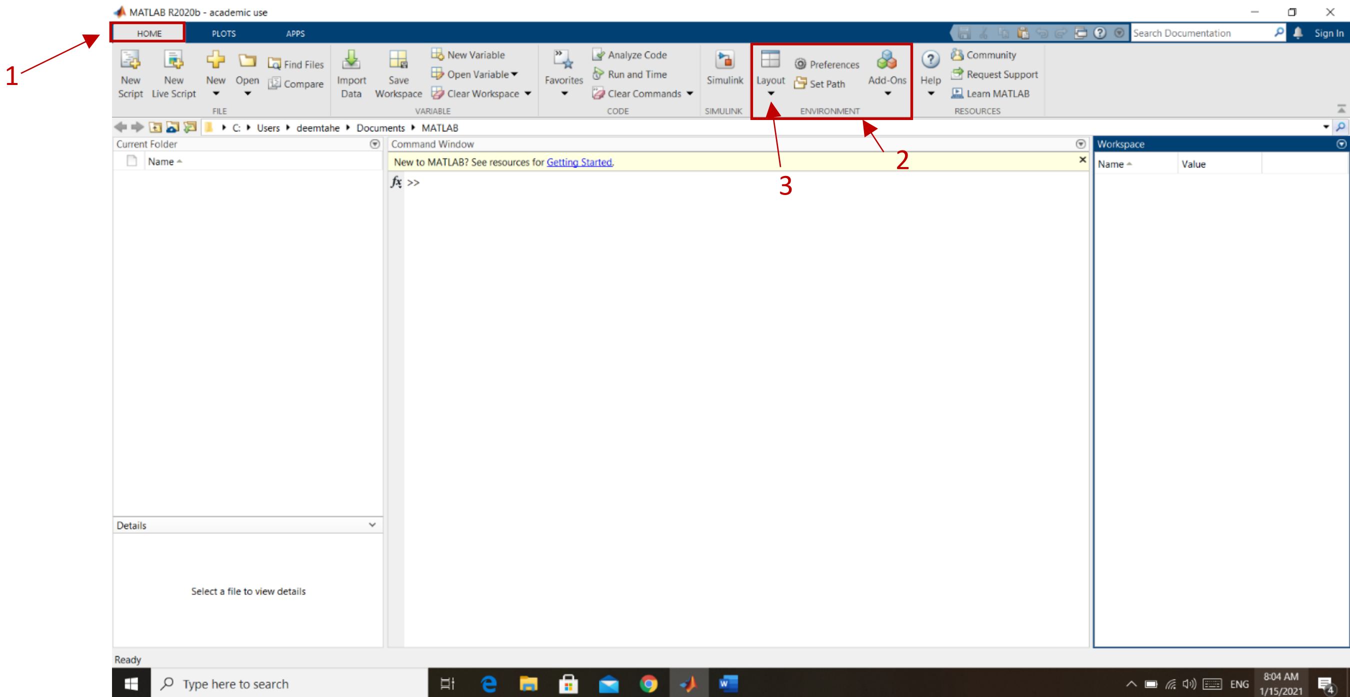
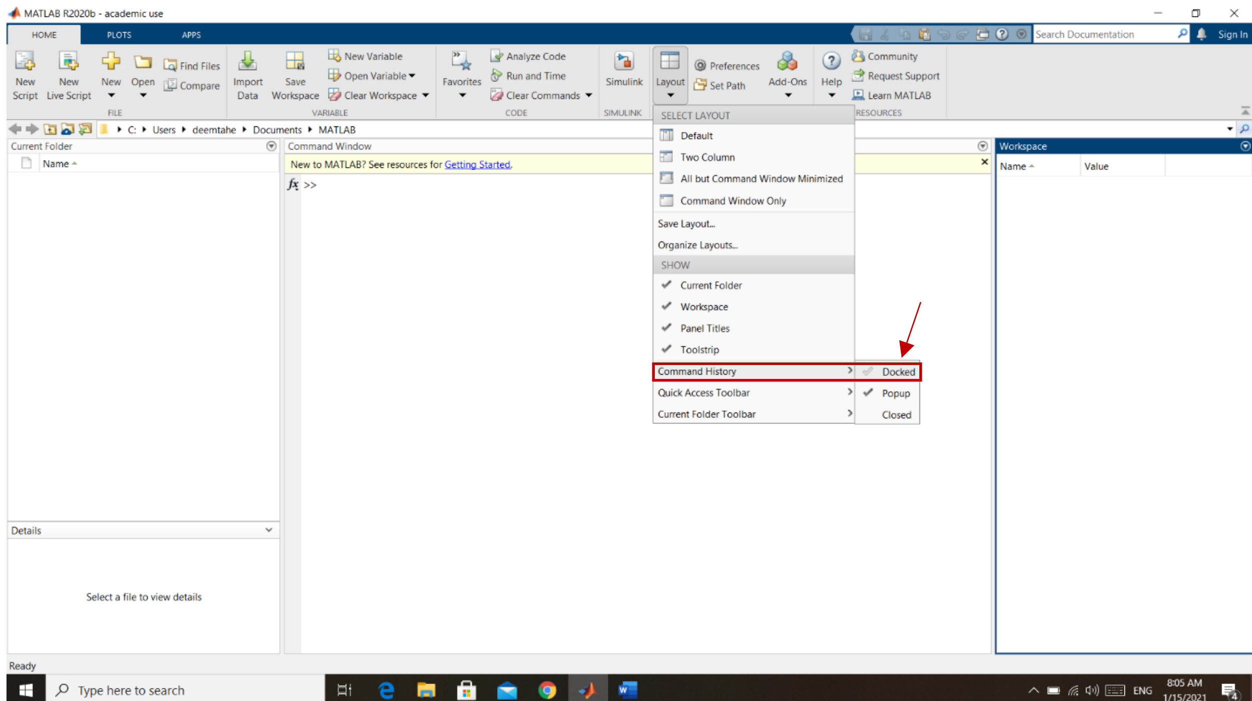
Session 1

After installation MATLAB:

