

>> z = zeros(2)         % generates matrix (2 x 2) with value of 0s
z =
     0     0
     0     0

>> z = zeros(3,1)       % generates matrix (3 x 1) with value of 0s
z =
     0
     0
     0
>> ones(3)              % generates matrix (3 x 3) with value of 1s
ans =
     1     1     1
     1     1     1
     1     1     1

>> rand(2,4)            % generates matrix (2 x 4) with random values range (0 →1)
ans =
    0.1363    0.4952    0.4950    0.0550
    0.6787    0.1897    0.1476    0.8507
```

```
>> rand(3)          % generates matrix (3 x 3) with random values range (0 → 1)
ans =
    0.5606    0.5828    0.9889
    0.9296    0.8154    0.0005
    0.6967    0.8790    0.8654

>> randi(10,3)      % generates matrix (3 x 3) with random integer values range (1→10)
ans =
     7     5     5
    10     9    10
     6     3     6

>> randi(10,5,3)    % generates matrix (5 x 3) with random integer values range (1→10)
ans =
     9     1    10
     8     7     8
     6     7     6
     3     8    10
     7     9     6

>> magic(3)         % generates matrix (3 x 3) with integer values range (1→ 3^2)
ans =
     8     1     6
     3     5     7
     4     9     2
```



Notes:

- Generate **real** random values range in an interval {a, b} using ***rand*** functions, is obtained by: ***(b – a)*rand +a***. For examples:

```
>> r =(2+3)*rand(3,6)-3   % generates a matrix (3x6) with random values range (-3→2)
r =
    0.4078   -2.1250    0.5135   -2.8172   -1.3708   -1.7231
   -0.6837   -2.1822   -2.2320    1.0460   -0.2678   -2.8973
   -1.9392    0.3299    1.7673    0.7431   -1.0056    1.6184
```

- Generate **integer** random values range in an interval {a,b} using ***randi*** functions, is obtained by: ***randi([a,b],m,n)***. For examples:

```
>> A = randi([5,8],3)     % generates A(3x3) with integer numbers between 5 and 8
A =
     8     7     8
     7     5     7
     6     6     7
```



```
>> B = randi([-2,4],5,4)        % generates B(5x4) with integer numbers between -2 and 4
B =
     1     0    -1    -2
     3    -2     4     3
     3     1     2     3
    -2     0     1     1
    -1    -1     4    -1
```

2.10 Matrix built-in functions

The common built-in functions that provide by MATLAB which are used with matrix, such as:

Function	Description
size(A)	Returns the size of matrix **A (m x n)** as row vector
sum(A)	Returns the **sum of each column** of matrix **A (m x n)** as row vector
max(A)	Returns the **largest of each column** of matrix **A (m x n)** as row vector
min(A)	Returns the **smallest of each column** of matrix **A (m x n)** as row vector
sort(A)	Returns the sort of **each column** of matrix **A (m x n)**
Reshape(A,m,n)	Rearranges the matrix **A(r x c)** to new matrix **A(m x n)** , where ***r*c = m*n***
inv(A)	Computes the inverse (A^{-1}) of square matrix **A(n x n)**
det(A)	Returns the determinant of square matrix **A(n x n)**
diag(A)	Returns a vector of the diagonal elements of matrix **A(m x n)**

```
>> r =[1 0 -2 4;3 8 9 -1;7 6 -5 2]        % creates matrix r(3 x 4)
r =
     1     0    -2     4
     3     8     9    -1
     7     6    -5     2

>> mat = size(r)        % finds the size of matrix r and assigned it in row vector mat
mat =
     3     4

>> mat = sum(r)        % computes the sum of each column of matrix r
mat =
    11    14     2     5

>> total_sum= sum (sum(r))        % computes the sum of all elements of matrix r
total_sum =
    32
```



```
>> mx = max(r)          % a vector of the largest value of each column of matrix r
mx =
   7     8     9     4

>> maximum = max(max(r))          % finds the largest value of matrix r
maximum =
   9

>> mn = min(r)          % a vector of the smallest value of each column of matrix r
mn =
     1     0    -5    -1

>> minimum = min(min(r))          % finds the smallest value of matrix r
minimum =
   -5

>> sort(r)                    % sorts ascending each column of matrix r
ans =
     1     0    -5    -1
     3     6    -2     2
     7     8     9     4

>> sort(r, 'descend')          % sorts descending each column of matrix r
ans =
     7     8     9     4
     3     6    -2     2
     1     0    -5    -1

>> sort(r, 2)                  % sorts ascending each row of matrix r

ans =
    -2     0     1     4
    -1     3     8     9
    -5     2     6     7

>> sort(r, 2,'descend')        % sorts descending each row of matrix r
ans =
     4     1     0    -2
     9     8     3    -1
     7     6     2    -5


>> x= [5 8 7;3 9 10;2 11 4;6 1 0]          % creates matrix x(4 x 3)
```



```
x =
     5     8     7
     3     9    10
     2    11     4
     6     1     0

>> reshape (x, 2, 6)          % Rearranges the matrix x(4 x 3) to x(6 x 2)
ans =
     5     2     8    11     7     4
     3     6     9     1    10     0

>> a= magic(3)                % generates matrix a(3 x 3)
a =
     8     1     6
     3     5     7
     4     9     2

>> inv(a) , a^(-1)            % computes the matrix inverse of  a(3 x 3) in two ways
ans =
    0.1472   -0.1444    0.0639
   -0.0611    0.0222    0.1056
   -0.0194    0.1889   -0.1028

ans =
    0.1472   -0.1444    0.0639
   -0.0611    0.0222    0.1056
   -0.0194    0.1889   -0.1028

>> diag(a)                    % finds the diagonal elements of matrix a
ans =
     8
     5
     2
>> det(a)                     % computes the determinant of matrix a(3 x 3)
ans =
  -360
```

***Notes*:**

- The ***min*** function can also use to find the **location** of the smallest element in each column using the *syntax: **[a b]=min(M)***.

Where "***a***" is a **vector** of the *smallest* elements in each column and "***b***" is a **vector** of the smallest element's *location* in each column in matrix **M.** Also the same for ***max*** function.



```
>> M=magic (5)          % generates M(5x5) using magic function
M =
    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

>> [A B]=min(M)
A =
    4    5    1    2    3          % vector A represents the smallest elements in each column
B =
    3    2    1    5    4          % vector B represents the element's location in each column

>> [A B]=max(M)
A =
   23   24   25   21   22          % vector A represents the largest elements in each column
B =
    2    1    5    4    3          % vector B represents the element's location in each column
```

- If the *argument* of ***diag*** function is a **vector**, its output is a matrix (***n x n***) which ***n*** is the length of the vector, the matrix's *main diagonal* is the *vector* elements and the other elements is *zeros* . For example:

```
>> v=[3 5 7 9]
v =
    3    5    7    9
>> h= diag (v)
h =
     3     0     0     0
     0     5     0     0
     0     0     7     0
     0     0     0     9
```



Example 2.2:

For the system of the equations

$$\begin{cases} 2x_1 + 3x_2 + x_3 = 9 \\ x_1 + 2x_2 + 3x_3 = 6 \\ 3x_1 + x_2 + 2x_3 = 8 \end{cases}$$

compute the unknowns $x_1, x_2,$ and x_3 using the inverse matrix method.

Solution:

In matrix form, the given set of equations is $AX = B$ where

$$A = \begin{bmatrix} 2 & 3 & 1 \\ 1 & 2 & 3 \\ 3 & 1 & 2 \end{bmatrix}, \quad X = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}, \quad B = \begin{bmatrix} 9 \\ 6 \\ 8 \end{bmatrix}$$

Then,

$$X = A^{-1}B$$

or

$$\begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 2 & 3 & 1 \\ 1 & 2 & 3 \\ 3 & 1 & 2 \end{bmatrix}^{-1} \begin{bmatrix} 9 \\ 6 \\ 8 \end{bmatrix}$$

```
>> A = [2 3 1; 1 2 3; 3 1 2];   % generates the matrix of x's coefficients
>> B = [9 6 8]';                % generates the column vectors of constants using (')
>> X = A\B                      % calculate the values of x
X =
    1.9444
    1.6111
    0.2778

>> X = (A^-1)*B                 % same result with other solution
X =
    1.9444
    1.6111
    0.2778

>> X = inv (A)*B                % same result using inv function
X =
    1.9444
    1.6111
```

```
0.2778
```

Exercises

Q1) Generate a random:

- Real number in the range from **0** to **1**
- Real number in the range from **0** to **20**
- Real number in the range from **20** to **50**
- Integer in the range from **0** to **10**
- Integer in the range from **2** to **11**
- Integer in the range from **50** to **100**

Q2) Using the **colon operator**, create the following vectors

```
3 4 5 6
1.0000 1.5000 2.0000 2.5000 3.0000
5 4 3 2
```

Q3) Using the **linspace** function, create the following vectors:

```
4       6      8
-3   -6   -9   -12   -15
9      7      5
```

Q4) Using the colon operator and the transpose operator, create a column vector that has the values **–1** to **1** in steps of **0.2**.

Q5) Let **x = [4 5 9 6].**

- *Subtract* **3** from each element.
- *Add* **11** to the *odd-index* elements.
- *Compute* the *square root* of each element.
- *Raise* to the power **3** each element.

Q6) Create a vector **x** with the elements:

- `2, 4, 6, 8, …`
- `9, 7, 5, 3, 1, -1, -3, -5`
- `1, 1/2, 1/3, 1/4, 1/5,…`
- `0, 1/2, 2/3, 3/4, 4/5,…`

Q7) Create a vector **x** with the elements: $x_n = \frac{(-1)^n}{2n-1}$ for $n = 1, 2, 3, \ldots$ Find the sum of the ***100***-element vector.

Q8) Given a vector **t**, write down the MATLAB expressions that will compute:

- $\ln(2 + t + t^2)$
- $\cos(t)^2 - \sin(t)^2$
- $e^t(1 + \cos(3t))$
- $\tan^{-1}(t)$



Test them for ***t = 1 : 0:2 : 2***.

Q9) The angle between two column vectors, which is defined as:

$$\cos\theta = \frac{x^T y}{\|x\|\|y\|}$$

Compute an angle between the vectors:

- `x = [3 2 1]` and `y = [1 2 3]`
- `x = 1 : 5` and `y = 6 : 10`

Q10) Using colon operators for the rows, create the matrix:

```
7 6 5
3 5 7
```

Q11) Generate a **2 × 3** matrix of random

- Real numbers, each in the range from **0** to **1**
- Real numbers, each in the range from **1** to **10**
- Integers, each in the range from **5** to **20**

Q12) Define the matrices**: T = [3 4; 1 8; -4 3]; A = [diag(-1:2:3) T; -4 4 1 2 1].**
Perform the following operations on the matrix **A**:

- *extract* a vector consisting of the **2nd** and **4th** elements of the **3rd** row
- *find* the minimum of the **3rd** column
- *find* the maximum of the **2nd** row
- *compute* the sum of the **2nd** column
- *extract* the submatrix consisting of the **1st** and **3rd** rows and all columns
- *extract* the submatrix consisting of the **1st** and **2nd** rows and the **3rd** , **4th** and **5th** columns
- *compute* the total sum of the **1st** and **2nd** rows
- *add* **3** to all elements of the **2nd** and **3rd** columns

Q13) Let **A = [2 4 1; 6 7 2; 3 5 9]**. Provide the commands which:

- *assign* the 1st row of A to a vector **x**
- *assign* the last **2** rows of **A** to a vector **y**
- *append* the columns of **A**
- *append* the rows of **A**

Q14) Create a variable, *rows*, which is a random integer in the range from **1** to **5**. Create a variable, *cols*, which is a random integer in the range from **1** to **5**. Create a matrix of all zeros with the dimensions given by the values of *rows* and *cols*.

Q15) Create a matrix variable, *mymat*, which stores the following:

mymat =

```
2 5 8
```



```
7 5 3
```

Using this matrix, find a simple expression that will transform the matrix into each of the following:

```
2 7
5 5
8 3
```

```
2 5
7 8
5 3
```

```
8 5 2
3 5 7
```

```
8 3
5 5
2 7
```

```
2 5 8 2 5 8
7 5 3 7 5 3
```

Q16) Create a **4 × 2** matrix of all zeros and store it in a variable. Then, replace the second row in the matrix with a **3** and a **6**.

Q17) Create a vector, x, which consists of **20** equally spaced points in the range from – to + . Create a y vector that is **sin(x)**.

Q18) Create a **4 × 6** matrix of random integers, each in the range from **–5** to **5**; store it in a variable. Create another matrix that stores for each element the absolute value of the corresponding element in the original matrix.

Q19) Given the vectors **x = [1 3 7], y = [2 4 2]** and the matrices **A = [3 1 6; 5 2 7], B = [1 4; 7 8; 2 2],** *determine* which of the following statements will correctly execute (and if not, try to understand why) and provide the result:

- `x + y`
- `x + A`
- `x' + y`
- `A + [x' y']`
- `[x ; y] + A`
- `[x ; y']`
- `[x ; y]`
- `A - 3`
- `A + B`
- `B' + A`
- `B * A`
- `A.* B`
- `A'.* B`
- `2 * B`
- `2.* B`
- `B ./ x'`
- `B ./ [x' x']`
- `2/A`
- `ones (1; 3)*A`
- `ones (1; 3)*B`

Q20) Let **A** be a random **(5x5)** matrix, and **b** a random **(5x1)** vector. Given that: ***Ax* = *b***, try to find ***x*** . Explain what is the *difference* between the operators / , \ and the *function* **inv**. Having found ***x***, check whether ***Ax* - *b*** is close to a *zero vector*.

Q21) Using ***ones*** and ***zeros*** functions to generate matrix forms:

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

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

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

Chapter 3

Programming In MATLAB

3.1 Script M-Files

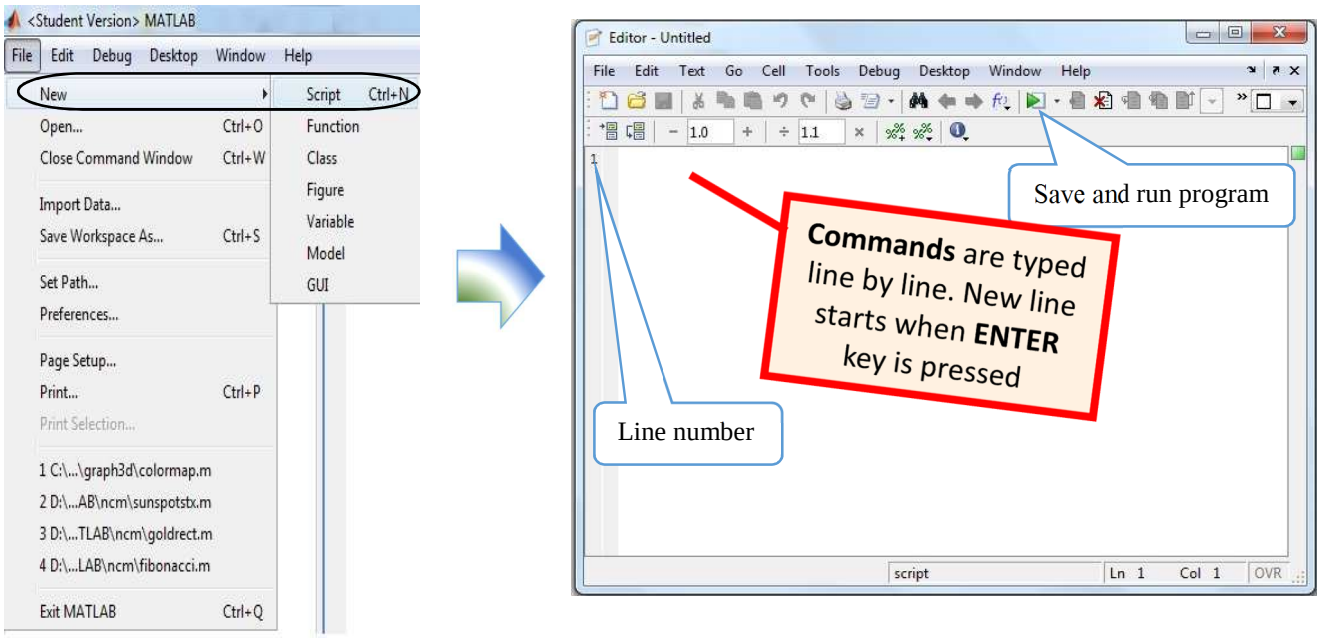
For simple problems, entering commands at the MATLAB prompt in the Command window is simple and efficient. However, when the number of commands increases, or you want to change the value of one or more variables, reevaluate a number of commands, typing at the MATLAB becomes tedious. You will find that for most uses of MATLAB, you will want to prepare a ***script***, which is a sequence of commands written to a file. Then, by simply typing the script file name at a MATLAB prompt, each command in the script file is executed as if it were entered at the prompt.

Script File*:** Group of MATLAB commands placed in a text file with a text editor. MATLAB can open and execute the commands exactly as if they were entered at the MATLAB prompt. The term "*script*" indicates that MATLAB reads from the "script" found in the file. Also called "*M-files*," as the filenames must end with the extension ***'.m', e.g. *example1.m*.

M-files are text files and may be created and modified with any text editor. The steps to create a script are:

1) Click on icon on the MATLAB toolbar.
2) Press keys (***Ctrl* + *N***)
3) Form (***File*** → ***New*** → ***Script***)

- A new window will activate called ***the Editor*** as shown.



- When finished, save the file using ***File*** → ***Save*** or click on icon. The rules for filenames are the same as for variables (they must start with a letter, after that there can be letters, digits, or the underscore, etc.). By default, scripts will be saved in the ***Work Directory***. If you want to save the file in a different directory, the *Current Directory* can be changed.



Example 3.1: Write a program (m-file) and named it "Qroots.m" to find the quadratic equation roots:

$$2x^2 - 5x + 3 = 0$$

Sol:

```
a=2;
b=-5;
c=3;
r1=(-b+sqrt(b^2-4*a*c))/(2*a)
r2=(-b-sqrt(b^2-4*a*c))/(2*a)
```

To execute the script M-file, simply type the name of the script file **Qroots** at the MATLAB prompt.

```
>> Qroots
r1 =
   1.5000
r2 =
    1
```

Notes :

- When file is executed, All its variables are displayed in workspace window
- It is useful to use functions such as (clc , clear , format ,…) in script file to improve the results.

Example 3.2: write a program (vector.m) to generate a vector with 12 random elements and find:

a. The largest element and its position.
b. The smallest element and its position.

Sol:

```
clear                    % clear variable from memory
clc                      % clear the commands windows
format bank              % real number with 2 digits
V=rand(1,12)             % generate row vector
[Vmax,Pmax]=max(V)       % find the maximum element and its position
[Vmin Pmin]=min(V)       % find the minimum element and its position
```

```
>> vector
V =
  Columns 1 through 6
      0.75      0.26      0.51      0.70      0.89      0.96
  Columns 7 through 12
```



```
    0.55      0.14      0.15      0.26      0.84      0.25
Vmax =
    0.96
Pmax =
    6.00
Vmin =
    0.14
Pmin =
    8.00
```

3.2 Input and Output Statements

The script would be much more useful if it were more general; for example, if the value of the *radius* could be read from an external source rather than being assigned in the script. Also, it would be better to have the script print the output in a nice, informative way. Statements that accomplish these tasks are called ***input/output*** statements, or ***I/O*** for short. With examples of input and output statements will be shown here from the Command Window, these statements will make the most sense in scripts.

3.2.1 input Function

The simplest input function in MATLAB is called **input**. The **input** function is used in an assignment statement. To call it, a string is passed, which is the prompt that will appear on the screen, and whatever the user types will be stored in the variable named on the left of the assignment statement. To make it easier to read the prompt, put a ***colon*** (:) and then a ***space*** after the prompt. For example:

```
>> r = input ('Enter the radius: ')
Enter the radius: 7
r =
    7
```

If character or string input is desired, ***'s'*** must be added after the prompt:

```
>> name = input ('Enter your Name: ', 's' )
Enter your Name: Ahmed
name =
Ahmed
```

MATLAB gave an ***error message*** and repeated the prompt. However, if the *input function* is used to enter number *but* the user instead enters a letter or vice versa

```
>> n = input ('Enter your Age: ')
Enter your Age: k
??? Error using ==> input
Undefined function or variable 'k'.
```



```
Enter your Age: 21
n =
    21
```

Separate **input** statements are necessary if more than one input is desired. For example

```
>> T = input('Enter the temperature: ');
Enter the temperature: 37
>> s = input('Is it "C" or "F" ?','s');
Is it "C" or "F" ?C
```

3.2.2 Output Statements (disp and fprintf) functions

The simplest output function in MATLAB is **disp**, which is used to display the result of an expression or a string *without* assigning any value to the default variable *ans*. However, **disp** does *not* allow *formatting*. For examples:

```
>> disp ('Hello')                    % displays string
Hello

>> disp (6^4)                        % displays numeric expression
      1296

>> disp ([2:8])                      % displays vector
   2    3    4    5    6    7    8

>> disp ([1:5 ; 5:5:25])             % displays matrix
   1    2    3    4    5
   5   10   15   20   25

>> disp('   Col.{1}   Col.{2}   Col.{3}') , disp(rand(5,3))    % displays as table

    Col.{1}    Col.{2}    Col.{3}
    0.3181     0.6393     0.5225
    0.1192     0.5447     0.9937
    0.9398     0.6473     0.2187
    0.6456     0.5439     0.1058
    0.4795     0.7210     0.1097
```

Formatted output can be printed to the screen using the **fprintf** function. For example:

```
>> fprintf ('The 7! value is %d\n' , factorial(7))
The 7! value is 5040
```



The ***fprintf*** function, first a string (called the ***format string***) is passed, which contains any text to be printed as well as formatting information for the expressions to be printed. In this example, the **%d** is an example of format information. The **%d** is sometimes called a ***placeholder***; it specifies where the value of the expression that is after the string is to be *printed*. The character in the placeholder is called the ***conversion character,*** and it specifies the *type of value* that is being printed. There are others of the simple placeholders:

Placeholder (character)	Description	Example
%d	Format as a *integer*	**>> fprintf ('%d', 4^5)** **1024>>**
%f	Format as a *floating point* value	**>> fprintf ('%f', sqrt(90.25))** **9.500000>>**
%g	Format as the *most compact* form (*no trailing zero*)	**>> fprintf ('%g', sqrt(90.25))** **9.5>>**
%e	Format as a floating point value in *scientific* notation	**>> fprintf ('%e', pi)** **3.141593e+000>>**
%s	Format as a *string*	**>> fprintf ('%s', '3*8/2')** **3*8/2>>**
\n	Insert a *new line* in the output string	**>> fprintf ('Welcome! \n this is MATLAB')** **Welcome!** **this is MATLAB>>**
\t	Insert a *tab* in the output string	**>> fprintf ('Welcome! \t this is MATLAB')** **Welcome! this is MATLAB>>**

Notes :

- It's important adding the character \n at the *end* of output string in order to *avoid* the prompt (>>) from *stick* to the result as shown before.

- The character \n can also use in *input* function, for example:

```
>> h = input ('Enter \n The shape height :')
Enter
 The shape height :
```

- The character \n in form '\n\n' use to get blank line in output, for example:

```
>> fprintf ('Hello \n\n This is MATLAB \n')
Hello

This is MATLAB
```

- A ***field width*** can also be included in the placeholder in **fprintf**, which specifies how many characters total are to be used in printing. For example, *%5d* would indicate a field width of 5 for printing an integer and %10s would



indicate a field width of 10 for a string. For floats, the number of decimal places can also be specified; for example, *%6.2f* means a field width of 6 (including the decimal point and the decimal places) with two decimal places. For floats, just the number of decimal places can also be specified; for example, *%.3f* indicates three decimal places

```
>> fprintf ('The integer is %3d and the float is %6.2f \n', 56 , 42.95897)
The integer is  56 and the float is  42.96
```

*Note that if the field width is wider than necessary, **leading** blanks are printed, and if more decimal places are specified than necessary, **trailing** zeros are printed.*

- For a **vector**, if a conversion character (%d, %f …) and the **\n** character are in the format string, it will print in a *column* regardless of whether the vector itself is a row vector or a column vector. For examples:

```
>> v=1:5;
>>fprintf ('%d', v) , fprintf ('\n')
12345

>> fprintf ( '%d\n', v)
1
2
3
4
5
```

- For **matrices**, Specifying one conversion character and then the \n character will print the elements from the matrix in one column. The first values printed are from the first column, then the second column, and so on. For examples:

```
>> mat=[1 2 3;4 0 6;7 9 8]          % create (3 x 3) matrix
mat =
     1     2     3
     4     0     6
     7     9     8

>>fprintf ('%d\n', mat)
1
4
7
2
0
9
3
6
```



```
8
```

To *reshape* the matrix to (**3 x 3**), three of **%d** characters are specified, the **fprintf** will print *three* numbers across on each *line* of output and so on.

```
>> fprintf ('%d %d %d \n', mat)
1 4 7
2 0 9
3 6 8
```

Example 3.3: write a program to find the cosine of angles *(0,15,45,60,75,90)* and format the output result as : "**The cosine of angle X is : Y** ", named file "**AngleCosine**"

Sol

```
clc
clear
theta=0:15:90;                  % generate angles (0-90) steps 15
cosine=cosd(theta);             % find the cosine of angle
result= [theta ; cosine];       % merge the vectors to produce a matrix
fprintf('The cosine of angle %d is :%4.3f \n',result)
```

```
>> AngleCosine
The cosine of angle 0 is :1.000
The cosine of angle 15 is :0.966
The cosine of angle 30 is :0.866
The cosine of angle 45 is :0.707
The cosine of angle 60 is :0.500
The cosine of angle 75 is :0.259
The cosine of angle 90 is :-0.000
```

Example 3.4: How many years **(Y)** needs to be a *millionaire* **(Fv)** when you investment an amount of **(N)$** with annual interest rate **(R)%**, write a program *based on* equation: (using I/O statements and named file "investment")

$$Fv = N(1+R)^{Y}$$

Sol :

1) Redefine the equation in terms of **Y**

$$Y = \frac{ln\frac{Fv}{N}}{ln(1+R)}$$

2) Write & run a program

```
clear
clc
N=input('The amount of Investment :');
R=input('The interest rate{%} :');
Fv=1000000;
Y=log(Fv/N)/log(1+R);
fprintf('\n The No. of years are :\t %0.2f \n',Y)
```



```
fprintf('\n Which is approximately :\t %d years \n',ceil(Y))
```

```
>> investment
The amount of Investment :80000
The interest rate{%} :0.12

 The No. of years are :      22.29

 Which is approximately :     23 years
```

3.3 Relational and logical Operators

In MATLAB the result of a **logical** operation is **1** if it is ***true*** and **0** if it is ***false***. The relational operators (<, <=, >, >=, == and ~=) can be used to compare two matrices of the *same size* or a vector with scalar. For examples:

Relational Operator	Description
<	Less than
>	Greater than
<=	Less than or equal to
>=	Greater than or equal to
= =	Equal to
~=	Not equal to

```
>> 3>8                 % checks if 3 is greater than 8
ans =
    0

>> x= 4<=7            % checks if 4 less or equal 7 and assigns answer to x
x =
    1

>> y=(5>4)+(2<7)+(4*3==36/3)  % sums the result of checking each parenthesis
y =
    3

>> A=[1 0 8 2 -5]; B=[9 0 -4 1 2];   % create vectors A and B with same length
>> C=A>=B                  % check if A elements is greater or equal to B elements
C =
    0    1    1    1    0

>> D=B~=C                  % checks which of B elements is not equal to C elements
D
   1    1    1    0    1
```



```
>> E=B-A>0                % subtracts A from B and checks which element is greater than zero
E =
    1    0    0    0    1

 >> M = [9 5 1 0;3 6 -5 7;8 10 -6 -4]        % creates (3x4) matrix
M =
     9     5     1     0
     3     6    -5     7
     8    10    -6    -4


>> H = M<5        % checks M elements if less than 5 and assigns results to matrix H
H =
     0     0     1     1
     1     0     1     0
     0     0     1     1
```

Notes **:**

- The vector with values of ***0s*** and ***1s*** which is the result of relational operation are called ***logical vector***. The real power of logical vectors is that they can be used as *on-off* subscripts in vector expressions. Suppose that the problem is not just to find out how many negative numbers are in the vector, but to extract them for future use:

```
 >> v=[-2 3 4 -1 -6 2 8 9 -7 0 -5]          % create a vector v
v =
   -2    3    4   -1   -6    2    8    9   -7    0   -5

 >> Vn = v<0                % checks which v elements less than zero and assign to logical
Vn =                        vector Vn with 1s at positions where v elements are negative
    1    0    0    1    1    0    0    0    1    0    1

>> Vneg= v(Vn)     % extracts the negative elements of v using Vn vector and assign to Vneg
Vneg =
   -2   -1   -6   -7   -5

>> Vneg=v(v<0)      % the same result is done in one step
Vneg =
   -2   -1   -6   -7   -5
```

- **The order of precedence** in mathematical expression, the arithmetic operators (+, -, *, /, \, ^) is higher than relational operators, whereas the relational operators have the equal precedence and are evaluated from left to right, parenthesis used to overcome the precedence. For examples:



```
>> 6+1 <= 18/3    % the / and + executes first and then relational operator <=
ans =
    0

>> 6+ (1<=18) /3   % the parenthesis executes first
ans =
   6.3333
```

- The relational operators can also uses with characters based on **ASCII code**. For examples:

```
>> 'n' > 'z'
ans =
    0

>> 'n' > 'Z'    % the ASCII code of 'n' is greater than 'Z'
ans =
    1

>> 'Not' >= 'not'    % check each character in both strings(must have same length)
ans =
    0    1    1

>> 'MatLab'  <  'matlab'
ans =
    1    0    0    1    0    0
```

The **logical** operators (**&**, **|** and **~**) allow for the logical combination or negation of relational operators. For examples:

Logical operator	Name	Description
&	**AND**	**Compares** between two operands(A,B) if *both* are **true**, the result is **true**(1) ;otherwise its **false**(0)
|	**OR**	**Compares** between two operands(A,B) if *either one* or *both* are **true**, the result is **true**(1) ;otherwise its **false**(0)
~	**NOT**	**Checks** one operand (A), if it is **true** (1), the result is **false**(0), or vice versa.

```
>> 8 & -1      % 8 AND -1 (both operands are nonzero)
ans =
    1

>> 9 | 0       % 9 OR 0
ans =
```



```
     1
 >> ~12          % NOT 12 (which is returning zero)
ans =
    0

 >> R=10*((1&-2) - (0|3) + (~0))      % using logical operators in math expression
R =
   10
 >> X=[9 5 -4 0 6] ; Y=[1 0 7 -3 11];  % creates vectors X and Y (must have same length)
 >> Z=X&Y                             % compares X elements with Y elements using logical AND
Z =
    1    0    1    0    1

 >> Z= 0 | Y                          % compares scalar 0 with Y elements using logical OR
Z =
    1    0    1    1    1

 >> W= ~(X .* Y)              % logical NOT checks the results of multiplying element by
W =                           element of X with Y vectors
    0    1    0    1    0
```

***Notes*:**

- **The order of precedence** for logical operator **NOT** is *higher* than both arithmetic and relational operations, whereas the logical **AND** and **OR** are *equal* and *lower* than both arithmetic and relational operations, if two or more operators have the same precedence is executed from left to right. For examples:

```
 >>  x = -1 ; y = 4;     % defines variables x and y
 >> -2 < x < 0           % executes (-2<x) first, then (result < 0)
ans =
    0

 >> -2<x & x<0  % executes < operators first, then compares their results with AND operator
ans =
    1

 >> ~ (y>=1)            % executes parenthesis first, then logical NOT
ans =
    0

 >> ~y<=1               % executes logical NOT first, then chicks result with <=
ans =
    1
```



```
>> ~ ((x<-2) | (y>=4))      % executes parentheses first, then compares results with OR and finally uses logical NOT
ans =
    0

>> ~ (x<-2) | (y>=4)        % executes parentheses first, then uses logical NOT and finally compares results with OR
ans =
    1
```

- MATLAB provides additional built-in logical functions:

Function	Description	Example
all(A)	Determine whether ***all*** array elements are **nonzero** or **true**	**>> X=[3 4 1] ; Y=[8 0 7 5];** **>> all(X) , all(Y)** **ans =** **1** **ans =** **0**
any(A)	Determine whether ***any*** array elements are **nonzero**	**>> X=[0 0 0] ; Y=[6 0 5];** **>> any(X) , any(Y)** **ans =** **0** **ans =** **1**
Find(A)	Find ***indices*** and values of **nonzero** elements	**>> X=[0 9 1 0 3 0 5 7];** **>> find(X)** **ans =** **2 3 5 7 8**
Find(A>K)	Find ***indices*** of elements that returns ***true*** of relational operators	**>> X=[0 9 1 0 3 0 5 7];** **>> find(X>5)** **ans =** **2 8**

Example 3.5: Write a program to test the degrees of **40** students and find: (named file "students")

1) The number of students those degrees *above* **80**
2) The number of students those degrees *between* **60** and **79**
3) The number of students those ***failed*** *and* ***their degrees***

Sol:

```
clear
clc
Degree= randi([25,100],1,40) % generate the degrees of 40 students their range (25-100)
DegreeAbove80=sum(Degree>80); % find no. of students which degree>80 using sum function
Degree60to79=sum(Degree>=60 & Degree<80); % find no. of students which 60<degree<80
Failures=sum(Degree<50); % find no. of failures students
fprintf('\n')              % format output results
disp('-----------------------------------------------')
fprintf('The no. of students with degree above 80 :
%d\n',DegreeAbove80)
```



```
fprintf('The no. of students with degrees 60 to 79 :
%d\n',Degree60to79)
fprintf('The no. of failures students: %d\n',Failures)
fprintf('And their degrees are :\n')
disp(Degree(Degree<50))          % display the degrees which are less than 50
```

```
>> students
Degree =
  Columns 1 through 15
    61    85    42    62    93    68    89    81    69    43    75    31    72    75    80
  Columns 16 through 30
    92    99    83    69    95    69    26    34    90    61    89    40    66    72    27
  Columns 31 through 40
    71    52    28    62    39    34    40    36    39    28

-----------------------------------------------
The no. of students with degree above 80 : 10
The no. of students with degrees 60 to 79 : 14
The no. of failures students: 14
And their degrees are :
    42    43    31    26    34    40    27    28    39    34    40    36    39    28
```

3.4 Flow Control

Selection statements that test the results of relational or logical functions or operators are the decision-making structures that allow the flow of command execution to be controlled.

MATLAB has two basic statements that allow *choices*: the **if** statement and the **switch** statement. The **if** statement has optional **else** and **elseif** clauses for branching. The **if** statement uses expressions that are logically true or false.

There are two different *loop* statements in MATLAB: the **for** statement and the **while** statement. In practice, the **for** statement usually is used as the *counted loop*, and the **while** is used as the *conditional loop*.

3.4.1 The {if – elseif – else – end} statement

An **if** statement can be followed by an (or more) optional **elseif**... and an **else** statement, which is very useful to test various condition. When using **if... elseif...else** statements, there are few points to be considered:

- The **if - end** uses with *one* condition, the **if – else – end** uses with *two* conditions and for more than two uses **if – elseif – else – end.**
- An **if** can have *zero or one* **else** and it must come *after* any **elseif**.
- An **if** can have *zero to many* **elseif** and they must come *before* the **else**.
- The *nested* **if** statements can use one **if** or **elseif** statement inside another **if** or **elseif** statement(s). The *syntax* of **if** statement in MATLAB is:



```
if <expression 1>
    % Executes when the expression 1 is true
     <statement(s)>
elseif <expression 2>
    % Executes when the boolean expression 2 is true
     <statement(s)>
Elseif <expression 3>
    % Executes when the boolean expression 3 is true
     <statement(s)>
else
    % Executes when the none of the above condition is true
     <statement(s)>
end
```

Example 3.6 Write a script file to prompt the user to enter an integer, and then display whether the integer is *zero*, *positive* or *negative*.(named file "testNumber")

Sol:

```
n=input('Enter an integer : ');
if n>0
   disp('The number is Positive')
elseif n<0
   disp('The number is Negative')
else
   disp('The number is Zero')
end
```

>>testNumber
Enter an integer : **9**
The number is Positive

>>testNumber
Enter an integer : **0**
The number is Zero

Example 3.7 Grades are to be assigned as follows:
A 80% - 100%
B 65% - 79%
C 50% - 64%.

Write a script file to prompt the user to input a mark and display the appropriate grade. If the user enters a number *greater* than **100** or *less* than **zero**, display a message that the mark is *invalid*.(file name "testMark"

Sol:

```
mark=input('Enter the mark : ');
if mark > 100 | mark < 0
```



```
   disp('Invalid mark')
elseif mark >= 80
   disp('A')
elseif mark >= 65
   disp('B')
elseif mark >= 50
   disp('C')
else
   disp('Fail')
end
```

>> **testMark**
Enter the mark : 58
C
>> **testMark**
Enter the mark : 91
A
>> **testMark**
Enter the mark : 102
Invalid mark

Example 3.8: write a program to find the Y value when: (named file "Yvalue")

$$Y = \begin{cases} \frac{2}{x^3} & -10 \le x < 0 \\ 2 & x = 0 \\ \frac{2}{\sqrt[3]{x^2+4}} & 0 < x \le 7 \end{cases}$$

Sol:

```
x=input('Enter the x value : ');
if x<-10 | x>7
    disp('Undefined the Y value')
elseif x>=-10 & x<0
    Y=2/x^3;
    fprintf('The Y value = %0.3f\n ',Y);
elseif x>0 & x<=7
    Y=nthroot(sqrt(x^2+4),3);
    fprintf('The Y value = %0.3f\n ',Y);
else
    fprintf('The Y value = %d\n ',2);
end
```

>> **Yvalue**
Enter the x value : **9**
Undefined the Y value

>> **Yvalue**
Enter the x value : **0**
The Y value = 2



>> **Yvalue**
Enter the x value : **-5**
The Y value = -0.016

>> **Yvalue**
Enter the x value : **1**
The Y value = 1.308

3.4.2 The {switch} statement

A **switch** block conditionally executes one set of statements from several choices. Each choice is covered by a **case** statement. The switch block tests each case until one of the cases is *true*.

When a **case** is *true*, MATLAB executes the corresponding statements and then *exits* the switch block. The **otherwise** block is *optional* and executes only when *no case* is *true*. The *syntax* of **switch** statement in MATLAB is:

switch <switch_expression>
case <case_expression>
    <statement(s)>
case <case_expression>
    <statement(s)>
...
...
otherwise
    <statement(s)>
end

Example 3.9: Write a program to select a color by entering the 1st letter of its name: And handle the invalid letter (named with "colortest")

G	**Green**
Y	**Yellow**
W	**White**
R	**Red**
B	**Blue**
C	**Cyan**

Sol:

```
color=input('Enter color letter : ','s');
switch color
    case {'G','g'}
        disp('The color is Green');
    case {'Y','y'}
        disp('The color is Yellow');
    case {'R','r'}
```



```
        disp('The color is Red');
    case {'W','w'}
        disp('The color is White');
    case {'B','b'}
       disp('The color is Blue');
    case {'C','c'}
        disp('The color is Cyan');
    otherwise
        disp('Undefined Color');
end
```

>> **colortest**
Enter color letter : **y**
The color is **Yellow**

>> **colortest**
Enter color letter **: b**
The color is **Blue**

>> **colortest**
Enter color letter **: R**
The color is **Red**

Example 3.10: Write a program to display the name of day by giving the day number as the following:

1	Saturday
2	Sunday
3	Monday
4	Tuesday
5	Wednesday
6	Thursday
7	Friday

And handle the invalid number (named file “dayname”)

Sol:

```
Nday=input('Enter The Day Number : ');
switch Nday
    case 1
        disp('The day is SATURDAY');
    case 2
         disp('The day is SUNDAY');
    case 3
         disp('The day is MONDAY');
    case 4
         disp('The day is TUESDAY');
    case 5
         disp('The day is WEDNESDAY');
```



```
    case 6
         disp('The day is THURSDAY');
    case 7
         disp('The day is FRIDAY');
    otherwise
         disp('Invalid');
end
```

>> **dayname**
Enter The Day Number : **3**
The day is MONDAY

>> **dayname**
Enter The Day Number : **7**
The day is FRIDAY

>> **dayname**
Enter The Day Number : **10**
Invalid

3.4.3 The {for loop} statement

A **for loop** is a repetition control structure that allows you to efficiently write a loop that needs to execute a specific number of times. The *syntax* of **for** statement in MATLAB is:

for index = *initval : step : endval*
  <statement(s)>
  ...
end

Increments **index** by the value **step** on each iteration, or decrements when **step** is *negative*, **step** is *omitted* when increment is **1**. For examples:

```
>> for N=10:20                    % print the N values range between (10→20)
fprintf ('value of N: %d\n', N);
end
value of N: 10
value of N: 11
value of N: 12
value of N: 13
value of N: 14
value of N: 15
value of N: 16
value of N: 17
value of N: 18
value of N: 19
value of N: 20
```

```
>> for A = [24,18,17,23,28]          % display the values of A
disp(A)
end
     24
     18
     17
     23
     28
```

Example 3.11: Create a script file to find $\sum_{x=1}^{x=n} \sqrt{x}$

Sol:

```
N=input('Enter N value : ');
xsum=0;
for x=1:N
    xsum=xsum+sqrt(x);
end
disp(xsum)
```

Enter N value : **7**
  13.4776

Note: MATLAB allows to use more than one **for** loop, each one inside another loop. The *syntax* for a *nested* **for** loop statement in MATLAB is:

```
for m = 1:j
    for n = 1:k
      <statements>;
    end
end
```

Example 3.12: Create a script file to generate **(N X N)** matrix in form like:

$$\begin{bmatrix} \mathbf{1} & 0 & \mathbf{1} & 0 & \mathbf{1} \\ 0 & \mathbf{1} & 0 & \mathbf{1} & 0 \\ \mathbf{1} & 0 & \mathbf{1} & 0 & \mathbf{1} \\ 0 & \mathbf{1} & 0 & \mathbf{1} & 0 \\ \mathbf{1} & 0 & \mathbf{1} & 0 & \mathbf{1} \end{bmatrix}$$

Sol:

```
clear
N=input('Enter the square matrix size : ');
for i=1:N
    for j=1:N
        if mod(i+j,2)==0
            a(i,j)=1;
        else
            a(i,j)=0;
        end
    end
end
```



```
disp(a)
```

Enter the square matrix size : **5**

1	0	1	0	1
0	1	0	1	0
1	0	1	0	1
0	1	0	1	0
1	0	1	0	1

Example 3.13: Given two vectors **x** and **y** with random values, create a matrix **A** whose elements are defined as $A_{ij} = x_i . y_j$. Write a script file.

Sol:

```
clear;clc
x=randi([1,10],1,7); % generates row vector x(7)
y=randi([1,4],5,1);  % generates column vector y(5)
for i=1:length(x)
    for j=1:length(y)
        A(i,j)=x(i)*y(j);
    end
end
disp(A)
```

3.4.4 The {while loop} statement

The **while** loop repeatedly executes statements while *condition* is **true**. The *syntax* of a **while** loop in MATLAB is:

while <expression>
  <statement(s)>
  …
End

The **while** loop repeatedly executes program statement(s) as long as the expression remains *true*.

Example 3.14: Write a program to compute the series:

$$y = 1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n} \qquad n \neq 0$$

Sol:

```
clc;clear
N=input('Enter N value : ');
Y=0;i=1;
while i<=N
    Y=Y+1/i;
    i=i+1;
end
fprintf('Y= %.3f\n',Y)
```



Enter N value : **8**
Y= 2.718

Example 3.15: Write a program to generate the series (***1, 2, 4, 8, … , 1024***) and display it as a vector

Sol:

```
clear;clc;
n=0;i=1;
while 2^n<=1024
    A(i)=2^n;
    n=n+1;
    i=i+1;
end
disp(A')
```

```
     1
     2
     4
     8
    16
    32
    64
   128
   256
   512
  1024
```

Note: Also likes **for** loop, more than one **while** loop are used, one loop *inside* the another loop. The *syntax* for a *nested* **while** loop statement in MATLAB is as follows:

while <expression1>
    while <expression2>
        <statement(s)>
    end
end

Example 3.16: Write a program to display the **multiplication table**

Sol:

```
clc;clear;
x=1;
while x<=10
    y=1;
    while y<=10
        z(x,y)=x*y;
        y=y+1;
    end
    x=x+1;
```



```
end
disp(z);
```

1	2	3	4	5	6	7	8	9	10
2	4	6	8	10	12	14	16	18	20
3	6	9	12	15	18	21	24	27	30
4	8	12	16	20	24	28	32	36	40
5	10	15	20	25	30	35	40	45	50
6	12	18	24	30	36	42	48	54	60
7	14	21	28	35	42	49	56	63	70
8	16	24	32	40	48	56	64	72	80
9	18	27	36	45	54	63	72	81	90
10	20	30	40	50	60	70	80	90	100

3.4.5 The {break} statement

The **break** statement *terminates* execution of **for** or **while** loop. Statements in the loop that appear after the **break** statement are not executed. In nested loops, **break** exits only from the loop in which it occurs. Control passes to the statement following the end of that loop.

Example 3.17 Write a program to generate 100 random numbers range (1-50) use **rand** function, ***stop*** the operation when number =**33**. *Display* the numbers until it stopped

Sol:

```
clc;clear
counter=0;
for i=1:100
    x=((50-1)*rand+1);      % generate value range between(1-50)
    disp(x)
    counter=counter+1;
    if fix(x)==33           % the stop condition(convert to integer)
        fprintf('The no. of generated elements = %d\n
',counter)
        break;              % exit from loop
    end
end
```

```
   20.6923
   31.9646
   49.2766
   28.4144
   46.7460
   36.2968
   24.7179
   32.3125
   44.4942
   10.7381
   20.3729
   49.6166
   20.7152
   33.2840
```



The no. of generated elements = **14**

3.4.6 The {continue} statement

The **continue** statement is used for *passing* control to *next* iteration of **for** or **while** loop. The **continue** statement in MATLAB works somewhat like the **break** statement. Instead of forcing termination, however, '**continue**' forces the next iteration of the loop to take place, skipping any code in between.

Example 3.18: Write a program to generate 10 integer x values range (0 – 3) and compute $\boldsymbol{y = \frac{1}{x}}$ $(\boldsymbol{x \neq 0})$, pass by when x=0. Find how many passes and display the y values.

Sol:

```
clear;clc
counter=0;
i=1;
while i<=10
    x=randi([0,3]);          % generates integer random numbers(0-3)
    y(i)=1/x;
    if (x==0)
        i=i+1;
        counter=counter+1;
        continue;
    end
i=i+1;
end
disp(y')
fprintf('the no. of passes : %d\n',counter)
```

```
    0.50
    0.33
    0.50
    0.33
    Inf
    Inf
    0.33
    1.00
    Inf
the no. of passes : 3
```

3.5 User Defined Functions

A **function** is a group of statements that together perform a task. In MATLAB, functions are defined in *separate* files. *The **name** of the file and of the function should be the **same**.*

Functions operate on variables within their own workspace, which is also called the **local workspace**, separate from the workspace you access at the MATLAB command prompt which is called the **base workspace**. **Functions** can accept more than *one input* arguments and may return more than one *output* arguments.

Syntax of a **function** statement is:



function [out1, out2, ... , outN] = **myfun** (in1, in2, in3, ... , inN)

Example 3.19: Create a function file, named ***Nmax*** should be written in a file named ***Nmax.m***. It takes three numbers as argument and returns the maximum of the numbers.

Sol

```
function max=Nmax(a,b,c)
% This function calculates the maximum of the
% three given numbers as input
if (a>b) & (a>c)
    max=a;
elseif (b>c)
    max=b;
else
    max=c;
end
end
```

Comment describing the function

```
>> x= Nmax(12,35,1)
x =
   35
>> x= Nmax(12,-35,1)
x =
   12
>> x= Nmax(-12,-35,1)
x =
    1
```

Note: The comment lines that come right after the function statement provide the **help** text. These lines are *printed* when type:

```
>> help Nmax
 This function calculates the maximum of the
 three given numbers as input
```

Example 3.20: Create a function to *sort* a matrix column by column *ascending*

Sol:

```
function B= sortBycol(array)  % This function sorts a matrix column by column ascending
[m,n]=size(array);
V=reshape(array,1,m*n);       % convert matrix to vector
V=sort(V);
B=reshape(V,m,n);
End
```

```
>> A=[9 5 1 0;7 6 3 4;2 10 12 11]
A =
     9     5     1     0
```

```
      7     6     3     4
      2    10    12    11
>> sortBycol(A)
ans =
      0     3     6    10
      1     4     7    11
      2     5     9    12
```

3.6 Anonymous Functions

An **anonymous function** is a very simple, one-line function. The advantage of an anonymous function is that it does not have to be stored in an *M-file*, it can be created in the *Command Window* or in any *script*. The syntax for an anonymous function is:

fnhandle = @ (arguments) **functionbody**

Where ***fnhandle*** stores the ***function handle***; it is essentially a way of referring to the function. The handle is assigned to this name using the **@** operator. The arguments, in *parentheses*, correspond to the argument(s) that are passed to the function. For examples:

```
>> power = @(x,n) x.^n;  % defines function to compute X^n (where X is scalar or vector)
>> power (2,5)           % computes 2^5
ans =
   32

>> V=[2 3 6 1];          % creates vector V
>> power (V,5)           % computes the power for each elements of V
ans =
      32     243    7776       1
```

Example 3.21: Use an anonymous function to define:

1) ln
2) F° to C° temperature
3) Area of circle

Sol:

```
>> ln = @(x) log(x);     % defines ln
>> ln(500) , log(500)
ans =
   6.2146
ans =
```



```
   6.2146

 >> FtoC = @(f) (f-32)/1.8;          % defines conversion function
 >> FtoC(110)                        % converts 110 F°
ans =
   43.3333
 >> FtoC(0)                          % converts 0 F°
ans =
  -17.7778

 >> circlarea = @ (radius) pi * radius .^2;   % defines function area = πr^2
 >> circlarea (8)
ans =
  201.0619
>> circlarea ([5 9 3 6])
ans =
   78.5398  254.4690   28.2743  113.0973
```

Notes: *Function handles* can also be created for functions other than anonymous functions, both built-in and user defined functions. For example, the following would create a function handle for the built-in **factorial** function:

```
>> Fact = @factorial;
>> Fact(8)
ans =
      40320
```

3.7 Primary and Sub-Functions

Any function other than an anonymous function must be defined within a file. Each function file contains a required primary function that appears first and any number of optional *sub-functions* that comes after the primary function and used by it. *Primary* functions can be called from outside of the file that defines them, either from command line or from other functions, but *sub-functions* cannot be called from command line or other functions, outside the function file. Sub-functions are visible only to the primary function and other sub-functions within the function file that defines them.

Example 3.22 Write a function named **quadratic** that would *calculate the roots* of a *quadratic equation*. The function file **quadratic.m** will contain the *primary* function ***quadratic*** and the *sub-function* ***disc***, which calculates the discriminant.

Sol:

```
Function [x1,x2]= quadratic(a,b,c)
%this function returns the roots of a quadratic equation.
%It takes 3 input arguments which are:
```



```
%the co-efficients of x2, x and the constant
d = disc(a,b,c);
x1 =(-b + d)/(2*a);
x2 =(-b - d)/(2*a);
end                    % end of quadratic(primary function)

function dis = disc(a,b,c)  % function calculates the discriminant
dis = sqrt(b^2-4*a*c);
end                         % end of sub-function
```

>> [r1,r2] = quadratic(-3,6,1)

r1 =

-0.1547

r2 =

2.1547



Exercises

Q1) Write an anonymous function to calculate and return the Volume of:

- Cube
- Cylinder
- Sphere
- Pyramid

Q2) Write an anonymous function to implement this. Compare yours to the built-in function **sinh**.

$$Hyperbolic\ sine(x)=(e^{x}-e^{-x})/2$$

Q3) Write a function *areaperim* that will calculate both the area and perimeter of a *polygon*. For a polygon with *n* sides inscribed in a circle with a radius of *r*, the area *A* and perimeter *P* of the polygon can be found by:

$$A=\frac{1}{2}nr^{2}\ sin\left(\frac{\pi}{n}\right) \qquad P=2nr\ sin\left(\frac{\pi}{n}\right)$$

Q4) The Fibonacci numbers is a sequence of numbers Fi: ***0 1 1 2 3 5 8 …***
Where

a. $F_0 = 0$
b. $F_1 = 1$
c. $F_n = F_{n-2} + F_{n-1}$ *if* $n > 1$

Write a function to implement this definition. The function will receive one integer argument **n**, and it will return one integer value, which is the **nth** Fibonacci number.

Q5) Write a function *conevol* to calculate the cone volume which is given by:

$$V=\frac{1}{3}\pi r^{2}\ h$$

Where ***r*** is the radius of the circular base and ***h*** is the height of the cone

Q6) A *closed cylinder* is being constructed of a material that costs a *dollar* amount per *square foot*. Write a function that will calculate and return the cost of the material, *rounded* up to the nearest square foot. The total surface area for the closed cylinder is:

$$SA=2\pi r^{2}+2\pi rh$$

Where ***r*** is the radius and ***h*** is the height of the cylinder

Q7) Write a simple script that will calculate the volume of a hollow sphere that is



$$\frac{4\pi}{3}(r_0^3 - r_1^3)$$

Where r_1 is the *inner* radius and r_0 is the *outer* radius. Assign a value to a variable for the inner radius, and also assign a value to another variable for the outer radius. Then, using these variables, assign the volume to a third variable.

Q8) Write a function *nexthour* that will receive one integer argument, which is an hour of the day, and will return the next hour. This assumes a **12**-hour clock, so for example the next hour after **12** would be **1**. Here are two examples of calling this function.

>> fprintf('The next hour will be %d.\n',nexthour(3))
The next hour will be 4.

>> fprintf('The next hour will be %d.\n',nexthour(12))
The next hour will be 1.

Q9) Write a script to calculate the ***volume of a pyramid***, which is (***1/3 * base * height***), where the ***base*** is (***length * width***). Prompt the user to enter values for the **length**, **width**, and the **height** and then calculate the volume of the pyramid. When the user enters each value, he or she will then be prompted also for either **i** for *inches*, or **c** for *centimeters*. (Note: ***2.54cm = 1 inch***). The script should print the volume in *cubic inches* with three decimal places. As an example, the format will be:

```
This program will calculate the volume of a pyramid.
Enter the length of the base: 50
Is that i or c? i
Enter the width of the base: 6
Is that i or c? c
Enter the height: 4
Is that i or c? i
The volume of the pyramid is xxx.xxx cubic inches.
```

Q10) Write a function *createvec_m_to_n* that will create and return a vector of integers from **m** to **n** (where **m** is the *first* input argument and **n** is the *second*), regardless of whether **m** is less than **n** or greater than **n**. If **m** is equal to **n**, the vector will just be **1 × 1** or a *scalar*. Here are some examples of calling the function:

```
>> createvec_m_to_n(8,5)
ans =
     5   6   7   8
>> createvec_m_to_n(6,6)
ans =
     6
>> result = createvec_m_to_n(4,5)
result =
```



```
        4   5
>> help createvec_m_to_n
Creates a vector of integers from m to n
```

Q11) Write a script that will generate *one random integer*, and will print whether the random integer is an **even** or an **odd** number.

Q12) A *Pythagorean* triple is a set of positive integers ***(a,b,c)*** such that $a^2 + b^2 = c^2$. Write a function ***ispythag*** that will receive three positive integers (*a, b, c in that order*) and will return **1** for true if they form a Pythagorean triple, or **0** for false if not.

Q13) Write a script ***area_menu*** that will print a list consisting of ***cylinder***, ***circle***, and ***rectangle***. It prompts the user to choose one, and then prompts the user for the appropriate quantities (e.g., the radius of the circle) and then prints its area. If the user enters an *invalid choice*, the script simply prints **an error** message. The script use **switch** statement to accomplish this. Here are two examples of running it (units are assumed to be *inches*).

