



Lecture One:

- ✓ Introduction to MATLAB
- ✓ Starting MATLAB
- ✓ Getting started
- ✓ Mathematical functions



➤ Introduction

The name MATLAB stands for MATrix LABoratory. MATLAB was written originally to provide easy access to matrix software developed by the LINPACK (linear system package) and EISPACK (Eigen system package) projects.

MATLAB is a high-performance language for technical computing. It integrates *computation*, *visualization*, and *programming* environment. Furthermore, MATLAB is a modern programming language environment: it has sophisticated *data structures*, contains built-in editing and *debugging tools*, and supports *object-oriented programming*. These factors make MATLAB an excellent tool for teaching and research.

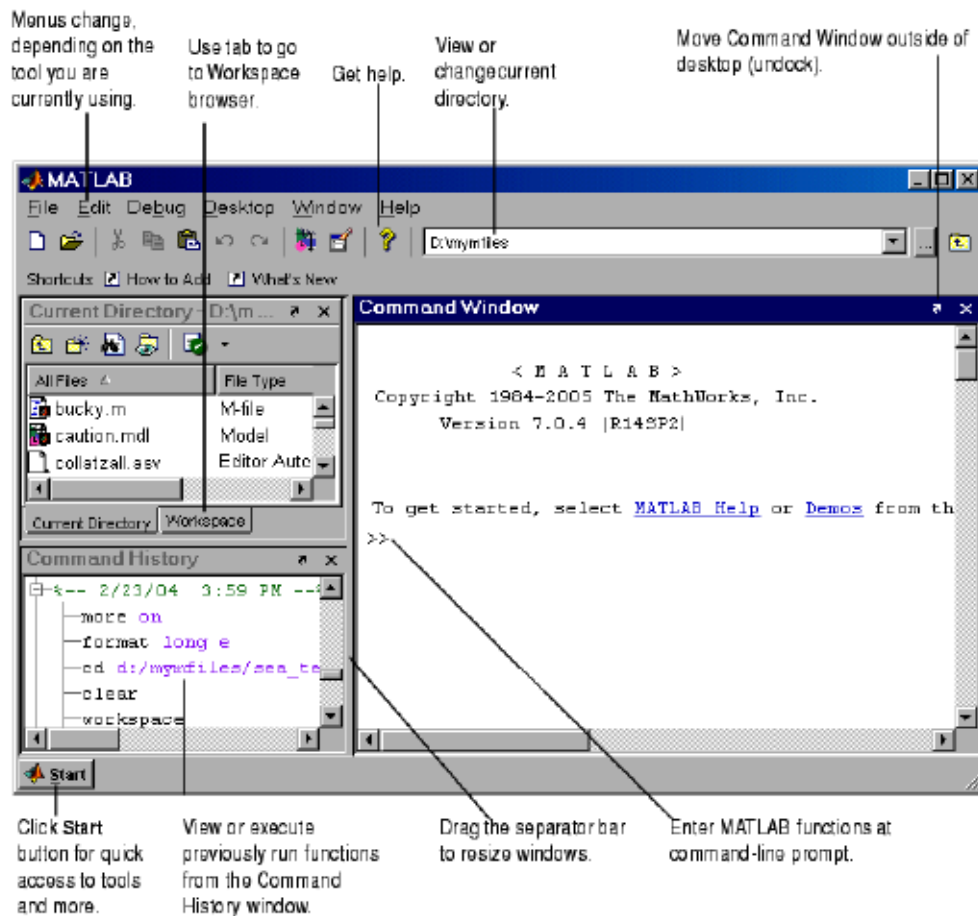
MATLAB has many advantages compared to conventional computer languages (e.g., C, FORTRAN) for solving technical problems. MATLAB is an interactive system whose basic data element is an *array* that does not require dimensioning. The software package has been commercially available since 1984 and is now considered as a standard tool at most universities and industries worldwide.

➤ Starting MATLAB

After logging into your account, you can enter MATLAB by double-clicking on the MATLAB shortcut *icon* (MATLAB) on your Windows desktop. When you start MATLAB, a special window called the MATLAB desktop appears. The desktop is a window that contains *other* windows. The major tools within or accessible from the desktop are:

- The Command Window
- The Command History
- The Workspace
- The Current Directory
- The Help Browser
- The Start button

When MATLAB is started for the first time, the screen looks like the one that shown in the Figure below. This illustration also shows the default configuration of the MATLAB desktop. You can customize the arrangement of tools and documents to suit your needs.



The general interface of MATLAB

➤ Getting started

- Creating MATLAB variables

MATLAB variables are created with an assignment statement. The syntax of variable assignment is :

variable name = a value (or an expression)

For example,

```
>> x = expression
```

where expression is a combination of numerical values, mathematical operators, variables, and function calls. On other words, expression can involve:

- manual entry
- built-in functions
- user-defined functions

- Overwriting variable

Once a variable has been created, it can be reassigned. In addition, if you do not wish to see the intermediate results, you can suppress the numerical output by putting a semicolon (;) at the end of the line. Then the sequence of commands looks like this:

```
>> t = 5;
>> t = t+1
t = 6
```



- **Error messages**

If we enter an expression incorrectly, MATLAB will return an error message. For example, in the following, we left out the multiplication sign, *, in the following expression

```
>> x = 10;
```

```
>> 5x
```

```
??? 5x
```

Error: Unexpected MATLAB expression.

- **Making corrections**

To make corrections, we can, of course retype the expressions. But if the expression is lengthy, we make more mistakes by typing a second time. A previously typed command can be recalled with the **up-arrow key** ↑. When the command is displayed at the command prompt, it can be modified if needed and executed.

- **Controlling the hierarchy of operations or precedence**

Let's consider the previous arithmetic operation, but now we will include parentheses. For example, $1 + 2 * 3$ will become $(1 + 2) * 3$

```
>> (1+2)*3
```

```
ans = 9
```

and, from previous example

```
>> 1+2*3
```

```
ans = 7
```

By adding parentheses, these two expressions give different results: 9 and 7.

Therefore, to make the evaluation of expressions unambiguous, MATLAB has established a series of rules. The order in which the arithmetic operations are evaluated is given in Table below. MATLAB arithmetic operators obey the same precedence rules as those in most computer programs. For operators of *equal* precedence, evaluation is from *left to right*.

PRECEDENCE	MATHEMATICAL OPERATIONS
First	The contents of all parentheses are evaluated first, starting from the innermost parentheses and working outward.
Second	All exponentials are evaluated, working from left to right
Third	All multiplications and divisions are evaluated, working from left to right
Fourth	All additions and subtractions are evaluated, starting from left to right

Now, consider another example:

$$\frac{1}{2+3^2} + \frac{4}{5} \times \frac{6}{7}$$

In MATLAB, it becomes

```
>> 1/(2+3^2)+4/5*6/7
```

```
ans = 0.7766
```

or, if parentheses are missing,

```
>> 1/2+3^2+4/5*6/7
```

```
ans = 10.1857
```



- *Managing the workspace*

The contents of the workspace persist between the executions of separate commands. Therefore, it is possible for the results of one problem to have an effect on the next one. To avoid this possibility, it is a good idea to issue a clear command at the start of each new independent calculation.

>> clear

The command clear or clear all removes all variables from the workspace. This frees up system memory.

In order to display a list of the variables currently in the memory, type

>> who

while, who will give more details which include size, space allocation, and class of the variables.

- **Entering multiple statements per line**

It is possible to enter multiple statements per line. Use commas (,) or semicolons (;) to enter more than one statement at once. Commas (,) allow multiple statements per line without suppressing output.

```
>> a=7; b=cos(a), c=cosh(a)
b = 0.6570
c = 548.3170
```

➤ **Mathematical functions**

MATLAB offers many predefined mathematical functions for technical computing which contains a large set of mathematical functions.

There is a long list of mathematical functions that are *built* into MATLAB. These functions are called *built-ins*. Many standard mathematical functions, such as $\sin(x)$, $\cos(x)$, $\tan(x)$, ex , $\ln(x)$, are evaluated by the functions sin, cos, tan, exp, and log respectively in MATLAB.

Table below lists some commonly used functions, where variables x and y can be numbers, vectors, or matrices.