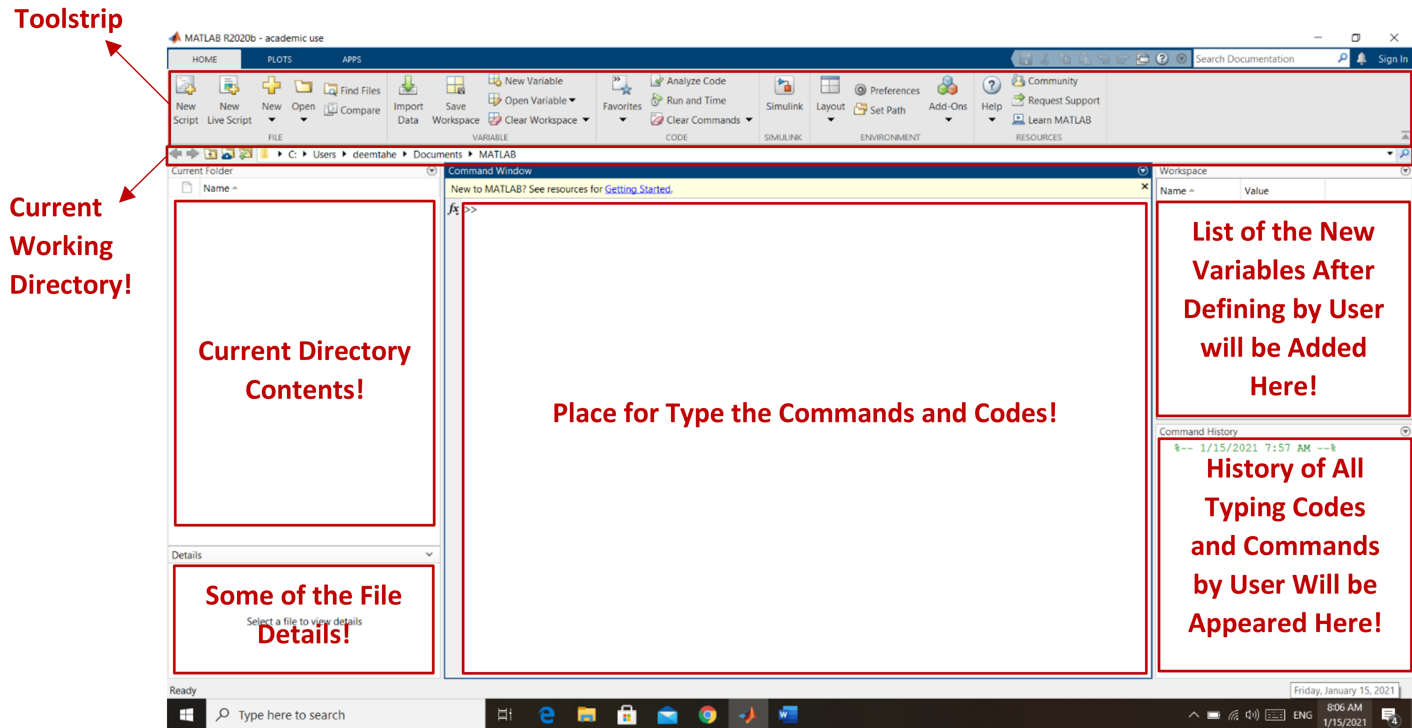
1) In the home page, Environment section, click on Layout icon.



2) Click on Command history and select Docked to become add to the windows.



The MATLAB Work Environment:



• What “MATLAB” stands for?

Matrix Laboratory: a programming environment built to handle matrices (arrays) and their arithmetic operations easily.

• What a MATLAB variable is: a named container for storing data.

• What types? Real (numeric) data, and less often, string data.

Fundamental MATLAB Classes for Data:

1) **Numeric Types**

Floating-Point Numbers: x = 325.499, y= -12.06, z=0.001

Integers: a = 3, c = 38, j = -11

2) **Characters and Strings**

Character arrays: A character array is a sequence of characters, just as a numeric array is a sequence of numbers. A typical use is to store short pieces of text as *character vectors*, such as c = 'Hello World'.

String arrays: A string array is a container for pieces of text. String arrays provide a set of functions for working with text as data. Starting in R2017a, you can create strings using double quotes, such as str = "Greetings friend". To convert data to string arrays, use the string function.

Variable (and script file) naming rules:

1. names must begin with an "alpha" character: a—z or A—Z
2. subsequent char's can be "alpha-numeric" or underscore; a—z, A—Z, or _
3. Must avoid keywords: end, while, if, for, return, else, function, ..., etc. These show up in the command window or editor as blue.
4. Suggested: avoid names or pre-defined MATLAB functions or constants: sin, cos, pi, ..., etc.

How to assign variables in the command window (or within script files):

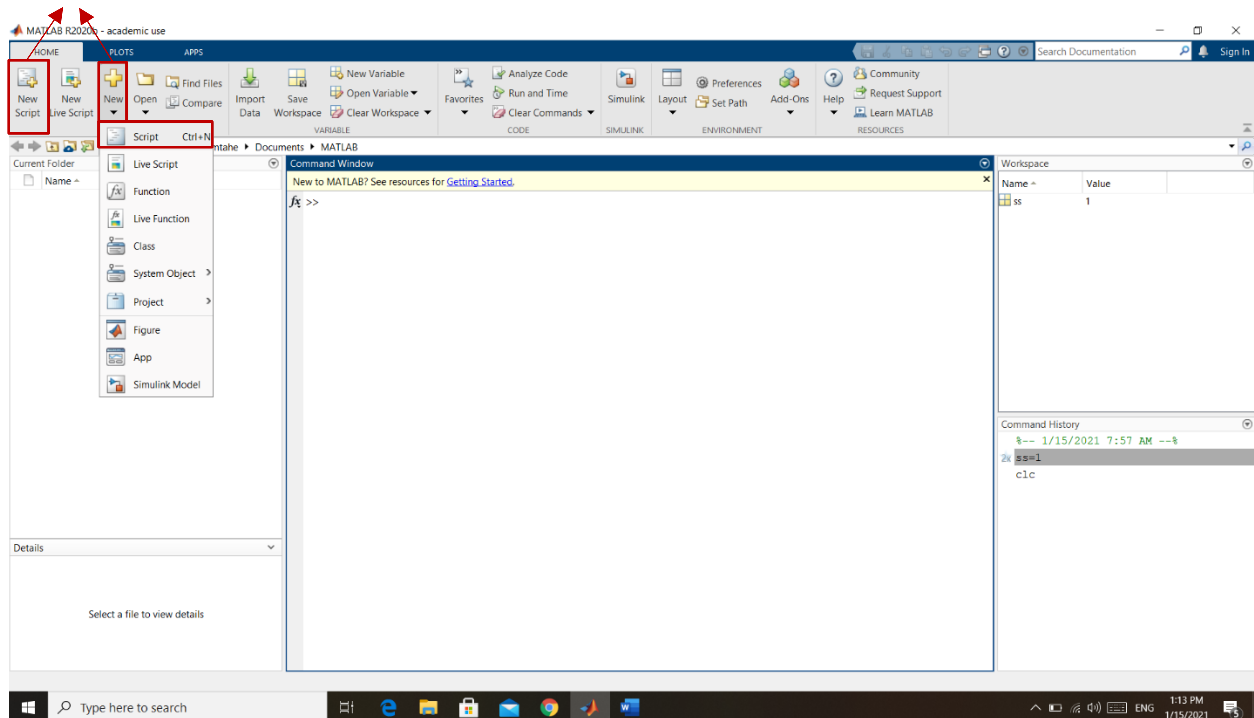
Example:
A = 42;
In this example:

- The "=" sign: "A gets the value 42", as opposed to "A equals 42".
- The terminating "**;** ": suppress the output of the operation.