```
>> area_menu
Menu
1. Cylinder
2. Circle
3. Rectangle
Please choose one: 2
Enter the radius of the circle: 4.1
The area is 52.81

>> area_menu
Menu
1. Cylinder
2. Circle
3. Rectangle
Please choose one: 3
Enter the length: 4
Enter the width: 6
The area is 24.00
```

Q14) Let **x = [3 16 9 12 -1 0 -12 9 6 1].** Provide the command(s) that will:

- *set* the positive values of **x** to **zero**
- *set* values that are multiples of **3** to **3**
- *multiply* the even values of **x** by **5**
- *extract* the values of **x** that are *greater* than **10** into a vector called **y**
- *extract* the values of **x** that are *less* than **0** into a vector called **z**

Q15) Let **A = randi ([-10,10],6,6).** Perform the following (using ***find*** function):

- *find* the indices and list all elements of **A** which are *smaller* than **-3**



- *find* the indices and list all elements of **A** which are smaller than **5** and larger than **-1**
- *remove* those columns of **A** which contain at least one **0** element.

Q16) Assume that the months are represented by numbers from **1** to **12**. Write a script that asks you to provide a month and returns the number of days in that particular month. Alternatively, write a script that asks you to provide a *month name* (e.g. 'June') instead of a *number*. Use the **switch** function.

Q17) Write a ***for*** loop that will print the column of *real* numbers from **1.1** to **2.9** in steps of **0.1**.

Q18) Write a function *sumsteps2* that calculates and returns the sum of **1** to **n** in steps of **2**, where **n** is an argument passed to the function. Do this using a **for** loop

Q19) Write a function *prodby2* that will receive a value of a positive integer ***n*** and will calculate and return the *product* of the odd integers from **1** to ***n*** (or from **1** to ***n*–1** if ***n*** is even).

Q20) Write a function called *geomser* that will receive values of ***r*** and ***n***, and will calculate and return the sum of the geometric series:

$$1 + r + r^2 + r^3 + r^4 + \ldots + r^n$$

Q21) Create a **1 × 6** vector of random integers, each in the range from **1** to **20**. Find the *minimum* and *maximum* values in the vector, also find the sum of vector.

Q22) Write a function that will receive a matrix as an input argument, and will calculate and return the overall *average* of all numbers in the matrix.

Q23) Create a vector of *five* random integers, each in the range from **–10** to **10**. Perform each of the following:

- *Subtract* **3** from each element.
- *Count* how many are positive.
- *Get* the *absolute* value of each element.
- *Find* the *maximum*.

Q24) Write a script that will print the following multiplication table:

```
1
2  4
3  6  9
4  8  12  16
5  10  15  20  25
```



Q25) The mathematician *Euler* proved the following:

$$\frac{\pi^2}{6} = 1 + \frac{1}{4} + \frac{1}{9} + \frac{1}{16} + \cdots$$

Loop until the sum is close to $\pi^2/6$

Q26) Write a script that will continue prompting the user for positive numbers, and storing them in a vector variable, until the user types a negative number.

Q27) An approximation for the *exponential* function can be found using what is called a *Maclaurin* series:

$$e^x \approx 1 + \frac{x^1}{1!} + \frac{x^2}{2!} + \frac{x^3}{3!} + \cdots$$

Write a program to investigate the value of e^x and the ***exp*** function.

Q28) The *area* of a triangle is:

$$area = \sqrt{s(s-a)(s-b)(s-c)}$$

Where ***a***, ***b***, and ***c*** are the lengths of the sides of the triangle, and ***s*** is equal to *half* the sum of the lengths of the three sides of the triangle. Write a script to calculate and print the *area* of the triangle.

Q29) Write a function *convert_sec* to convert seconds in term of (hours : minutes : seconds)

Q30) Determine the *sum* of the first **50** squared numbers with a control ***loop***.

Q31) Given **x** = **[4 1 6 -1 -2 2]** and **y** = **[6 2 -7 1 5 -1]**, *compute* matrices whose elements are created according to the following formulas:

- $a_{ij} = y_i \,/\, x_j$
- $b_i = x_i \; y_i$
- $c_{ij} = x_i \,/(2 + x_i + y_j)$
- $d_{ij} = 1/\, max(x_i; \; y_j)$

Q32) Write a script that *transposes* a matrix **A**. Check its correctness with the Matlab operation: **A'**.



Q33) Create an **m-by-n** array of *random* numbers (use ***rand*** function). Move through the array, element by element, and set any value that is *less* than **0.5** to **0** and any value that is *greater* than or *equal* to **0.5** to **1**.

Chapter 4

Symbolic Mathematics And Calculus



4.1 Symbolic Algebra

Symbolic mathematics is used regularly in math, engineering, and science classes. It is often preferable to manipulate equations *symbolically* before you substitute values for variables. Its means doing mathematics on symbols (not numbers!).For example, ***a* + *a*** is ***2a***. The symbolic math functions are in the *Symbolic Math Toolbox* in MATLAB. Toolboxes contain related functions and are add-ons to MATLAB. The Symbolic Math Toolbox includes an alternative method for solving equations.

4.2 Creating Symbolic Variables and Expressions

Before we can solve any equations, we need to create some symbolic variables. Simple symbolic variables can be created in two ways. For example, to create the symbolic variable ***x*** , type either

```
>> syms x
```
OR
```
>> x = sym ('x');
```

Both techniques set the character **'x'** equal to the symbolic variable **x**. More complicated variables can be created by using existing symbolic variables, as in the expression:

```
>> y = 2*(x - 3)^2/(x^2 + 6*x -9)
```

Notice that both **x** and **y** are *symbolic* variables, **whos** command showing that:

```
>> whos
  Name        Size                Bytes  Class     Attributes

  x           1x1                    60  sym
  y           1x1                    60  sym
```

The **syms** command is particularly convenient, because it can be used to create multiple symbolic variables at the same time, as with the command

```
>>syms pi r h
```

These variables could be combined mathematically to create another symbolic variable, **Vcylinder**:

```
>> Vcylinder=pi*r^2*h
```

The **sym** function can be used to create either an entire expression or an entire equation. For example



>> E = sym ('m*c^2')

Creates a symbolic variable named **E**. Notice that **m** and **c** are not listed in the workspace window, they have not been specifically defined as symbolic variables. Instead, the input to **sym** was a character string, identified by the *single quotes* inside the function.

```
>> whos
  Name            Size                Bytes  Class     Attributes

  Vcylinder       1x1                    60  sym
  h               1x1                    60  sym
  r               1x1                    60  sym
```

All basic mathematical operations can be performed on symbolic variables and expressions (e.g., add, subtract, multiply, divide, raise to a power, etc.). For examples:

```
>> A=sym('x^2');
>> B=sym('x^4');
>> A/B
ans =
1/x^2

>> sqrt(B)
ans =
(x^4)^(1/2)

>> A^3
ans =
x^6

>> B*A
ans =
x^6

>> A + sym('5*x^2')          % adding the x^2 and 5x^2 to result in 6x^2
ans =
6*x^2

>> sym ('z^3 + 2*z^3')
ans =
3*z^3
```



Example 4.1: Generate symbolic series:

$$y = x \;\; x^2 \;\; x^3 \;\; x^4 \;\ldots\; x^n$$

$$z = \frac{1}{2x} \;\; \frac{1}{4x} \;\; \frac{1}{6x} \ldots \frac{1}{2nx}$$

Sol:

```
>> syms x
>> n=7;
>> y=x.^(1:n)                % uses element by element .^ with variable x
y =
[ x, x^2, x^3, x^4, x^5, x^6, x^7]

>> n=16;
>> z=1./(x*(2:2:n))          % uses element by element ./ with variable x
z =
[ 1/(2*x), 1/(4*x), 1/(6*x), 1/(8*x), 1/(10*x), 1/(12*x), 1/(14*x), 1/(16*x)]
```

Notice that the series results appeared as vectors. Also the **sym** function uses to create the symbolic matrix. For examples:

```
>> A= sym('a', 3)            % creates a matrix (3x3) of symbolic variable a
A =
[ a1_1, a1_2, a1_3]
[ a2_1, a2_2, a2_3]
[ a3_1, a3_2, a3_3]

>> A= sym('a' , [3 5])       % creates a matrix (3x5) of symbolic variable a
A =
[ a1_1, a1_2, a1_3, a1_4, a1_5]
[ a2_1, a2_2, a2_3, a2_4, a2_5]
[ a3_1, a3_2, a3_3, a3_4, a3_5]

>> A= sym ('a%d%d' , [3 5])  % approve the appearance by removing underscore
A =
[ a11, a12, a13, a14, a15]
[ a21, a22, a23, a24, a25]
[ a31, a32, a33, a34, a35]
```

Example 4.2: Use an anonymous function to create a symbolic matrix as shown:

$$\begin{bmatrix} \frac{1}{1*1} & \cdots & \frac{1}{1*n} \\ \vdots & \ddots & \vdots \\ \frac{1}{m*1} & \cdots & \frac{1}{n*m} \end{bmatrix}$$



Sol:

```
>> SymbolicMatrix=@(m,n)  sym(1./((1:m)'*(1:n)));
>> SymbolicMatrix(3,5)
ans =
[   1, 1/2, 1/3,  1/4,  1/5]
[ 1/2, 1/4, 1/6,  1/8, 1/10]
[ 1/3, 1/6, 1/9, 1/12, 1/15]
```

Note: In symbolic expressions the **real** numbers are converted to **rational values**, for examples:

```
>> sym (2.5 + 3.75)
ans =
25/4

>> sym(sqrt(5.5))
ans =
(2^(1/2)*11^(1/2))/2

>> sym(9.7^2/2)
ans =
9409/200
```

4.3 Simplification Functions

There are several functions that work with expressions, and simplify the terms such as:

4.3.1 simplify Function

The ***simplify*** function does whatever it can to *simplify* expressions, including gathering *like terms*. For example:

```
>> y=sym ('x^2+3*x-5=1');
>> simplify (y)
ans =
x*(x + 3) = 6

>> z = sym ('3*x-(x+3)*(x-3)^2');
>> simplify (z)
ans =
3*x - (x - 3)^2*(x + 3)

>> w = sym ('x^3-1 = (x-3)*(x+3)');
>> simplify (w)
ans =
```



```
x^3 + 8 = x^2
```

```
>> syms x
>> simplify (cos(x)^2 + sin(x)^2)
ans =
1

>> simplify (tan(x)^2 - sec(x)^2)
ans =
-1
```

4.3.2 expand Function

The ***expand*** function *multiplies* out all the portions of the expression or equation. For examples:

```
>> syms x
>> expand ((x-2)*(x-4))
ans =
x^2 - 6*x + 8

>> syms y
>> expand (sin(x+y))
ans =
cos(x)*sin(y) + cos(y)*sin(x)

>> expand ((x+8)^3)
ans =
x^3 + 24*x^2 + 192*x + 512

>> y = (1-x)^3
y =
-(x - 1)^3
>> expand (y)
ans =
- x^3 + 3*x^2 - 3*x + 1
```

4.3.3 factor Function

The ***factor*** function uses to *analysis* the equations to their factors. For examples:

```
>> syms x
>> factor (x^3-1)
ans =
(x - 1)*(x^2 + x + 1)

>> syms y
```



```
>> factor (x^4-y^2)
ans =
(x^2 - y)*(x^2 + y)

>> factor (sym('15236987456'))     % factors symbolic number
ans =
    2^6*173*1376173
```

4.3.4 collect Function

The ***collect*** function uses to *collect* coefficients. For examples:

```
>> syms x,y
>> collect ((x-1)^2*(x+1))
ans =
x^3 - x^2 - x + 1

>> collect ((x^2-x^3)^2-4)
ans =
x^6 - 2*x^5 + x^4 - 4

>> collect ((x^2-x^3)^2*(y-1))
ans =
(y - 1)*x^6 + (2 - 2*y)*x^5 + (y - 1)*x^4

>> collect ( (x^3-x^2)^2 - (y-1)^3 , y)     % collects y coefficient only
ans =
- y^3 + 3*y^2 - 3*y + (x^2 - x^3)^2 + 1
```

4.3.5 subs Function

The ***subs*** function will *substitute* a value for a symbolic variable in an expression or in an equation. For examples:

```
>> myexp = x^3 + 3*x^2 - 2
myexp =
x^3 + 3*x^2 - 2
>> x = 3;
>> subs (myexp,x)
ans =
     52

>> syms x y
>> subs (x^2-y^2+x*y-3 , x , 2)     % substitutes variable x only with 2
ans =
```



```
- y^2 + 2*y + 1
>> subs (x^2-y^2+x*y-3,{x , y},{3 , -2})    % substitutes variables x and y with (3 , -2)
ans =
     -4
```

4.4 Solving Expressions and Equations

A highly useful function in the symbolic toolbox is ***solve***. It can be used to determine the *roots* of expressions, to find numerical answers when there is a single variable, and to solve for an unknown symbolically. The ***solve*** function can also solve systems of equations, the ***solve*** function allows the user to find analytical solutions to a variety of problems.

4.4.1 solve Function

The function ***solve*** solves an equation and returns the *root(s)* as symbolic expressions. The solution can be converted to numbers using any *numeric* function, such as ***double***, for example:

```
>> syms x
>> R = solve ('2*x^2 + x = 6')
R =
  -2
 3/2

>> double (R)      % double function converts symbolic results to real numbers
ans =
   -2.0000
    1.5000
```

The ***solve*** function sets the expression equal to *zero* and solves for the roots. For example:

```
>> solve ('3*x^2 + x')
ans =
 -1/3
    0
```

If there is more than one variable, MATLAB *preferentially* solves for ***x***. If there is no ***x*** in the expression, MATLAB finds the variable closest to ***x***. For example:

```
>> solve ('a*x^2+b*x +c')
ans =
 -(b + (b^2 - 4*a*c)^(1/2))/(2*a)
 -(b - (b^2 - 4*a*c)^(1/2))/(2*a)
```

However, it is possible to specify which variable to solve for:



```
>> solve ('a*x^2+b*x +c', 'a')        % solves for variable a
ans =
-(c + b*x)/x^2
```

MATLAB can also solve sets of equations. In this example, the solutions for ***x***, ***y***, and ***z*** are returned as a ***structure*** consisting of fields for ***x***, ***y***, and ***z***. The individual solutions are symbolic expressions stored in fields of the structure.

```
>> R=solve ('4*x-2*y+z=7' , 'x+y+5*z=10' , '-2*x+3*y-z=2')
R =
    x: [1x1 sym]
    y: [1x1 sym]
    z: [1x1 sym]
```

To *refer* to the individual solutions, which are in the structure fields, the *dot operator* (**.**) is used.

```
>> x = R.x          % returns the value of variable x using ( . ) operation
x =
   124/41

>> y = R.y          % returns the value of variable y using ( . ) operation
y =
   121/41

>> z = R.z          % returns the value of variable z using ( . ) operation
z =
   33/41
```

The ***double*** function can then be used to *covert* the *symbolic expressions* to *numbers*, and store the results from the three unknowns in a vector.

```
>> double ([x y z])
ans =
    3.0244    2.9512    0.8049
```

4.5 Calculus

MATLAB provides various ways for solving problems of differential and integral calculus, solving differential equations of any degree and calculation of limits. Best of all, you can easily plot the graphs of complex functions and check maxima, minima and other stationery points on a graph by solving the original function, as well as its derivative. We will deal with the problems of calculus, and discuss pre-calculus concepts i.e., calculating limits of functions and verifying the



properties of limits. We will also discuss solving differential equations. Finally, we will discuss integral calculus.

4.5.1 Limit Calculation

The ***limit*** function takes *expression* as an argument and finds the limit of the expression as the independent variable goes to zero. For examples:

```
>> syms x
>> limit ((x^3+5)/(x^4+7))          % x tends to zero as default
ans =
     5/7

>> limit ((x -3)/(x-1) , -1)        % x tends to -1
ans =
     2
```

Algebraic Limit Theorem provides some basic properties of limits. These are as follows:

$$\begin{aligned}
\lim_{x\to p} (f(x)+g(x)) &= \lim_{x\to p} f(x) + \lim_{x\to p} g(x)\\
\lim_{x\to p} (f(x)-g(x)) &= \lim_{x\to p} f(x) - \lim_{x\to p} g(x)\\
\lim_{x\to p} (f(x)\cdot g(x)) &= \lim_{x\to p} f(x) \cdot \lim_{x\to p} g(x)\\
\lim_{x\to p} (f(x)/g(x)) &= \lim_{x\to p} f(x) / \lim_{x\to p} g(x)
\end{aligned}$$

Example 4.3: Two functions: ***f(x) = (3x + 5)/(x - 3)*** and ***g(x) = x^2 + 1***. *Calculate* the ***limits*** of the functions as ***x*** tends to ***4***, of both functions and verify the basic properties of limits using these two functions and MATLAB.

```
>> syms x
>> f = (3*x +5)/(x-3);
>> g = x^2+1;
>> lim_f = limit (f,4)
lim_f =
     17

>> lim_g = limit (g,4)
lim_g =
     17

>> limAdd = limit (f + g,4)
limAdd =
     34
```



```
>> limSub = limit (f - g,4)
limSub =
     0

>> limMulti = limit (f * g,4)
limMulti =
     289

>> limDiv = limit (f / g,4)
limDiv =
     1
```

When limits of a function ***f(x)*** has *discontinuity* at $x = a$. This leads to the concept of *left-handed* and *right-handed* limits. A left-handed limit is defined as the limit as $x \rightarrow a$, from the left, i.e., x approaches **a**, for values of $x < a$. A right-handed limit is defined as the limit as $x \rightarrow a$, from the right, i.e., x approaches a, for values of $x > a$. When the left-handed limit and right-handed limits are *not equal*, the limit does not exist. For example:

```
>> f =(x -3)/abs(x-3);
>> left_lim = limit (f, x, 3, 'left')
left_lim =
     -1

>> right_lim = limit (f, x, 3, 'right')
right_lim =
     1
```

4.5.2 Differential

MATLAB provides the ***diff*** function for computing *symbolic derivatives*. In its simplest form, for examples:

```
>> syms t
>> f = 3*t^2+2*t^(-2);
>> diff (f)
ans =
     6*t - 4/t^3

>> syms x
>> diff ((x^2 + 3)*(x + 2))
ans =
     2*x*(x + 2) + x^2 + 3

>> der = diff ((2*t^2 + 3*t)/(t^3 + 1))
der =
```



```
(4*t + 3)/(t^3 + 1) - (3*t^2*(2*t^2 + 3*t))/(t^3 + 1)^2
```

>> **diff (cos(x)^2)**

ans =

```
-2*cos(x)*sin(x)
```

>> **diff(exp(3*x^3))**

ans =

```
9*x^2*exp(3*x^3)
```

>> **diff (log(t))**

ans =

```
1/t
```

>> **diff (log10(t))**

ans =

```
1/(t*log(10))
```

To compute higher derivatives of a function ***f***, we use the *syntax*: ***diff(f,n)***. For examples:

>> **f=2*x^3-3*x^2+4*x-1;**

>> **der1= diff (f)**

der1 =

```
6*x^2 - 6*x + 4
```

>> **der2= diff(f,2)**

der2 =

```
12*x - 6
```

>> **der3= diff(f,3)**

der3 =

```
12
```

The basic rules of derivatives for function ***f*** and ***g*** are:

$$
\begin{aligned}
&[f+g]' = f' + g' \\
&[f-g]' = f' - g' \\
&[fg]' = f'g + fg' && \text{product rule} \\
&\left[\frac{f}{g}\right]' = \frac{f'g - fg'}{g^2} && \text{quotient rule} \\
&[g(f)]' = g'(f)\cdot f' && \text{chain rule}
\end{aligned}
$$



Example 4.4: Two functions $f(x)=2x^2-x+2$ and $g(x)=3x^3-8$, prove the product rule $(f \cdot g)'=f'.g + f \cdot g'$. Write a program to verify the result.

Sol:

```
syms x
f=2*x^2-x+2;                    % defining f(x)=2x^2-x+2
g=3*x^3-3*x;                    % defining g(x)=3x^3-3x
lhs=diff(f*g);
rhs= diff(f)*g+diff(g)*f;
if lhs==rhs                     % verify the product role of derivative
    disp('the LHS is equal RHS and the result is :')
    disp(lhs)
else
    disp('there is an Error')
end
```