<code>cos(x)</code>	Cosine	<code>abs(x)</code>	Absolute value
<code>sin(x)</code>	Sine	<code>sign(x)</code>	Signum function
<code>tan(x)</code>	Tangent	<code>max(x)</code>	Maximum value
<code>acos(x)</code>	Arc cosine	<code>min(x)</code>	Minimum value
<code>asin(x)</code>	Arc sine	<code>ceil(x)</code>	Round towards $+\infty$
<code>atan(x)</code>	Arc tangent	<code>floor(x)</code>	Round towards $-\infty$
<code>exp(x)</code>	Exponential	<code>round(x)</code>	Round to nearest integer
<code>sqrt(x)</code>	Square root	<code>rem(x)</code>	Remainder after division
<code>log(x)</code>	Natural logarithm	<code>angle(x)</code>	Phase angle
<code>log10(x)</code>	Common logarithm	<code>conj(x)</code>	Complex conjugate

In addition to the elementary functions, MATLAB includes a number of predefined constant values. A list of the most common values is given in Table below.

<code>pi</code>	The π number, $\pi = 3.14159\dots$
<code>i, j</code>	The imaginary unit i , $\sqrt{-1}$
<code>Inf</code>	The infinity, ∞
<code>NaN</code>	Not a number



Examples:

Algebraic form	Matlab form
$\frac{5^3}{2^4 + 1}$	<code>5^3 / (2^4+1)</code>
$3 \frac{\ln 4 - 2}{(\sqrt{3} + 1)^2} - 5$	<code>3*(log(4)-2)/(sqrt(3)+1)^2-5</code>
$e^4 + \log_{10}(x) - \pi^y$	<code>exp(4)+log10(x)-pi^y</code>
$C = \sin^2\left(\frac{\pi}{2}\right) + \tan(3\pi x)$	<code>C=(sin(pi/2))^2+tan(3*pi*x)</code>
$B = \pi \tan\left\{\frac{7\sqrt{x-\pi}}{12-x}\right\} - \frac{e^{x^5-8} + 4\pi}{\ln^2 x + 1}$	<code>B=pi*tan(7*sqrt(x-pi)/(12-x))-(exp(x^5-8)+4*pi)/((log(x))^2+1)</code>
$R = \frac{\frac{3}{4} \ln \sqrt{x+1} - \csc\left\{\frac{78-\theta}{r+12}\right\}}{5x}$	<code>R=(3/4*log(sqrt(x+1))-csc((78-theta)/(r+12)))/5/x</code>

➤ `round(3.1)`
➤ 3

➤ `ceil(3.1)`
➤ 4

➤ `floor(3.1)`
➤ 3

➤ `fix(3.1)`
➤ 3

➤ `round(-3.1)`
➤ -3

➤ `ceil(-3.1)`
➤ -3

➤ `floor(-3.1)`
➤ -4

➤ `fix(-3.1)`
➤ -3

➤ `round(9.9)`

➤ `ceil(9.9)`

➤ `floor(9.9)`

➤ `fix(9.9)`



Lecture Two:

✓ Matrix generation

- Entering a vector
- Entering a matrix
- Matrix indexing
- Colon operator
- Linear spacing
- Colon operator in a matrix
- Creating a sub-matrix
- Matrix generators



➤ Introduction

Matrices are the basic elements of the MATLAB environment. A matrix is a two-dimensional array consisting of m rows and n columns. Special cases are column vectors ($n = 1$) and row vectors ($m = 1$). In this section we will illustrate how to apply different operations on matrices. MATLAB supports two types of operations, known as matrix operations and array operations.

➤ Matrix generation

Matrices are fundamental to MATLAB. Therefore, we need to become familiar with matrix generation and manipulation. Matrices can be generated in several ways.

- *Entering a vector*

A vector is a special case of a matrix. The purpose of this section is to show how to create vectors and matrices in MATLAB. As discussed earlier, an array of dimension $1 \times n$ is called a row vector, whereas an array of dimension $m \times 1$ is called a column vector. The elements of vectors in MATLAB are enclosed by square brackets and are separated by spaces or by commas. For example, to enter a row vector, v , type

```
>> v = [1 4 7 10 13]
v = 1 4 7 10 13
```

Column vectors are created in a similar way, however, semicolon (;) must separate the components of a column vector,

```
>> w = [1;4;7;10;13]
w =
1
4
7
10
13
```

On the other hand, a row vector is converted to a column vector using the transpose operator. The transpose operation is denoted by an apostrophe or a single quote (').

```
>> w = v'
w =
1
4
7
10
13
```

Thus, $v(1)$ is the first element of vector v , $v(2)$ its second element, and so forth. Furthermore, to access blocks of elements, we use MATLAB's colon notation (:). For example, to access the first three elements of v , we write,

```
>> v(1:3)
ans = 1 4 7
```




Or, all elements from the third through the last elements,

```
>> v(3:end)
```

```
ans = 7 10 13
```

where end signifies the last element in the vector. If v is a vector, writing

```
>> v(:)
```

produces a column vector, whereas writing

```
>> v(1:end)
```

produces a row vector.

- **Entering a matrix**

A matrix is an array of numbers. To type a matrix into MATLAB you must

- begin with a square bracket, [
- separate elements in a row with spaces or commas (,)
- use a semicolon (;) to separate rows
- end the matrix with another square bracket,].

Here is a typical example. To enter a matrix A, such as,

$$A = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$$

type,

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

MATLAB then displays the 3 x 3 matrix as follows,

```
A =
```

```
1 2 3
```

```
4 5 6
```

```
7 8 9
```

Note that the use of semicolons (;) here is different from their use mentioned earlier to suppress output or to write multiple commands in a single line.

Once we have entered the matrix, it is automatically stored and remembered in the Workspace. We can refer to it simply as matrix A. We can then view a particular element in a matrix by specifying its location. We write,

```
>> A(2,1)
```

```
ans = 4
```

- **Matrix indexing**

We select elements in a matrix just as we did for vectors, but now we need two indices. The element of row *i* and column *j* of the matrix A is denoted by A(i,j). Thus, A(i,j) in MATLAB refers to the element *A_{ij}* of matrix A. The *first* index is the *row* number and the *second* index is the *column* number. For example, A(1,3) is an element of *first* row and *third* column. Here, A(1,3)=3.

Correcting any entry is easy through indexing. Here we can substitute A(3,3) = 9 by A(3,3) = 0.



- *Colon operator*

The colon operator will prove very useful and understanding how it works is the key to efficient and convenient usage of MATLAB. It occurs in several different forms. Often we must deal with matrices or vectors that are too large to enter one element at a time. For example, suppose we want to enter a vector x consisting of points

(0; 0:1; 0:2; 0:3; ... ; 5). We can use the command

```
>> x = 0:0.1:5;
```

The row vector has 51 elements.

- *Linear spacing*

On the other hand, there is a command to generate linearly spaced vectors: `linspace`. It is similar to the colon operator (:), but gives direct control over the number of points. For example,

```
y = linspace(a,b)
```

generates a row vector y of 100 points linearly spaced between and including a and b .

```
y = linspace(a,b,n)
```

generates a row vector y of n points linearly spaced between and including a to b with steps, where :

$$steps = \frac{b - a}{n - 1}$$

```
>>linspace(1,5,9)
```

```
ans=
```

```
1 1.5 2 2.5 3 3.5 4 4.5 5
```

divides the interval $[1; 9]$ into 10 equal subintervals (0.5), then creating a vector of 10 elements.

```
>> theta = linspace(0,2*pi,101)
```

divides the interval $[0; 2\pi]$ into 100 equal subintervals(0.0628), then creating a vector of 101 elements.

This is useful when we want to divide an interval into a number of subintervals of the same length.

- *Colon operator in a matrix*

The colon operator can also be used to pick out a certain row or column. For example, the statement `A(m:n , k:l)` specifies rows m to n and column k to l . Subscript expressions refer to portions of a matrix. For example,

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

```
ans =
```

```
4 5 6
```

is the second row elements of A .

The colon operator can also be used to extract a sub-matrix from a matrix A .

```
>> A(:,2:3)
```

```
ans =  
2 3  
5 6  
8 0
```



A row or a column of a matrix can be deleted by setting it to a *null* vector, [].

```
>> A(:,2)=[]  
ans =  
    1    3  
    4    6  
    7    0
```

- **Creating a sub-matrix**

To extract a *submatrix* B consisting of rows 2 and 3 and columns 1 and 2 of the matrix A, do the following

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

To interchange rows 1 and 2 of A, use the vector of row indices together with the colon operator.

```
>> C = A([2 1 3],:)  
C =  
    4    5    6  
    1    2    3  
    7    8    0
```

It is important to note that the colon operator (:) stands for all columns or all rows. To create a vector version of matrix A, do the following

```
>> A(:)  
ans =  
    1  
    2  
    3  
    4  
    5  
    6  
    7  
    8  
    0
```

The submatrix comprising the intersection of rows p to q and columns r to s is denoted by $A(p:q,r:s)$.