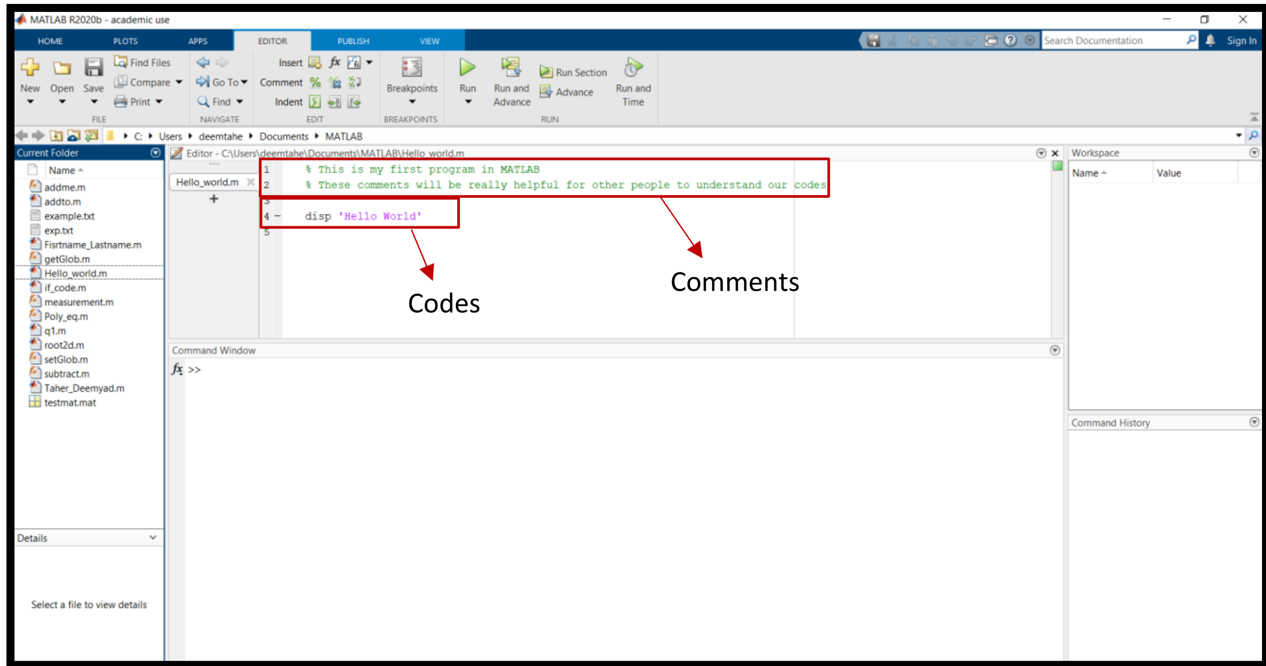
Script files:

How to open it:

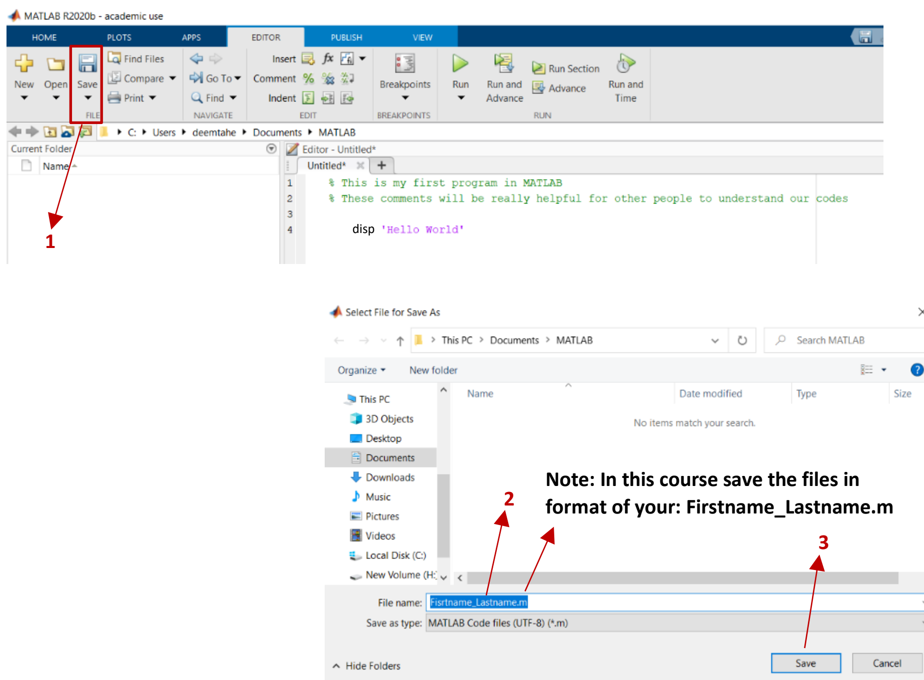
One of these two ways



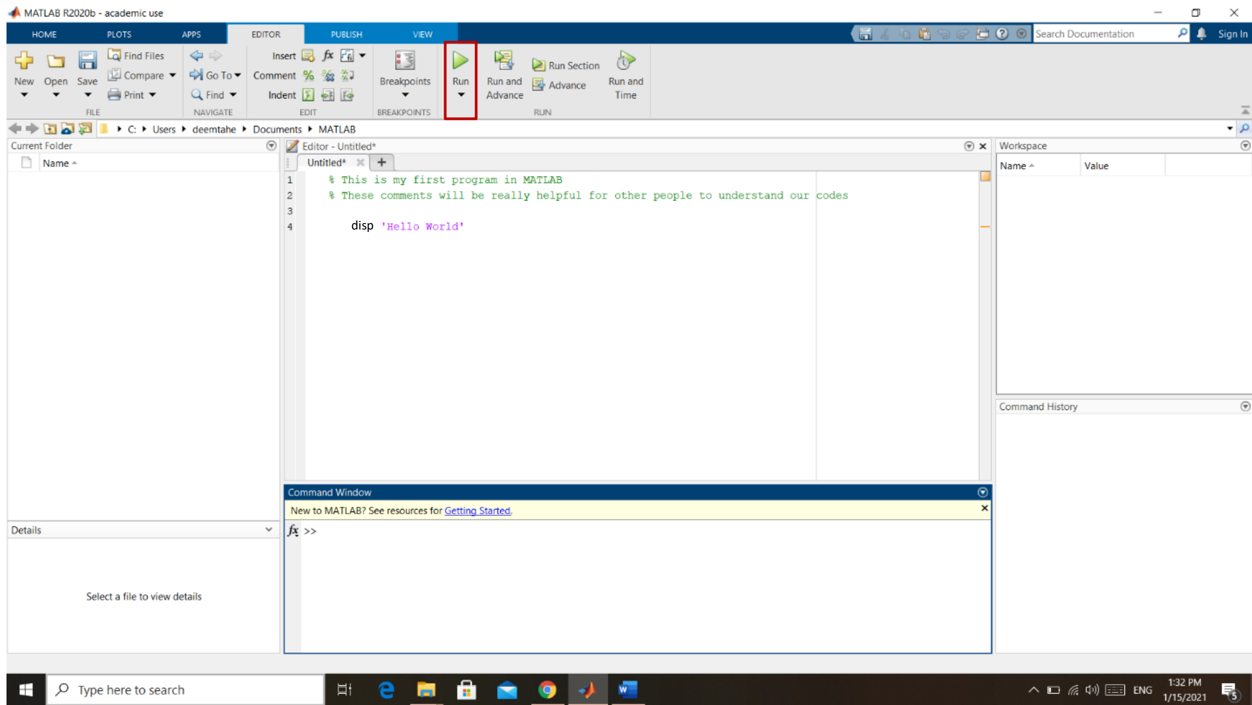
• Comments within a script file: they begin with %...stuff for the user to read.....
and continue to the end of the line. MATLAB ignores the material following %.....



How to save it:



How to execute it:



Session 2

List of Index: MATLAB Commands (Make sure never use any of them for assigning a variable!)

Command	Description
abs	Absolute value
acker	Compute the K matrix to place the poles of A-BK, see also place
axis	Set the scale of the current plot, see also plot, figure
bode	Draw the Bode plot, see also logspace, margin, nyquist1
c2d	Continuous system to discrete system
clf	Clear figure
conv	Convolution (useful for multiplying polynomials), see also deconv
ctrb	The controllability matrix, see also obsv
deconv	Deconvolution and polynomial division, see also conv
det	Find the determinant of a matrix
dlqr	Linear-quadratic regulator design for discrete-time systems, see also lqr
eig	Compute the eigenvalues of a matrix
eps	MATLAB's numerical tolerance
feedback	Connect linear systems in a feedback loop
figure	Create a new figure or redefine the current figure, see also subplot, axis
for	For loop
format	Number format (significant digits, exponents)
function	Creates function m-files
grid	Draw the grid lines on the current plot
gtext	Add a piece of text to the current plot, see also text
help	Matlab help documentation
hold	Hold the current graph, see also figure
if	Conditionally execute statements

imag	Returns the imaginary part of a complex number, see also real
impulse	Impulse response of linear systems, see also step, lsim
input	Prompt for user input
inv	Find the inverse of a matrix
legend	Graph legend
length	Length of a vector, see also size
linspace	Returns a linearly spaced vector
lnyquist	Produce a Nyquist plot on a logarithmic scale, see also nyquist1
log	Natural logarithm, also log10: common logarithm
loglog	Plot using log-log scale, also semilogx/semilogy
logspace	Returns a logarithmically spaced vector
lqr	Linear quadratic regulator design for continuous systems, see also dlqr
lsim	Simulate a linear system, see also step, impulse
margin	Returns the gain margin, phase margin, and crossover frequencies, see also bode
minreal	Produces a minimal realization of a system (forces pole/zero cancellations)
norm	Norm of a vector
nyquist1	Draw the Nyquist plot, see also lnyquist. Note this command was written to replace the MATLAB standard command nyquist to get more accurate Nyquist plots.
obsv	The observability matrix, see also ctrb
ones	Returns a vector or matrix of ones, see also zeros
place	Compute the K matrix to place the poles of A-BK, see also acker
plot	Draw a plot, see also figure, axis, subplot.
poly	Returns the characteristic polynomial
polyval	Polynomial evaluation
print	Print the current plot (to a printer or postscript file)

pzmap	Pole-zero map of linear systems
rank	Find the number of linearly independent rows or columns of a matrix
real	Returns the real part of a complex number, see also imag
rlocfind	Find the value of k and the poles at the selected point
rlocus	Draw the root locus
roots	Find the roots of a polynomial
rscale	Find the scale factor for a full-state feedback system
set	Set(gca,'Xtick',xticks,'Ytick',yticks) to control the number and spacing of tick marks on the axes
sgrid	Generate grid lines of constant damping ratio (zeta) and natural frequency (Wn), see also sigrid, zgrid
size	Gives the dimension of a vector or matrix, see also length
sqrt	Square root
ss	Create state-space models or convert LTI model to state space, see also tf
ssdata	Access to state-space data. See also tfdata
stairs	Stairstep plot for discrete response
step	Plot the step response, see also impulse, lsim
subplot	Divide the plot window up into pieces, see also plot, figure
text	Add a piece of text to the current plot, see also title, xlabel, ylabel, gtext
tf	Creation of transfer functions or conversion to transfer function, see also ss
tfdata	Access to transfer function data, see also ssdata
title	Add a title to the current plot
wbw	Returns the bandwidth frequency given the damping ratio and the rise or settling time.
xlabel/ylabel	Add a label to the horizontal/vertical axis of the current plot, see also title, text, gtext
zeros	Returns a vector or matrix of zeros
zgrid	Generates grid lines of constant damping ratio (zeta) and natural frequency (Wn), see also sgrid, sigrid

Arrays: Containers with multiple elements

Matrix: Matrices (m x n) arrays of values

$$x = \begin{bmatrix} a_{11} & \cdots & a_{1n} \\ \vdots & \ddots & \vdots \\ a_{m1} & \cdots & a_{mn} \end{bmatrix}_{m \times n} \qquad A = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}_{2 \times 3}$$

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

A =

     1     2     3
     4     5     6

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

A =

     1     2     3
     4     5     6
```

Row Vectors: Row vectors (1 x n) arrays of values

$$\mathbf{B} = [\mathbf{1} \quad \mathbf{4} \quad \mathbf{9}]_{1 \times 3}$$

```
>> B=[1 4 9]

B =

     1     4     9

>> B=[1, 4, 9]

B =

     1     4     9
```

- **We can use comma or space between the numbers for making a row vector.**

Column Vectors: Column vectors (m x 1) arrays of values

$$C = \begin{bmatrix} 6 \\ 4 \\ 9 \\ 3 \end{bmatrix}_{4\times 1}$$

```
>> C=[6;4;9;3]

C =

     6
     4
     9
     3
```

- **We must use semicolon (;) between the numbers for making a column vector.**
- **Using semicolon (;) after any comment to avoid showing the result after running that line.**

- **Changing a column vector to row vector or reverse with using apostrophe (')**