```
the LHS is equal RHS and the result is :
(9*x^2 - 3)*(2*x^2 - x + 2) - (4*x - 1)*(3*x - 3*x^3)
```

Example 4.5: Write a script to solve a problem**.** Given that a function $y = f(x) = 3\sin(x) + 7\cos(5x)$. We will have to find out whether the equation $f'' + f = -5\cos(2x)$ holds true.

Sol:

```
syms x
y =3*sin(x)+7*cos(5*x);         % defining the function
lhs = diff(y,2)+y;              % evaluting the lhs of the equation
rhs =-5*cos(2*x);               % rhs of the equation
if lhs==rhs
    disp('Yes, the equation holds true');
else
    disp('No, the equation does not hold true');
end
disp('Value of LHS is: ')
disp(lhs);
```

```
No, the equation does not hold true
Value of LHS is:
-168*cos(5*x)
```

Note: MATLAB provides the **dsolve** function for solving differential equations symbolically. The most basic form of the **dsolve** function for finding the solution to a single equation is: ***dsolve ('eqn')*** where ***eqn*** is a text string used to enter the equation. It returns a *symbolic* solution with *default* independent variable is ***t*** and a



set of arbitrary constants that MATLAB labels **C1**, **C2**, and so on. The equation: ***f''(x) + 2f'(x) = 5sin3x*** should be *entered* as: **'D2y + 2Dy = 5*sin(3*x)'** where derivatives are indicated with a **"D"**. For example, the 1st differential equation: **y' =5y**

```
>> s = dsolve ('Dy = 5*y')
s =
    C2*exp(5*t)
```

For 2nd differential equation: **y" - y = 0 , y(0) = -1 , y'(0) = 2.**

```
>> dsolve ('D2y - y = 0' , 'y(0) = -1' , 'Dy(0) = 2')
ans =
     exp(t)/2 - 3/(2*exp(t))
```

To substitute the variable ***t*** with any other variables, the expression should be:

```
>> dsolve ('D2y - y = 0' , 'y(0) = -1' , 'Dy(0) = 2' , 'x')
ans =
     exp(x)/2 - 3/(2*exp(x))
```

4.5.3 Integration

MATLAB provides an ***int*** function for calculating *integral* of an expression. For examples:

```
>> syms x
>> f=2*x;
>> int (f)
ans =
     x^2

>> syms x n
>> int(n^x)
ans =
     n^x/log(n)

>> int (x^n)
ans =
    piecewise([n = -1, log(x)], [n <> -1, x^(n + 1)/(n + 1)])

>> f= sin(n*x);
>> int(f)
ans =
     -cos(n*x)/n
```



```
>> syms a t
>> int (a*cos(pi*t))
ans =
    (a*sin(pi*t))/pi

>> syms x
>> f= int (x^5*cos(5*x))
f =
(24*cos(5*x))/3125 + (24*x*sin(5*x))/625 - (12*x^2*cos(5*x))/125
+ (x^4*cos(5*x))/5 - (4*x^3*sin(5*x))/25 + (x^5*sin(5*x))/5
```

As shown before, the result of integration function seems difficult to understand, so MATLAB provides the ***pretty*** function which returns an expression in a more readable format, for example:

```
>> pretty(f)
                                  2              4                3              5
 24 cos(5 x)   24 x sin(5 x)   12 x  cos(5 x)   x  cos(5 x)   4 x  sin(5 x)   x  sin(5 x)
 ----------- + ------------- - -------------- + ----------- - ------------- + -----------
    3125            625             125              5              25              5
```

The ***int*** function can be used for *definite* integration by passing the *limits* over which you want to calculate the integral. To calculate:

$$\int_a^b f(x)dx = f(b) - f(a)$$

The *syntax* of ***int*** function is: **`int(x,a,b).`** For example, to calculate the value of $\int_2^9 xdx$

```
>> syms x
>> int (x,2,9)
ans =
    77/2
```

Example 4.6: Calculate the *area* enclosed between the x-axis, and the curve $y = x^3-2x+5$ and the ordinates $x = 1$ and $x = 2$.

Sol:

The required area is given by : $A = \int_1^2 (x^3 - 2x + 5)dx$ and the commands are:

```
>> syms x
>> f =x^3-2*x+5;
>> area=int(f,1,2)
area =
    23/4
>> disp('Area = '), disp (double(area))  % converts and displays the result to a real number
Area =
```



```
    5.7500
```

Example 4.7: Find the area under the curve: $f(x) = x^2 \cos(x)$ for $-4 \le x \le 9$.

Sol:

```
>> f=x^2*cos(x);
>> area=int (f,-4,9)
area =
    8*cos(4) + 18*cos(9) + 14*sin(4) + 79*sin(9)

>> fprintf ('the area = %0.3f \n', double(area))
the area = 0.333
```



Exercises

Q1) Create the following symbolic ***expressions*** using either the ***sym*** or ***syms*** functions:

I. $x^2 - 1$
II. $(x+1)^2$
III. $ax^2 - 2$
IV. $ax^2 + bx + c$
V. $ax^3 + bx^2 + cx + d$
VI. $\frac{y^2 - x^2}{a\,xy^{(2b-c)}}$
VII. $sin^2x + cosy$
VIII. $tan^2y + \frac{sinx}{secy}$

Q2) Use the symbolic expressions from **Q1** to find:

1) Multiply **I** and **II** and named result ***y1***
2) Divide **I** by **II** and named result ***y2***
3) Add **III** and **IV** and named result ***y3***
4) Multiply **VII** and **VIII**, named result ***y4***
5) Divide **VII** by **VIII**, named result ***y5***
6) Use the ***factor***, ***expand***, ***collect*** and ***simplify*** functions on ***y1*** ,***y2*** , ***y3*** , ***y4*** and ***y5***.
7) Find the final result of **VII** ,**VIII** and ***y5*** when ***x=30°*** and ***y=70°***

Q3) Define series *symbolically* :

- $1\ \frac{1}{2}\ \frac{1}{3}\ \frac{1}{4} \dots \frac{1}{n}$
- $\frac{3}{1\times2}\ \frac{4}{2\times3}\ \frac{5}{3\times4} \dots \frac{(n+2)}{n\times(n+1)}$
- $-1\ -\frac{1}{3}\ -\frac{1}{9} \dots -\frac{1}{3^n}$
- $\frac{\pi}{2}\ \frac{2\pi}{2}\ \frac{3\pi}{2} \dots \frac{n\pi}{2}$
- $\frac{y}{x}\ \frac{y^2}{x^3}\ \frac{y^3}{x^5} \dots \frac{y^n}{x^{2n-1}}$

Q4) Use the variables and expressions in **Q1**:

- Use the ***solve*** function to solve **I** and **II**
- Use the ***solve*** function to solve **III** , for both ***x*** and ***a***,
- Find the value of ***x*** and ***a*** by solving both **II** and **III**. Then find the same values when (exp. **II** equals ***3***) and (exp. **III** equals ***5***)



Q5) Consider the following system of *linear* equations:

$$5x + 6y - 3z = 10$$
$$3x - 3y + 2z = 14$$
$$2x - 4y - 12z = 24$$

Use the ***solve*** function to solve for ***x***, ***y***, and ***z***, ***resolve*** equations using *linear algebra* techniques.

Q6) Consider the following *nonlinear* system of equations:

$$x^2 + 5y - 3z^3 = 15$$
$$4x + y^2 - z = 10$$
$$x + y + z = 15$$

Solve the *nonlinear* system with the ***solve*** function. Make your results more readable.

- Define a vector ***v*** of the even numbers from ***0*** to ***10***. *Substitute* this vector into **I** and **II**

Q7) Find the ***1st*** derivative of the following expressions:

$$x^2 + x - 1$$
$$sin(x)$$
$$tan(x)$$
$$ln(x)$$

Q8) Find the ***1st*** and ***2nd*** partial derivatives with respect to ***x*** of the following expressions:

$$ax^2 + bx + c$$
$$x^{0.5} - 3y$$
$$tan\,(x + y)$$
$$3x + 4y - 3xy$$
$$2y - 3x^2$$

Refined ***2nd*** partial derivative with respect to ***y***

Q9) Integrate the expressions in **Q8** and **Q9** with respect to ***x***, then integrate the expressions in **Q9** with respect to ***y***

Q10) Perform a double integration with respect to y for each of the expressions in **Q9**

Q11) A college student goes to the cafeteria and buys lunch. The next day he spends twice as much. The third day he spends **$1** less than he did the second day. At the end of **3** days he has spent **$35**. How much did he spend each day? Solve this problem.



Q12) Consider the following set of ***seven*** equations:

$$\begin{aligned}
3x_1 + 4x_2 + 2x_3 - x_4 + x_5 + 7x_6 + x_7 &= 42 \\
2x_1 - 2x_2 + 3x_3 - 4x_4 + 5x_5 + 2x_6 + 8x_7 &= 32 \\
x_1 + 2x_2 + 3x_3 + x_4 + 2x_5 + 4x_6 + 6x_7 &= 12 \\
5x_1 + 10x_2 + 4x_3 + 3x_4 + 9x_5 - 2x_6 + x_7 &= -5 \\
3x_1 + 2x_2 - 2x_3 - 4x_4 - 5x_5 - 6x_6 + 7x_7 &= 10 \\
-2x_1 + 9x_2 + x_3 + 3x_4 - 3x_5 + 5x_6 + x_7 &= 18 \\
x_1 - 2x_2 - 8x_3 + 4x_4 + 2x_5 + 4x_6 + 5x_7 &= 17
\end{aligned}$$

Define a *symbolic* variable for each of the equations, and use MATLAB to *solve* for each unknown. **Compare** the amount of time it takes to solve the preceding problem by using symbolic math with the ***tic*** and ***toc*** functions, whose *syntax* is:

tic
⋮
code to be timed
⋮
toc

Q13) Determine the ***1st*** and ***2nd*** *derivatives* of the following functions:

- $f_1(x) = y = x^3 - 4x^2 + 3x + 8$
- $f_2(x) = y = (x^2 - 2x + 1)(x - 1)$
- $f_3(x) = y = \cos(2x)\sin(x)$
- $f_4(x) = y = 3xe^{4x^2}$

Q14) Use MATLAB symbolic functions to perform the following integrations:

$$\int (x^2 + 1)dx$$

$$\int_{0.2}^{1.3} (x^2 + x)dx$$

$$\int (x^2 + y^2)dx$$

$$\int_{3.5}^{21} (ax^2 + bx + c)dx$$

Q15) The following polynomial represent the ***altitude*** in meters during the first ***48*** hours following the launch of a ***weather balloon***:

$$h(t) = -0.12t^4 + 12t^3 - 380t^2 + 4100t + 220$$



Assume that the unit of ***t*** is hours.

1) Find the *velocity* which it's the ***1st*** *derivative* of the altitude to determine the equation for the velocity of the balloon.
2) Find the *acceleration* is the derivative of *velocity*, or the ***2nd*** *derivative* of the altitude, to determine the equation for the acceleration of the balloon.
3) Determine when the balloon hits the ground. Because ***h*(*t*)** is a *fourth-order* polynomial, there will be four answers. However, only one answer will be physically meaningful.
4) Determine the *maximum* height reached by the balloon. Use the fact that the *velocity* of the balloon is zero at the maximum height.