As a special case, a colon (:) as the row or column specifier covers all entries in that row or column; thus

- $A(:,j)$ is the jth column of A, while
- $A(i,:)$ is the ith row, and
- $A(end,:)$ picks out the last row of A.
-

The keyword end, used in $A(end,:)$, denotes the last index in the specified dimension. Here are some examples.



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

>> A(2:3,2:3)
ans =
     5     6
     8     9

>> A(end:-1:1,end)
ans =
     9
     6
     3
```

- **Deleting row or column**

To delete a row or column of a matrix, use the *empty vector* operator, [].

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

```
     1     2     3
     4     5     6
```

Third row of matrix A is now deleted. To restore the third row, we use a technique for creating a matrix

```
>> A = [A(1,:);A(2,:);[7 8 0]]
```

```
A =
     1     2     3
     4     5     6
     7     8     0
```

Matrix A is now restored to its original form.

- **Dimension**

To determine the dimensions of a matrix or vector, use the command size. For example,

```
>> size(A)
ans =
     3     3
```

means 3 rows and 3 columns, or more explicitly with,

```
>> [m,n]=size(A)
```

- **Transposing a matrix**

The *transpose* operation is denoted by an apostrophe or a single quote ('). It flips a matrix about its main diagonal and it turns a row vector into a column vector. Thus,

```
>> A'
ans =
     1     4     7
     2     5     8
     3     6     0
```

By using linear algebra notation, the transpose of $m \times n$ real matrix A is the $n \times m$ matrix that results from interchanging the rows and columns of A. The transpose matrix is denoted A^T .



- Matrix generators

MATLAB provides functions that generate elementary matrices. The matrix of zeros, the matrix of ones, and the identity matrix are returned by the functions `zeros`, `ones`, and `eye`, respectively.

<code>eye(m,n)</code>	Returns an m-by-n matrix with 1 on the main diagonal
<code>eye(n)</code>	Returns an n-by-n square identity matrix
<code>zeros(m,n)</code>	Returns an m-by-n matrix of zeros
<code>ones(m,n)</code>	Returns an m-by-n matrix of ones
<code>diag(A)</code>	Extracts the diagonal of matrix A
<code>rand(m,n)</code>	Returns an m-by-n matrix of random numbers

```
>> b=ones(3,1)
```

```
b =
```

```
1
1
1
```

Equivalently, we can define b as `>> b = [1;1;1]`

```
>> eye(3)
```

```
ans =
```

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

```
>> c = zeros(2,3)
```

```
c =
```

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

In addition, matrices can be constructed in a block form. With C defined by $C = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$, we may create a matrix D as follows

```
>> D = [C zeros(2); ones(2) eye(2)]
```

```
D =
```

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

Examples

<pre>ones(2)</pre> <pre>1 1</pre> <pre>1 1</pre>	<pre>zeros(2,3)</pre> <pre>0 0 0</pre> <pre>0 0 0</pre>
<pre>ones(1,3)</pre> <pre>1 1 1</pre>	



(a)

```
D=[1:5 ; 6:10 ; 11:2:20];
x=[18;22;21];
K=[x,D ; ones(2,6)];
K(:,2)=99;
K(4,4)=77;
disp(K');
```

Answers:

```
D=[1:5 ; 6:10 ; 11:2:20];
1     2     3     4     5
6     7     8     9    10
11    13    15    17    19
```

```
x=[18;22;21];
18
22
21
```

```
K=[x,D ; ones(2,6)];
18     1     2     3     4     5
22     6     7     8     9    10
21    11    13    15    17    19
1     1     1     1     1     1
1     1     1     1     1     1
```

```
K(:,2)=99;
K=
18    99     2     3     4     5
22    99     7     8     9    10
21    99    13    15    17    19
1    99     1     1     1     1
1    99     1     1     1     1
```

```
K(4,4)=77;
K=
18    99     2     3     4     5
22    99     7     8     9    10
21    99    13    15    17    19
1    99     1    77     1     1
1    99     1     1     1     1
```

```
disp(K')
18    22    21     1     1
99    99    99    99    99
2     7    13     1     1
3     8    15    77     1
4     9    17     1     1
5    10    19     1     1
```

(b)

```
D=[-11:-3:-17; 8:-1:6];
Y=[zeros(2,2),ones(2,1),D];
Y(:,1)=Y(:,4)+12;
Y(:,2)=Y(:,6);
Y(1,2:4)=99;
disp(Y');
```

Answers:

```
D=[-11:-3:-17; 8:-1:6];
-11    -14    -17
8       7       6
```

```
Y=[zeros(2,2),ones(2,1),D];
0       0       1    -11    -14    -17
0       0       1     8     7     6
```

```
Y(:,1)=Y(:,4)+12;
Y=
1       0       1    -11    -14    -17
20      0       1     8     7     6
```

```
Y(:,2)=Y(:,6);
Y=
1    -17     1    -11    -14    -17
20     6     1     8     7     6
```

```
Y(1,2:4)=99;
Y=
1     99     99     99    -14    -17
20     6     1     8     7     6
```

```
disp(Y')
1     20
99     6
99     1
99     8
-14     7
-17     6
```



Lecture Three

- ✓ Array operations
 - *Matrix arithmetic operations*
 - *Array arithmetic operations*



➤ Array operations

MATLAB has two different types of arithmetic operations: matrix arithmetic operations and array arithmetic operations.

- *Matrix arithmetic operations*

As we mentioned earlier, MATLAB allows arithmetic operations: +, -, *, and ^ to be carried out on matrices. Thus,

$A+B$ or $B+A$ is valid if A and B are of the same size

$A*B$ is valid if A 's number of column equals B 's number of rows

A^2 is valid if A is square and equals $A*A$

$\alpha*A$ or $A*\alpha$ multiplies each element of A by α

- *Array arithmetic operations*

On the other hand, array arithmetic operations or *array operations* for short, are done *element-by-element*. The period character, ., distinguishes the array operations from the matrix operations. However, since the matrix and array operations are the same for addition (+) and subtraction (-), the character pairs (.+) and (-) are not used. The list of array operators is shown below in Table.

.*	Element-by-element multiplication
./	Element-by-element division
.^	Element-by-element exponentiation

If A and B are two matrices of the same size with elements $A = [a_{ij}]$ and $B = [b_{ij}]$, then the command

```
>> C = A.*B
```

produces another matrix C of the same size with elements $c_{ij} = a_{ij}b_{ij}$. For example, using the same 3 x 3 matrices,

$$A = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}, \quad B = \begin{bmatrix} 10 & 20 & 30 \\ 40 & 50 & 60 \\ 70 & 80 & 90 \end{bmatrix}$$

we have,

```
>> C = A.*B
```

C =

```
10    40    90
160   250   360
490   640   810
```

To raise a scalar to a power, we use for example the command 10^2 . If we want the operation to be applied to each element of a matrix, we use $.^2$. For example, if we want to produce a new matrix whose elements are the square of the elements of the matrix A , we enter



```
>> A.^2
```

```
ans =
```

```
    1    4    9
   16   25   36
   49   64   81
```

The relations below summarize the above operations. To simplify, let's consider two vectors U and V with

$U.*V$ produces $[u_1v_1 \ u_2v_2 \ \dots \ u_nv_n]$
 $U./V$ produces $[u_1/v_1 \ u_2/v_2 \ \dots \ u_n/v_n]$
 $U.^V$ produces $[u_1^{v_1} \ u_2^{v_2} \ \dots \ u_n^{v_n}]$

$$\begin{array}{c}
 \text{A} \text{ B} = \text{C} \\
 \begin{array}{ccc}
 \underbrace{\hspace{1cm}} & \underbrace{\hspace{1cm}} & \underbrace{\hspace{1cm}} \\
 n \times m & m \times p & n \times p
 \end{array}
 \end{array}$$

OPERATION	MATRIX	ARRAY
Addition	+	+
Subtraction	−	−
Multiplication	*	.*
Division	/	./
Left division	\	.\
Exponentiation	^	.^



Examples:

```
X=[1,2,3;4,5,6]
Y=[12,11,10;9,8,7]
>>X+Y
ans =
    13    13    13
    13    13    13

>>X-Y
ans =
   -11   -9   -7
    -5   -3   -1

>>X+3
ans =
     4     5     6
     7     8     9

>>X*3
ans =
     3     6     9
    12    15    18

>>X.*Y
ans =
    12    22    30
    36    40    42

>>X*Y'
ans =
    64    46
   163   118

>>X'*Y
ans =
    48    43    38
    69    62    55
    90    81    72
```



Lecture Four

Solving Linear Equations in MATLAB



➤ Solving linear equations

One of the problems encountered most frequently in scientific computation is the solution of systems of simultaneous linear equations. With matrix notation, a system of simultaneous linear equations is written

$$Ax = b$$

where there are as many equations as unknown. A is a given square matrix of order n , b is a given column vector of n components, and x is an unknown column vector of n components.

In linear algebra we learn that the solution to $Ax = b$ can be written as $x = A^{-1}b$, where A^{-1} is the inverse of A . For example, consider the following system of linear equations

$$x + 2y + 3z = 1$$

$$4x + 5y + 6z = 1$$

$$7x + 8y = 1$$

The coefficient matrix A is

$$A = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 0 \end{bmatrix}$$

and the vector b is

$$b = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix}$$

There are typically two ways to solve for x in MATLAB:

1. The first one is to use the matrix inverse, `inv`.