```
>> B=[1 4 9]

B =

     1     4     9
```

```
>> B'

ans =

     1
     4
     9
```

```
>> C=[6;4;9;3]

C =

     6
     4
     9
     3
```

```
>> C'

ans =

     6     4     9     3
```

Indexing: if a variable is a vector, just one index is needed. If a variable is a matrix, then the convention is (row index, column index), consistent with linear algebra.

MATALB's array creation operators and functions:

x = a: incr : b; % create a row vector running from a to b with increment = incr

```
>> x=1:5:10

x =

     1     6
```

x = a: b; % create a row vector from a to b, but with default increment 1

```
>> x=1:10

x =

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

x = linspace (a, b, N); % create a row vector from a to b, with N equally spaced points

```
>> x=linspace(1,10,4)

x =

     1     4     7    10
```

x = linspace (a, b); % create a row vector from a to b, with 100 equally spaced points

```
>> x=linspace(1,100)

x =

  Columns 1 through 17

     1     2     3     4     5     6     7     8     9    10    11    12    13    14    15    16    17

  Columns 18 through 34

    18    19    20    21    22    23    24    25    26    27    28    29    30    31    32    33    34

  Columns 35 through 51

    35    36    37    38    39    40    41    42    43    44    45    46    47    48    49    50    51

  Columns 52 through 68

    52    53    54    55    56    57    58    59    60    61    62    63    64    65    66    67    68

  Columns 69 through 85

    69    70    71    72    73    74    75    76    77    78    79    80    81    82    83    84    85

  Columns 86 through 100

    86    87    88    89    90    91    92    93    94    95    96    97    98    99   100
```

A = [1 2 3; 3 4 5; 6 7 42]; % create an array by hand, typing in the elements row by row, with ; denoting "go to the next row".

Session 3

MATLAB's special structure array functions:

eye (n, m):

```
>> eye(2,10)

ans =

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

eye (n):

```
>> eye(8)

ans =

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

zeros (n, m):

```
>> zeros(2,3)

ans =

     0     0     0
     0     0     0
```

zeros (n):

```
>> zeros(3)

ans =

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

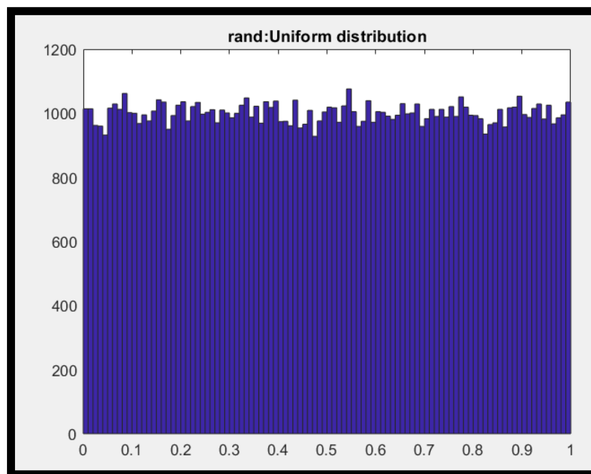
ones (n, m):

```
>> ones(4,3)

ans =

     1     1     1
     1     1     1
     1     1     1
     1     1     1
```

ones (n):

```
>> ones(4)

ans =

     1     1     1     1
     1     1     1     1
     1     1     1     1
     1     1     1     1
```

rand (n, m): creates uniform random numbers ("with replacement") in the range (0,1) exclusive.

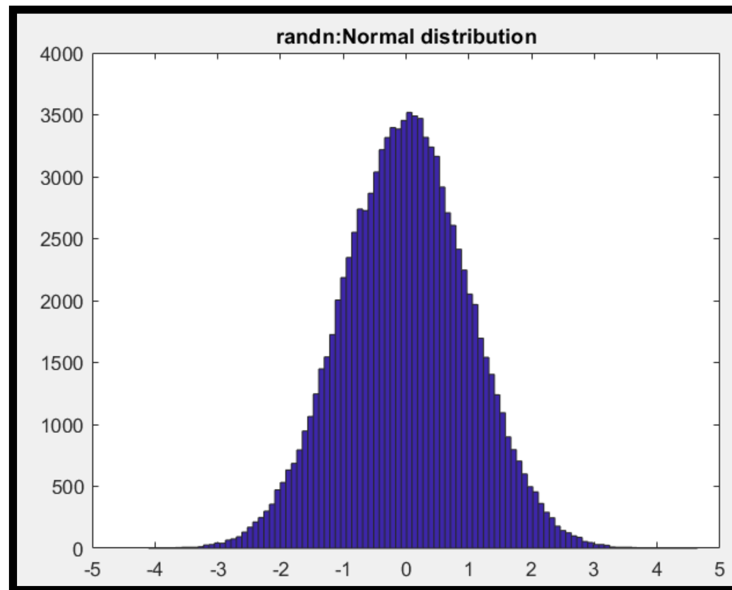



```
>> rand(2,7)

ans =

    0.5469    0.9649    0.9706    0.4854    0.1419    0.9157    0.9595
    0.9575    0.1576    0.9572    0.8003    0.4218    0.7922    0.6557
```

rand (n):

```
>> rand(2)

ans =

    0.0357    0.9340
    0.8491    0.6787
```

randn (n, m): creates normally distributed random numbers.




```
>> randn(2,7)

ans =

    1.0347   -0.3034   -0.7873   -1.1471   -0.8095    1.4384   -0.7549
    0.7269    0.2939    0.8884   -1.0689   -2.9443    0.3252    1.3703
```

randn (n):

```
>> randn(2)

ans =

   -1.7115   -0.2414
   -0.1022    0.3192
```

randi (imax, n): Uniformly distributed pseudorandom integers

```
>> randi(7,3)

ans =

     7     5     1
     5     1     7
     7     6     2
```

randi (imax, n, m):

```
>> randi(5,2,3)

ans =

     5     1     4
     5     5     1
```

diag (A):

```
>> A = [12 62 93 -8];
B = diag(A)

B =

    12     0     0     0
     0    62     0     0
     0     0    93     0
     0     0     0    -8
```

tril (A): Lower triangular part of matrix

L = tril(A) returns the lower triangular portion of matrix A.
L = tril(A,k) returns the elements on and below the kth diagonal of A.