```
>> A = [1 2 3; 4 5 6; 7 8 0];
```

```
>> b = [1; 1; 1];
```

```
>> x = inv(A)*b
```

```
x =
```

```
-1.0000
```

```
1.0000
```

```
-0.0000
```



2. The second one is to use the backslash (\) operator. The numerical algorithm behind this operator is computationally efficient. This is a numerically reliable way of solving system of linear equations by using a well-known process of Gaussian elimination.

```
>> A = [1 2 3; 4 5 6; 7 8 0];
```

```
>> b = [1; 1; 1];
```

```
>> x = A\b
```

x =

-1.0000

1.0000

-0.0000

Ex. Solving a set of linear equations

$$-6x = 2y - 2z + 15$$

$$4y - 3z = 3x + 13$$

$$2x + 4y - 7z = -9$$

First, rearrange the equations

$$-6x - 2y + 2z = 15$$

$$-3x + 4y - 3z = 13$$

$$2x + 4y - 7z = -9$$

Second, write the equations in a matrix form $Ax = b$

The coefficient matrix is

$$A = \begin{bmatrix} -6 & -2 & 2 \\ -3 & 4 & -3 \\ 2 & 4 & -7 \end{bmatrix}$$

The constant column vector is

$$b = \begin{bmatrix} 15 \\ 13 \\ -9 \end{bmatrix}$$

Third, solve the simultaneous equations in Matlab

```
>> x = A\b
```

The Matlab answer is:

x = -2.7273

2.7727

2.0909



Lecture Five

Programming in MATLAB



➤ Introduction

So far in these lab sessions, all the commands were executed in the Command Window. The problem is that the commands entered in the Command Window cannot be saved and executed again for several times. Therefore, a different way of executing repeatedly commands with MATLAB is:

1. to *create* a file with a list of commands,
2. *save* the file, and
3. *run* the file.

If needed, corrections or changes can be made to the commands in the file. The files that are used for this purpose are called script files or *scripts* for short.

This section covers the following topics:

- M-File Scripts
- M-File Functions

➤ M-File Scripts

A *script file* is an external file that contains a sequence of MATLAB statements. Script files have a filename extension **.m** and are often called M-files. M-files can be *scripts* that simply execute a series of MATLAB statements, or they can be *functions* that can accept arguments and can produce one or more outputs.

Example 1

Consider the system of equations:

$$x + 2y + 3z = 1$$

$$3x + 3y + 4z = 1$$

$$2x + 3y + 3z = 2$$

Find the solution x to the system of equations.

Solution:

1. Use the MATLAB *editor* to create a file: File → New → M-file.
2. Enter the following statements in the file:
 $A = [1 \ 2 \ 3; 3 \ 3 \ 4; 2 \ 3 \ 3];$
 $b = [1; 1; 2];$
 $x = A \backslash b$
3. Save the file, for example, example1.m.
4. Run the file, in the command line, by typing:
`>> example1`

```
x =  
-0.5000  
1.5000  
-0.5000
```

When execution completes, the variables (A, b, and x) remain in the workspace. To see a listing of them, enter *whos* at the command prompt.

Note: The MATLAB editor is both a text editor specialized for creating M-files and a graphical MATLAB debugger. The MATLAB editor has numerous menus for tasks such as *saving*, *viewing*, and *debugging*.



Because it performs some simple checks and also uses color to differentiate between various elements of codes, this text editor is recommended as the tool of choice for writing and editing M-files.

There is another way to open the editor:

```
>> edit
```

or

```
>> edit filename.m
```

to open filename.m.

➤ M-File functions

As mentioned earlier, functions are programs (or routines) that accept input arguments and return output arguments. Each M-file function (or function or M-file for short) has its own area of workspace, separated from the MATLAB base workspace.

- Anatomy of a M-File function

This simple function shows the basic parts of an M-file.

- 1) function f = factorial(n)
- 2) % FACTORIAL(N) returns the factorial of N.
- 3) % Compute a factorial value.
- 4) f = prod(1:n);

The first line of a function M-file starts with the keyword function. It gives the function name and order of arguments. In the case of function factorial, there are up to one output argument and one input argument.

Part no.	M-file element	Description
(1)	Function definition line	Define the function name, and the number and order of input and output arguments
(2)	H1 line	A one line summary description of the program, displayed when you request Help
(3)	Help text	A more detailed description of the program
(4)	Function body	Program code that performs the actual computations

As an example, for n = 5, the result is,

```
>> f = factorial (5)
```

```
f =
```

```
120
```




Both functions and scripts can have all of these parts, except for the function definition line which applies to function only. In addition, it is important to note that function name must begin with a letter, and must be no longer than the maximum of 63 characters. Furthermore, the name of the text file that you save will consist of the function name with the extension .m. Thus, the above example file would be factorial.m.

Table below summarizes the differences between scripts and functions.

SCRIPTS	FUNCTIONS
<ul style="list-style-type: none"> - Do not accept input arguments or return output arguments. - Store variables in a workspace that is shared with other scripts - Are useful for automating a series of commands 	<ul style="list-style-type: none"> - Can accept input arguments and return output arguments. - Store variables in a workspace internal to the function. - Are useful for extending the MATLAB language for your application

- **Input and output arguments**

As mentioned above, the input arguments are listed inside parentheses following the function name. The output arguments are listed inside the brackets on the left side. They are used to transfer the output from the function file. The general form looks like this

function [outputs] = function_name(inputs)

Function file can have none, one, or several output arguments. Table below illustrates some possible combinations of input and output arguments.

<code>function C=FtoC(F)</code>	One input argument and one output argument
<code>function area=TrapArea(a,b,h)</code>	Three inputs and one output
<code>function [h,d]=motion(v,angle)</code>	Two inputs and two outputs



- *Input to a script file*

When a script file is executed, the variables that are used in the calculations within the file must have assigned values. The assignment of a value to a variable can be done in three ways.

1. The variable is defined in the script file.
2. The variable is defined in the command prompt.
3. The variable is entered when the script is executed.

We have already seen the two first cases. Here, we will focus our attention on the third one. In this case, the variable is defined in the script file. When the file is executed, the user is prompted to assign a value to the variable in the command prompt. This is done by using the input command.

Example.

```
% This script file calculates the average of points  
% scored in three games.  
% The point from each game are assigned to a variable  
% by using the 'input' command.
```

```
game1 = input('Enter the points scored in the first game ');  
game2 = input('Enter the points scored in the second game ');  
game3 = input('Enter the points scored in the third game ');  
average = (game1+game2+game3)/3
```

The following shows the command prompt when this script file (saved as example) is executed.

```
>> example  
>> Enter the points scored in the first game 15  
>> Enter the points scored in the second game 23  
>> Enter the points scored in the third game 10  
average =  
16
```

- *Output commands*

As discussed before, MATLAB automatically generates a display when commands are executed. In addition to this automatic display, MATLAB has several commands that can be used to generate displays or outputs.

Two commands that are frequently used to generate output are: **disp** and **fprintf**.

The main differences between these two commands can be summarized as follows:

disp	. Simple to use. . Provide limited control over the appearance of output
fprintf	. Slightly more complicated than disp. . Provide total control over the appearance of output



The display *disp* function allows the programmer to display the contents of a string or a matrix in the command window. Although the *disp* function is adequate for many display tasks, the *fprintf* function gives the programmer considerably more control over the way results are displayed. Table below illustrates the various formats supported by *fprintf*

Special Characters	
'	begins and ends a string
%	placeholder used in the <code>fprintf</code> command
%f	fixed-point, or decimal, notation
%d	signed integer notation
%e	exponential notation
%g	either fixed point or exponential notation
%s	string notation
%%	cell divider
\n	linefeed
\r	carriage return (similar to linefeed)
\t	tab
\b	backspace

Example 2:

Write a function file that converts temperature in degrees Fahrenheit (F°) to degrees Centigrade (C°). Use *input* and *fprintf* commands to display a mix of text and numbers. Recall the conversion formulation, $C = 5/9 * (F - 32)$.

Answer

```
function [ C ] = FtoC( F )  
F = input('Enter the temperature in degrees Fahrenheit ');  
C = 5/9 * ( F - 32 )  
fprintf(' The temperature in degrees Centigrade=%d\n', C);  
end
```

This M-file produces the following interaction in the command window:

```
Enter the temperature in degrees Fahrenheit 5  
The temperature in degrees Centigrade = -15
```



Example 3:

Consider the behavior of a freely falling object under the influence of gravity, where the position of the object is described by:

$$d = \frac{1}{2} g t^2$$

Where,

- d - distance the object travels
- g - acceleration due to gravity
- t - elapsed time.

Describe the Input and Output as:

Input

Value of g the acceleration due to gravity, provided by the user (one value)

Time t provided by the user, (starting time, ending time, increments)

Output

Distances calculated for each value of time.

Find the distance traveled by a freely falling object with generate a table of output results (distances calculated with time. Use *disp* and *fprintf* to create a table.

Answer

```
g = input('What is the value of acceleration due to gravity?');
start = input('What starting time would you like?');
finish = input('What ending time would you like?');
incr = input('What time increments would you like calculated?');
time = start:incr:finish;
%Calculate the distance
distance = 1/2*g*time.^2;
%Create a matrix of the output data
result = [time;distance]';
disp(' time,s    distance,m')
disp(result)
fprintf('%5d %10d\n',result)
```

This M-file produces the following interaction in the command window:

What is the value of acceleration due to gravity?5

What starting time would you like?0

What ending time would you like?6

What time increments would you like calculated?2

time,s distance,m

0 0

2 10

4 40

6 90



Lecture Six

Control Flow and Operators



➤ Introduction

MATLAB is also a *programming language*. Like other computer programming languages, MATLAB has some decision making structures for control of command execution. These decision making or *control flow* structures include for loops, while loops, and if-else-end constructions. Control flow structures are often used in script M-files and function M-files.

By creating a file with the extension .m, we can easily write and run programs. We do not need to *compile* the program since MATLAB is an interpretative (not compiled) language.

➤ Control Flow

MATLAB has four control flow structures:

- The If statement,
- The For Loop,
- The While Loop,
- The Switch statement.

- The ``if...end'' structure

MATLAB supports the variants of "if" construct.

- if ... end
- if ... else ... end
- if ... elseif ... else ... end

The simplest form of the if statement is

```
if expression
statements
end
```

Here are some examples based on the familiar quadratic formula.

1.

```
discr = b*b - 4*a*c;
if discr < 0
disp('Warning: discriminant is negative, roots are imaginary');
end
```
2.

```
discr = b*b - 4*a*c;
if discr < 0
disp('Warning: discriminant is negative, roots are imaginary');
else
disp('Roots are real, but may be repeated')
end
```

Discriminant Notes	
$D = b^2 - 4ac$	
$D > 0$	2 real roots
$D = 0$	1 real root
$D < 0$	2 complex roots



```
3. discr = b*b - 4*a*c;  
   if discr < 0  
       disp('Warning: discriminant is negative, roots are imaginary');  
   elseif discr == 0  
       disp('Discriminant is zero, roots are repeated')  
   else  
       disp('Roots are real')  
   end
```

It should be noted that:

- *elseif* has no space between else and if (one word)
- no semicolon (;) is needed at the end of lines containing if, else, end
- indentation of *if* block is not required, but facilitate the reading.
- the end statement is required

Relational and logical operators

A relational operator compares two numbers by determining whether a comparison is *true* or *false*. Relational operators are shown in Table below.

Relational and logical operators

OPERATOR	DESCRIPTION
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to
==	Equal to
~=	Not equal to
&	AND operator
	OR operator
~	NOT operator

Note that the "equal to" relational operator consists of two equal signs (==) (with no space between them), since = is reserved for the *assignment* operator.



Comparisons are either true or false, and most computer programs (including MATLAB) use the number 1 for true and 0 for false. (MATLAB actually takes any number that is not 0 to be true.) If we define two scalars

```
x = 5;
```

```
y = 1;
```

and use a relational operator such as `<`, the result of the comparison

```
x < y
```

is either true or false. In this case, **x** is not less than **y**, so MATLAB responds

```
ans =
```

```
0
```

indicating that the comparison is false. MATLAB uses this answer in selection statements and in repetition structures to make decisions.

Of course, variables in MATLAB ® usually represent entire matrices. If we redefine **x** and **y**, we can see how MATLAB handles comparisons between matrices. For example,

```
x = [ 1, 2, 3, 4, 5];
```

```
y = [-2, 0, 2, 4, 6];
```

```
x < y
```

```
ans =
```

```
1×5 logical array
```

```
0 0 0 0 1
```

MATLAB also allows us to combine comparisons with the logical operators *and* , *not* , and *or* .
The code

```
x = [ 1, 2, 3, 4, 5];
```

```
y = [-2, 0, 2, 4, 6];
```

```
z = [ 8, 8, 8, 8, 8];
```

```
z > x & z > y
```

```
ans =
```

```
1×5 logical array
```

```
1 1 1 1 1
```




Example:

Create a function to determine test grades based on the score and assuming a single input into the function. The grades should be based on the following criteria:

Grade	Score
A	90 to 100
B	80 to 90
C	70 to 80
D	60 to 70
E	<60

Answer

```
function results = grade(x)
%This function requires a scalar input
x = input('Enter the Score ');
if(x>=0 & x<=100)
if(x>=90)
results = 'A';
elseif(x>=80)
results = 'B';
elseif(x>=70)
results = 'C';
elseif(x>=60)
results = 'D';
else
results = 'E';
end
else
results = 'Illegal Input';
end
```



- The ``for...end'' loop

In the *for ... end* loop, the execution of a command is repeated at a fixed and predetermined number of times. The syntax is

```
for variable = expression
    statements
end
```

Usually, expression is a vector of the form **i:s:j** (**start : inc : final**)

A simple example of for loop is

```
for k = 1:3
    a = 5^k
end
```

```
a =
    5
a =
   25
a =
  125
```

A common way to use a for loop is in defining a new matrix. Consider, for example, the code

```
For k = 1:5
    a(k) = k^2
end
```

This loop defines a new matrix, a , one element at a time. Since the program repeats its set of instructions five times, a new element is added to the a matrix each time through the loop, with the following output in the command window:

```
a =
    1
a =
    1    4
a =
    1    4    9
a =
    1    4    9   16
a =
    1    4    9   16   25
```



Another common use for a *for* loop is to combine it with an *if* statement and determine how many times something is true. For example, in the list of test scores shown in the first line, how many are above 90?

```
scores = [76,45,98,97];  
count = 0;  
for k=1:length(scores)  
    if scores(k)>90  
        count = count + 1;  
    end  
end  
disp(count)
```

It is a good idea to indent the loops for readability, especially when they are nested. Note that MATLAB editor does it automatically.

Multiple for loops can be nested, in which case *indentation* helps to improve the readability. The following statements form the 5-by-5 symmetric matrix A with (i; j) element i/j for $j > i$:

```
n = 5; A = eye(n);  
for j=2:n  
    for i=1:j-1  
        A(i,j)=i/j;  
        A(j,i)=i/j;  
    end  
end
```



Example:

Use MATLAB capability to create a degrees-to-radians table from 1 to 360 degree with step 10, you can demonstrate the use of for loops.

```
for k=1:36
    deg(k) = k*10;
    rad(k)=deg(k)*pi/180;
end
t = [deg;rad]
disp('Degrees to Radians')
disp('Degrees Radians')
fprintf('%8.0f %8.2f \n',t)
```

Degrees to Radians

Degrees Radians

10	0.17
20	0.35
30	0.52
40	0.70
50	0.87
60	1.05
70	1.22
80	1.40
90	1.57
100	1.75
110	1.92
120	2.09
130	2.27
140	2.44
150	2.62
160	2.79
170	2.97
180	3.14
190	3.32
200	3.49
210	3.67
220	3.84
230	4.01
240	4.19
250	4.36
260	4.54
270	4.71
280	4.89
290	5.06
300	5.24
310	5.41
320	5.59
330	5.76
340	5.93
350	6.11
360	6.28



Example:

What will be the results of the following program?