```
>> A=ones(5)

A =

     1     1     1     1     1
     1     1     1     1     1
     1     1     1     1     1
     1     1     1     1     1
     1     1     1     1     1
```

```
>> tril(A)

ans =

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

```
>> tril(A,-1)

ans =

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

```
>> tril(A,1)

ans =

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

Specific matrix array:

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

A =

     2     3     3
     4     5     3

>> A(2,3)

ans =

     3
```

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

ans =

     5     3
```

Concatenating Matrices:

```
>> A = ones(1,4);
B = zeros(1,4);
C = [A B]

C =

     1     1     1     1     0     0     0     0
```

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

D =

     1     1     1     1
     0     0     0     0
```

Expanding a Matrix:

```
>> k = [10  20  30; 60  70  80]

k =

    10    20    30
    60    70    80

>> A(3,4) = 1

A =

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

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

A =

     1     1     1     1     0     0
     0     0     0     0     0     0
     0     0     0     1     0     0
     0     0     0     0     2     3
     0     0     0     0     4     5
```

Session 4

Math operators:
Elementwise array operations (vectorized) on arrays: The arithmetic operators are +, -, *, /, and ^, and their usage is consistent with linear algebra. They perform component wise operations in the case of +, - and (when used with an array and scalar) * and / With arrays, * and ^ are reserved for the linear algebra meaning (i.e., compatible matrix multiplication). If you want component wise operations, we need the "dot" operators: .*, ./, .^, for component-wise multiplication, division, and exponentiation. All MATLAB math functions are "vectorized".

Example:

```
>> a=[5 5 5 4]

a =

     5     5     5     4

>> b=[3 3 3 3]

b =

     3     3     3     3
```

+, -

```
>> a+b

ans =

     8     8     8     7

>> a-b

ans =

     2     2     2     1
```

/, \

```
>> a/b

ans =

    1.5833

>> a\b

ans =

    0.6000    0.6000    0.6000    0.6000
         0         0         0         0
         0         0         0         0
         0         0         0         0
```

* :

```
>> a*b
Error using  *
Incorrect dimensions for matrix multiplication. Check that the number of columns in the first matrix
matches the number of rows in the second matrix. To perform elementwise multiplication, use '.*'.
```

```
>> a*b'

ans =

    57
```

^ :

```
>> a^b
Error using  ^  (line 51)
Incorrect dimensions for raising a matrix to a power. Check that the matrix is square and the power is
a scalar. To perform elementwise matrix powers, use '.^'.
```

.^ :

```
>> a.^b

ans =

   125   125   125    64
```

./ :

```
>> a./b

ans =

    1.6667    1.6667    1.6667    1.3333
```

.* :

```
>> a.*b

ans =

    15    15    15    12
```

Order of priority of math operators in an equation:

( ) > ^ > *, / > +, -

```
>> 2+(3*4)-5^2/2

ans =

    1.5000
```

```
>> (2+3)*4-5^(2/2)

ans =

    15
```

MATLAB relational operators:

Symbol	Function Equivalent	Description
<	lt	Less than
<=	le	Less than or equal to
>	gt	Greater than
>=	ge	Greater than or equal to
==	eq	Equal to
~=	ne	Not equal to

MATLAB help function: help name of function

Example:

help plot
help mean
help sum
.
.
.

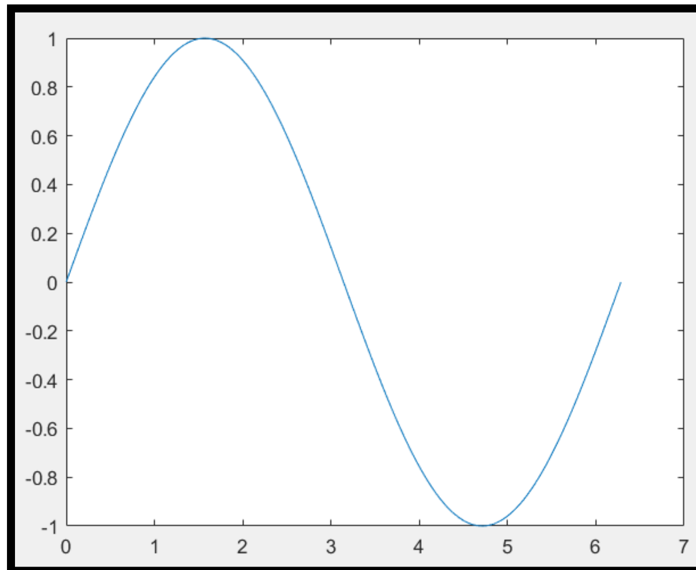
plot:

```
1       % This is my first program in MATLAB
2       % These comments will be really helpful for other people to understand our codes
3 –     close all
4 –     clear
5 –     clc
6 –     x = 0:pi/100:2*pi;
7 –     y = sin(x);
8 –     plot(x,y)
```

close all → Closes All Figures

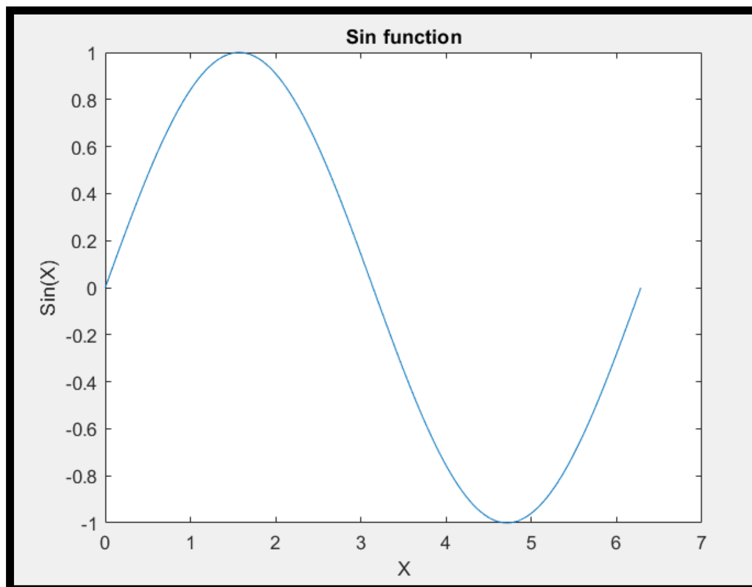
clear → Clear your workspace

clc → Clear Command Window



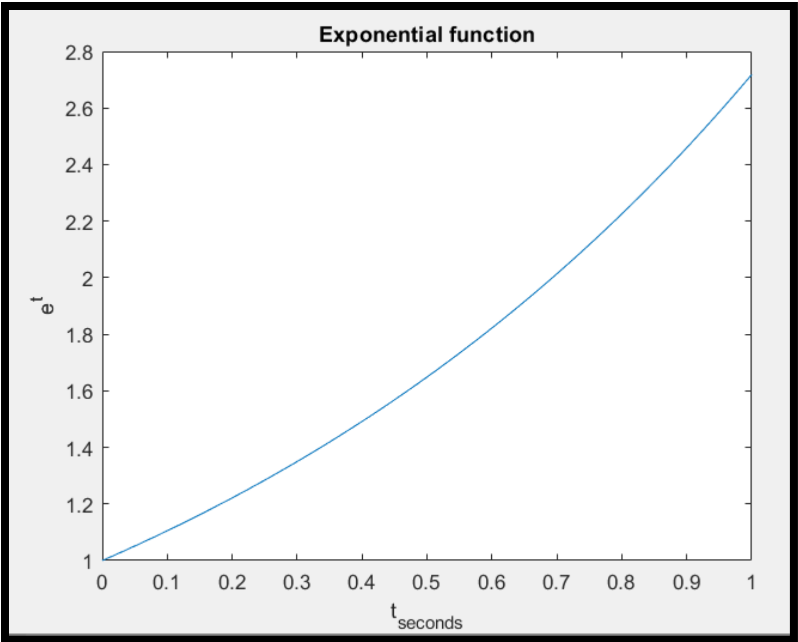
Label the plots:

```
1      % This is my first program in MATLAB
2      % These comments will be really helpful for other people to understand our codes
3 -    close all
4 -    clear
5 -    clc
6 -    x = 0:pi/100:2*pi;
7 -    y = sin(x);
8 -    plot(x,y)
9 -    xlabel('X')
10 -   ylabel('Sin(X)')
11 -   title('Sin function')
```



Use the '^' and '_' characters to include superscripts and subscripts in the axis labels. Use curly braces { } to modify more than one character.

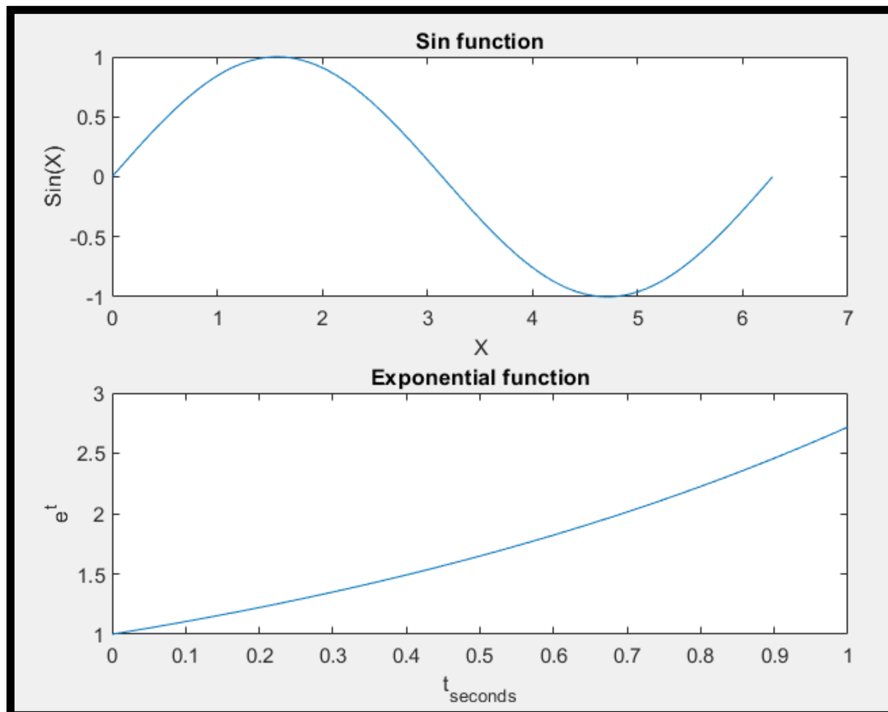
```
1      % This is my first program in MATLAB
2      % These comments will be really helpful for other people to understand our codes
3 -    close all
4 -    clear
5 -    clc
6 -    t = linspace(0,1);
7 -    y = exp(t);
8 -    plot(t,y)
9 -    xlabel('t_{seconds}')
10 -   ylabel('e^t')
11 -   title('Exponential function')
```


Exponential function
2.8
2.6
2.4
2.2
2
1.8
1.6
1.4
1.2
1
0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
1
e^t
t_seconds

Session 5

subplot: subplot Create axes in tiled positions. H = subplot(m,n,p), or subplot(mnp), breaks the Figure window into an m-by-n matrix of small axes, selects the p-th axes for the current plot, and returns the axes handle. The axes are counted along the top row of the Figure window, then the second row, etc.

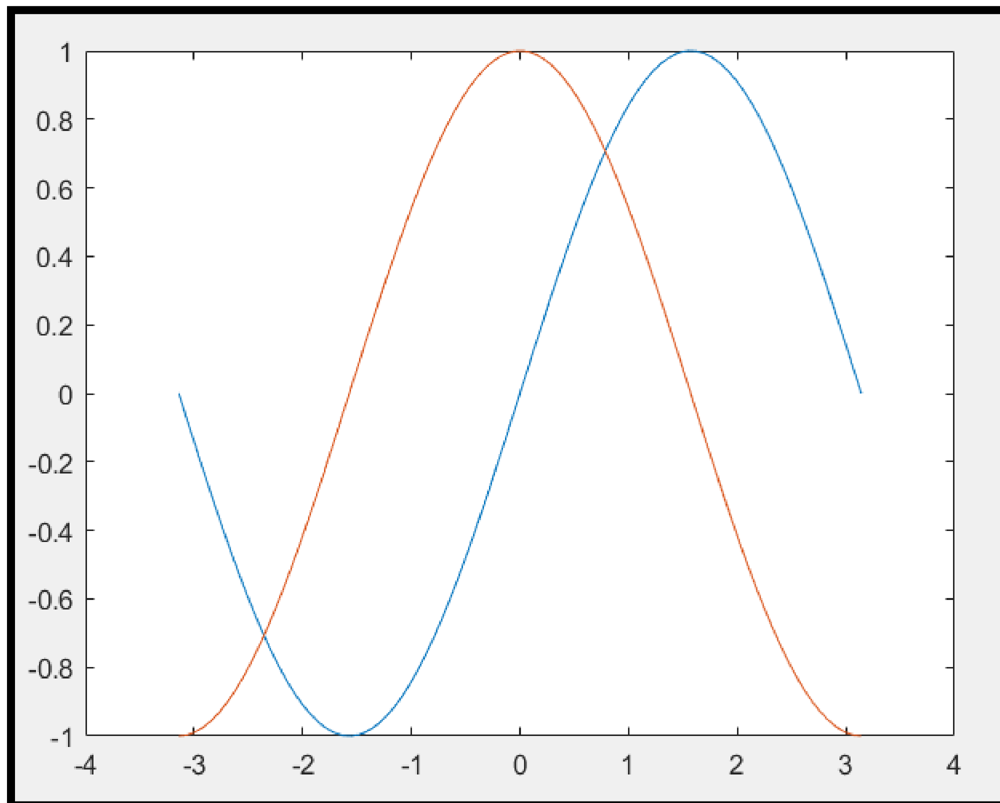
```
1      % This is my first program in MATLAB
2      % These comments will be really helpful for other people to understand our codes
3 -    close all
4 -    clear
5 -    clc
6
7 -    subplot(2,1,1);
8 -    x = 0:pi/100:2*pi;
9 -    y = sin(x);
10 -   plot(x,y)
11 -   xlabel('X')
12 -   ylabel('Sin(X)')
13 -   title('Sin function')
14
15 -   subplot(2,1,2);
16 -   t = linspace(0,1);
17 -   y = exp(t);
18 -   plot(t,y)
19 -   xlabel('t_{seconds}')
20 -   ylabel('e^t')
21 -   title('Exponential function')
```



hold on / hold off: Sometimes a plot needs to be amended, keeping currently displayed material on the plot. For this case we can use “hold on” and “hold off”!

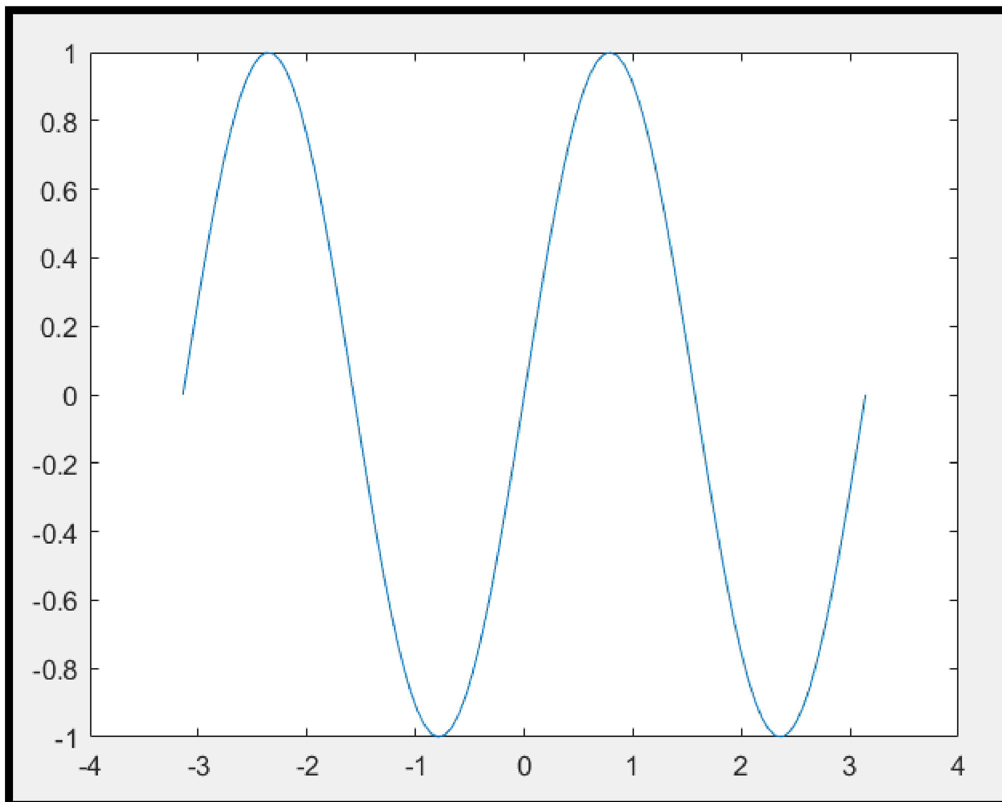
Example 1: Create a line plot. Use hold on to add a second line plot without deleting the existing line plot. The new plot uses the next color and line style based on the ColorOrder & LineStyleOrder properties of the axes. Then reset the hold state to off.