```
R=12:-3:2.7;  
A=100;  
for m=1:2:4  
    A=A-R(m);  
    if A<=(14*R(m+1))  
        disp(A);  
    else  
        disp(R(m));  
    end  
end  
disp(m);
```

Answer:

```
R = 12 9 6 3  
m= 1 3
```

```
1s loop  
A= 100-12=88  
88 <= (14*9) True  
disp(A) = 88
```

```
2nd loop  
A= 88-6 = 82  
82 <= (14*3) False  
disp(R(m)) = 6
```

```
disp(m) = 3
```



Lecture Seven

Control Flow and Operators

- **``while...end'' loop**
- **Switch and Case**



➤ The ``while...end'' loop

This loop is used when the number of *passes* is not specified. While loops are similar to for loops. The big difference is the way MATLAB decides how many times to repeat the loop. While loops continue until some criterion is met. The while loop has the form:

```
while expression (criterion)
    statements
end
```

The statements are executed as long as expression is true.

```
x = 1
while x <= 10
    x = 3*x
end
```

It is important to note that if the condition inside the looping is not well defined, the looping will continue *indefinitely*. If this happens, we can stop the execution by pressing Ctrl-C.

One common use for a while loop is error checking of user input. Consider a program where we prompt the user to input a positive number, and then we calculate the log base 10 of that value. We can use a while loop to confirm that the number is positive, and if it is not, to prompt the user to enter an allowed value. The program keeps on prompting for a positive value until the user finally enters a valid number.

```
x = input('Enter a positive value of x')
while (x<=0)
    disp('log(x) is not defined for negative numbers')
    x = input('Enter a positive value of x')
end
y = log10(x);
fprintf('The log base 10 of %4.2f is %5.2f \n',x,y)
```

If, when the code is executed, a positive value of **x** is entered, the while loop does not execute (since **x** is not less than 0). If, instead, a zero or negative value is entered, the while loop is executed, an error message is sent to the command window, and the user is prompted to reenter the value of **x**. The while loop continues to execute until a positive value of **x** is finally entered.



Example:

Create a new function called **fact2** that uses a while loop to find $N!$. Include an if statement to check for negative numbers and to confirm that the input is a scalar.

Answer

```
function output = fact2(x)
%This function uses a while loop to find x!
%The input must be a positive integer
if(length(x)>1 | x<0)
    disp('The input must be a positive integer')
else
%Initialize the running product
a = 1;
%Initialize the counter
k = 1;
while k<x
%Increment the counter
    k = k + 1;
%Calculate the running product
a = a*k;
end
output = a;
end
```

Test the function in the command window:

```
fact2(5)
ans =
120
fact2(-10)
ans =
The input must be a positive integer
fact2([1:10])
ans =
The input must be a positive integer
```




Example:

Write out the values of x^2 for all positive integer values x such that $x^3 < 1000$ using while.

Answer

```
x=1;  
while x^3<1000  
    disp (x^2)  
    x=x+1;  
end
```

Example:

Write a program that computes the sum:

$$P = 4 \sum_{k=0}^{\infty} \frac{k}{2k+1}$$

Stop the summation operation when the value of $P \geq 100$, and display the value of the last k .

```
k=0;  
sum=0;  
p=1;  
while p < 100  
    sum = sum + (k/(2*k+1))  
    p=4*sum  
    k=k+1;  
end  
disp(k)
```



➤ Switch and Case

The switch/case structure is often used when a series of programming path options exists for a given variable, depending on its value. The switch/case is similar to the if/else/elseif. As a matter of fact, anything you can do with switch/case could be done with if/else/elseif. However, the code is a bit easier to read with switch/case, a structure that allows you to choose between multiple outcomes, based on some criterion. This is an important distinction between switch/case and elseif. The criterion can be either a scalar (a number) or a string. In practice, it is used more with strings than with numbers. The structure of switch/case is

```
switch variable
case option1
    code to be executed if variable is equal to option 1
case option2
    code to be executed if variable is equal to option 2
.
.
.
case option_n
    code to be executed if variable is equal to option n
otherwise
    code to be executed if variable is not equal to any of the options
end
```

Here's an example: Suppose you want to create a function that tells the user what the airfare is to one of three different cities:

```
city = input('Enter the name of a city in single quotes: ')
switch city
case 'Basra'
    disp('$120')
case 'Irbil'
    disp('$150')
case 'Mousel'
    disp('$110')
otherwise
    disp('Not on file')
end
```

```
Enter the name of a city in single quotes: 'Basra'
city =
    'Basra'
$120
```



Example:

Create a program to prompt the user to enter the number of candy bars he or she would like to buy. The input will be a number. Use the switch/case structure to determine the bill, where

1 bar _ \$0.75
2 bars _ \$1.25
3 bars _ \$1.65
more than 3 bars= \$1.65 +\$0.30 (number ordered - 3)

Answer

```
num= input('How many candy bars would you like? ');  
switch num  
case 1  
    bill = 0.75;  
case 2  
    bill = 1.25;  
case 3  
    bill = 1.65;  
otherwise  
    bill = 1.65 + (num-3)*0.30;  
end  
fprintf('Your bill is %5.2f \n',bill)
```



Lecture Eight

Plotting in MATLAB



➤ Introduction

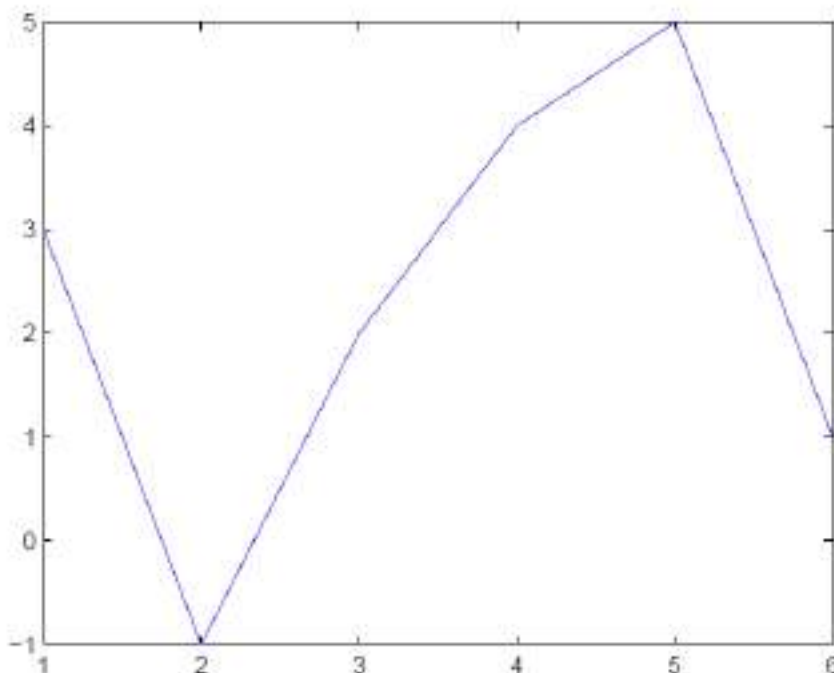
MATLAB has an excellent set of graphic tools. Plotting a given data set or the results of computation is possible with very few commands. You are highly encouraged to plot mathematical functions and results of analysis as often as possible. Trying to understand mathematical equations with graphics is an enjoyable and very efficient way of learning mathematics. Being able to plot mathematical functions and data freely is the most important step, and this section is written to assist you to do just that. Engineers use graphing techniques to make the information easier to understand. With a graph, it is easy to identify trends, pick out highs and lows, and isolate data points that may be measurement or calculation errors. Graphs can also be used as a quick check to determine whether a computer solution is yielding expected results.

➤ Basic Plotting

The basic MATLAB graphing procedure is to take a vector of x -coordinates, $x = (x_1; : : : x_N)$, and a vector of y -coordinates, $y = (y_1; : : : y_N)$, locate the points $(x_i; y_i)$, with $i = 1; 2; : : : n$ and then join them by straight lines. You need to prepare x and y in an identical array form; namely, x and y are both row arrays or column arrays of the *same* length.

The MATLAB command to plot a graph is `plot(x,y)`. The vectors $x = (1; 2; 3; 4; 5; 6)$ and $y = (3; -1; 2; 4; 5; 1)$ produce the picture shown in below.

```
>> x = [1 2 3 4 5 6];  
>> y = [3 -1 2 4 5 1];  
>> plot (x,y)
```





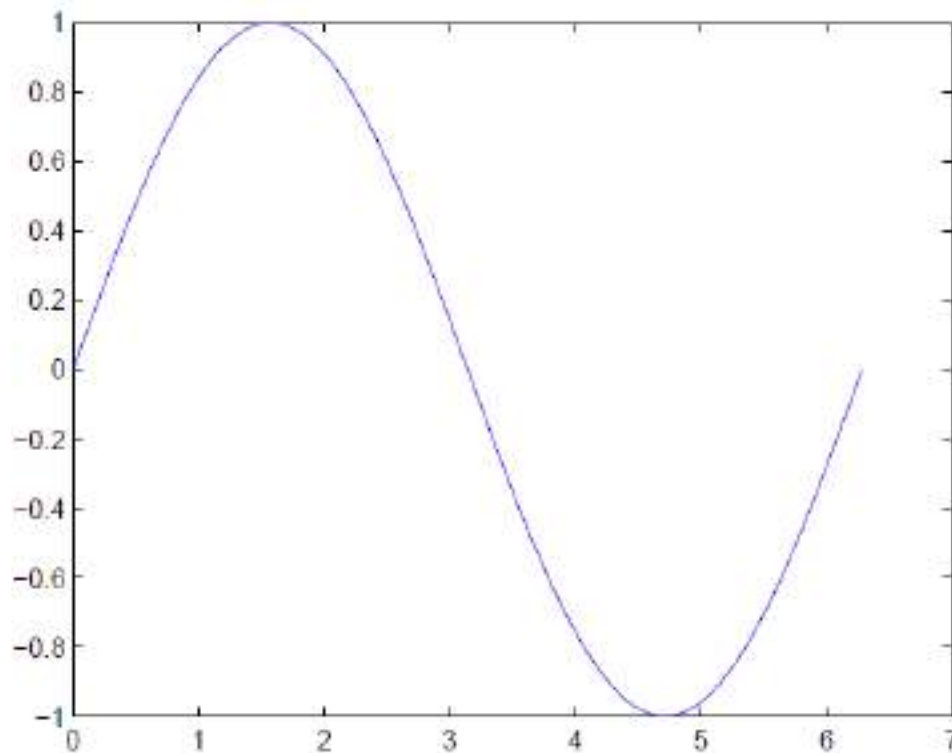
Note: The plot functions has different forms depending on the input arguments. If y is a vector plot (y) produces a piecewise linear graph of the elements of y versus the index of the elements of y . If we specify two vectors, as mentioned above, plot (x,y) produces a graph of y versus x .

For example, to plot the function $\sin(x)$ on the interval $[0, 2\pi]$, we first create a vector of x values ranging from 0 to 2π , then compute the *sine* of these values, and finally plot the result:

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

Notes:

- $0:\pi/100:2\pi$ yields a vector that
- starts at 0,
- takes steps (or increments) of $\pi/100$,
- stops when 2π is reached.
- If you omit the increment, MATLAB automatically increments by 1.





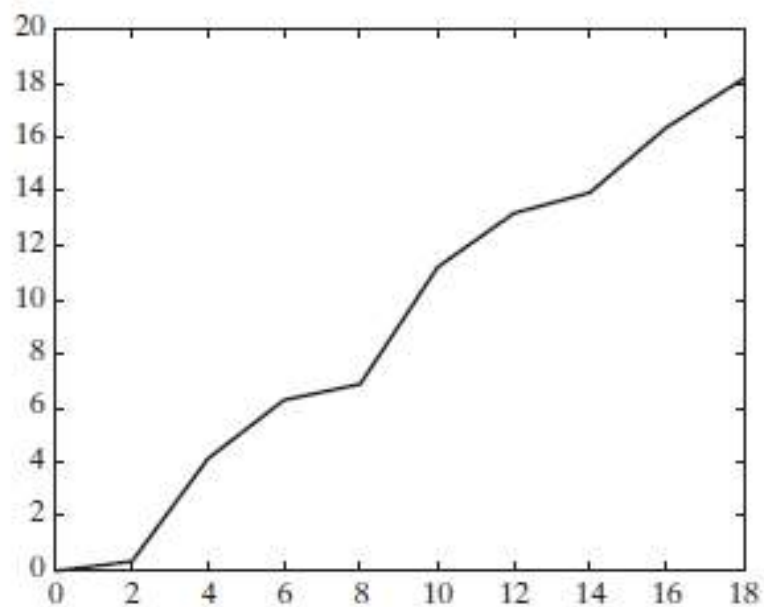
Suppose a set of time versus distance data were obtained through measurement. We can store the time values in a vector called **x** (the user can define any convenient name) and the distance values in a vector called **y** :

Time, s	Distance, ft
0	0
2	0.33
4	4.13
6	6.29
8	6.85
10	11.19
12	13.19
14	13.96
16	16.33
18	18.17

```
x = [0:2:18];
```

```
y = [0, 0.33, 4.13, 6.29, 6.85, 11.19, 13.19, 13.96, 16.33, 18.17];
```

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





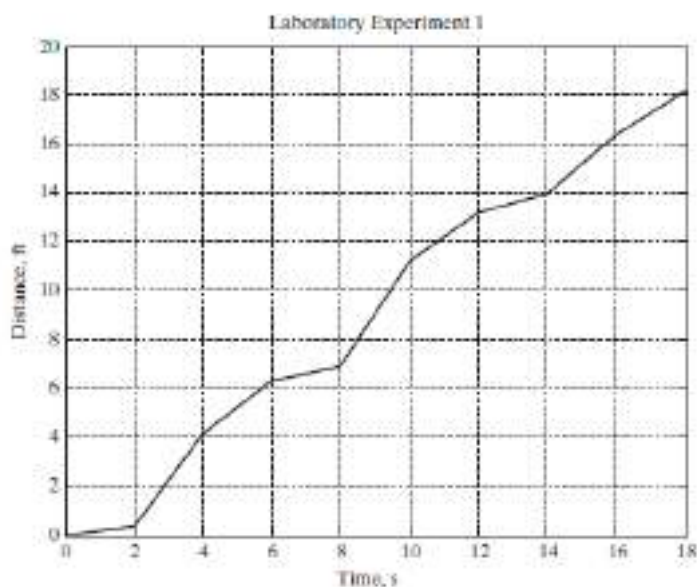
➤ Adding titles, axis labels, and annotations

Good engineering practice requires that we include axis labels and a title in our plot. The following commands add a title, x - and y -axis labels, and a background grid:

```
plot(x,y)
xlabel('Time, sec')
ylabel('Distance, ft')
grid on
```

These commands generate the plot in Figure below. As with any MATLAB commands, they could also be combined onto one or two lines, separated by commas:

```
plot(x,y) , title('Laboratory Experiment 1')
xlabel('Time, sec' ), ylabel('Distance, ft'), grid
```





➤ Specifying line styles and colors

By default, MATLAB uses *line style* and *color* to distinguish the data sets plotted in the graph. However, you can change the appearance of these graphic components or add annotations to the graph to help explain your data for presentation.

It is possible to specify *line styles*, *colors*, and *markers* (e.g., circles, plus signs, . . .) using the plot command:

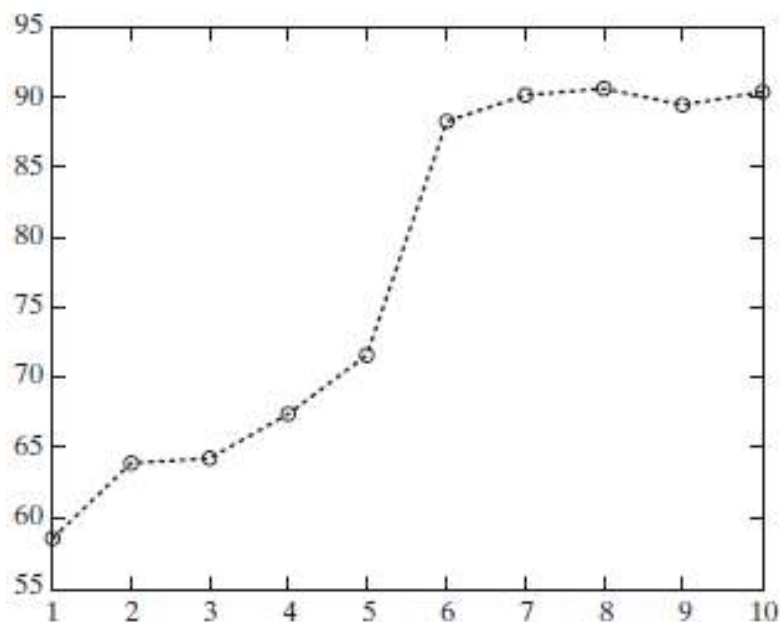
Plot (x,y,'style_color_marker')

where style_color_marker is a *triplet* of values from Table below:

SYMBOL	COLOR	SYMBOL	LINE STYLE	SYMBOL	MARKER
k	Black	—	Solid	+	Plus sign
r	Red	--	Dashed	o	Circle
b	Blue	:	Dotted	*	Asterisk
g	Green	-.	Dash-dot	.	Point
c	Cyan	none	No line	×	Cross
m	Magenta			s	Square
y	Yellow			d	Diamond

The following commands illustrate the use of line, color, and mark styles:

```
x = [1:10];
y = [58.5, 63.8, 64.2, 67.3, 71.5, 88.3, 90.1, 90.6, 89.5, 90.4];
plot(x,y,'ok')
```

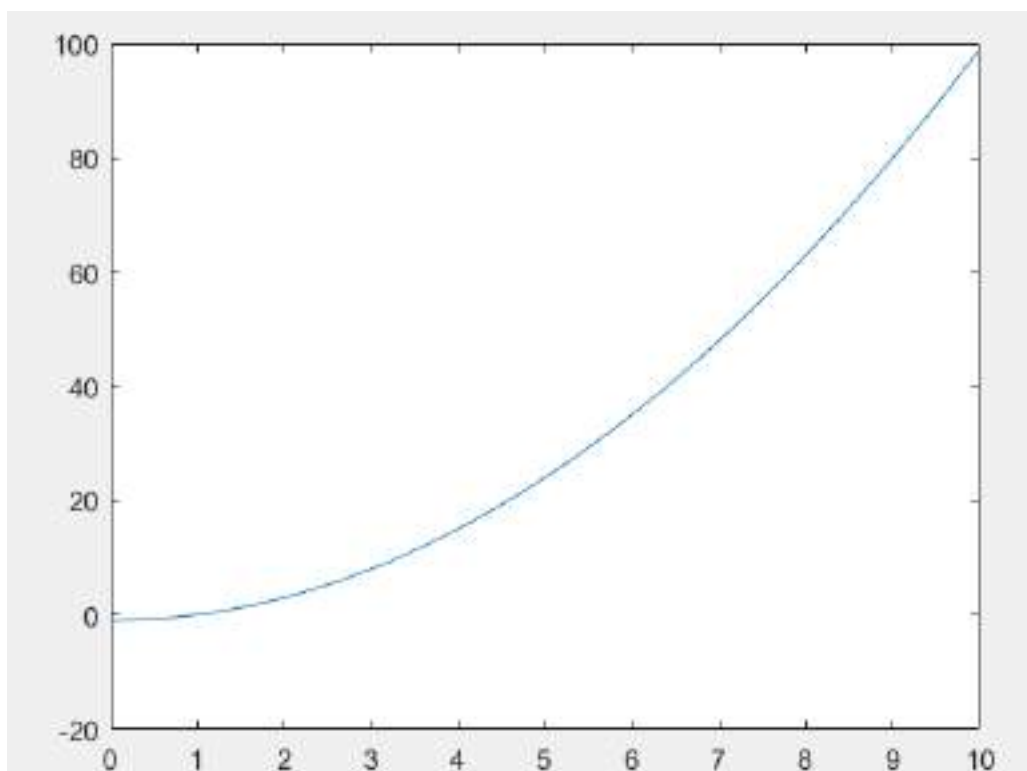




Example:

Plot the polynomial $y = x^2 - 1$ between $x=0$ and $x=10$ (using twenty points).

```
>>x=linspace(0,10,20); // or x=0:10/20:10;  
>>y= x.^2-1;  
>>plot(x,y)
```





Lecture Nine

Plotting in MATLAB

➤ SUBPLOTS



➤ SUBPLOTS

You can use the **subplot** command to obtain several smaller “subplots” in the same figure. The subplot command allows you to subdivide the graphing window into a grid of m rows and n columns. The function

subplot(m,n,p)

splits the figure into an $m \times n$ matrix. The variable **p** identifies the portion of the window where the next plot will be drawn. For example, if the command

subplot(2,2,1)

is used, the window is divided into two rows and two columns, and the plot is drawn in the upper left-hand window (Figure below)

p = 1	p = 2
p = 3	p = 4

The windows are numbered from left to right, top to bottom. Similarly, the following commands split the graph window into a top plot and a bottom plot.

subplot(3,2,5)

creates an array of six panes, three rows and two columns, and directs the next plot to appear in the fifth pane (in the bottom left corner).

p = 1	p = 2
p = 3	p = 4
p = 5	p = 6

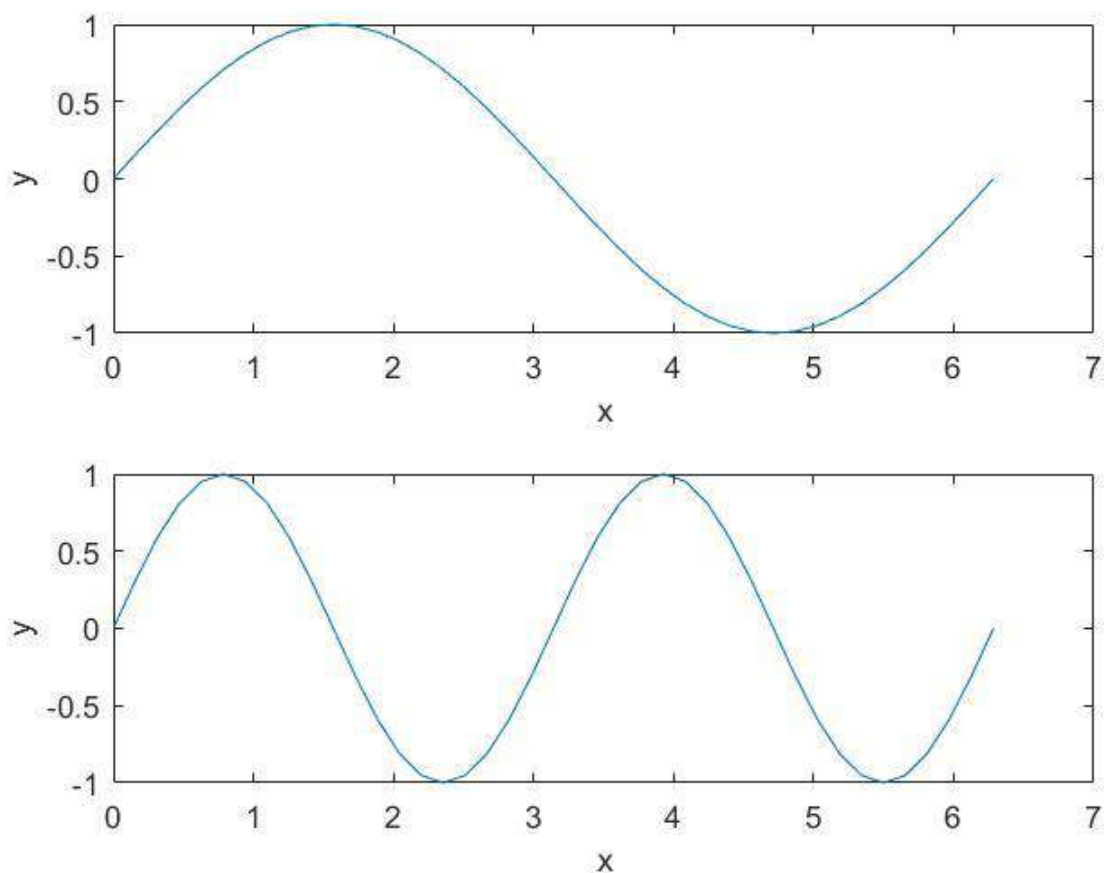


Example 1:

- Subdivide a figure window into two rows and one column.
- In the top window, plot $y = \sin(x)$ for x from 0 to 2π with increment of $\pi/20$.
- In the bottom window, plot $y = \sin(2x)$ for the same range.

```
x = 0:pi/20:2*pi;  
subplot(2,1,1)  
plot(x,sin(x))  
xlabel('x'),ylabel('y')  
subplot(2,1,2)  
plot(x,sin(2*x))  
xlabel('x'),ylabel('y')
```

The first graph is drawn in the top window, since $p=1$. Then the subplot command is used again to draw the next graph in the bottom window.





Example 2:

Use the subplot command to plot the functions (use an increment of 0.01):

$$y = e^{-1.2x} \sin(10x + 1) \quad \text{for } 0 \leq x \leq 5$$

$$y = |x^3 - 100| \quad \text{for } -6 \leq x \leq 6$$

```
x = 0:0.01:5;  
y = exp(-1.2*x).*sin(10*x+5);  
subplot(1,2,1)  
plot(x,y)  
xlabel('x'),ylabel('y')  
x = -6:0.01:6;  
y = abs(x.^3-100);  
subplot(1,2,2)  
plot(x,y)  
xlabel('x'),ylabel('y')
```