```
x = linspace(-pi,pi);
y1 = sin(x);
plot(x,y1)

hold on
y2 = cos(x);
plot(x,y2)
hold off
```



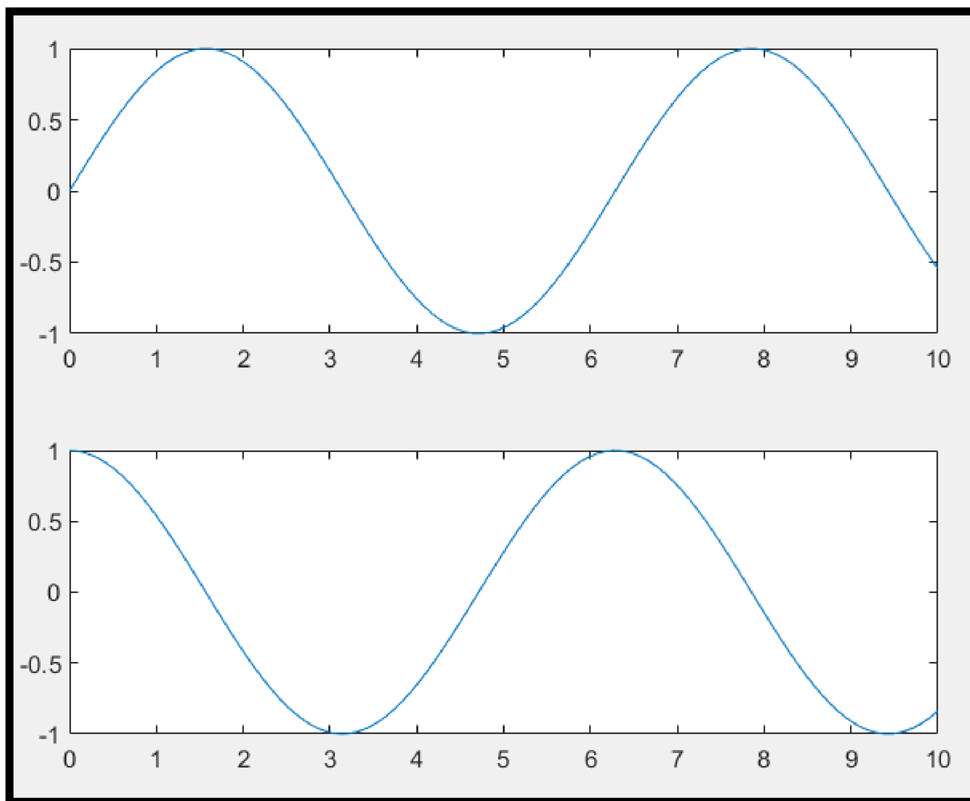
When the hold state is off, new plots delete existing plots. New plots start from the beginning of the color order and line style order.

```
x = linspace(-pi,pi);
y1 = sin(x);
plot(x,y1)

hold on
y2 = cos(x);
plot(x,y2)
hold off

y3 = sin(2*x);
plot(x,y3)
```



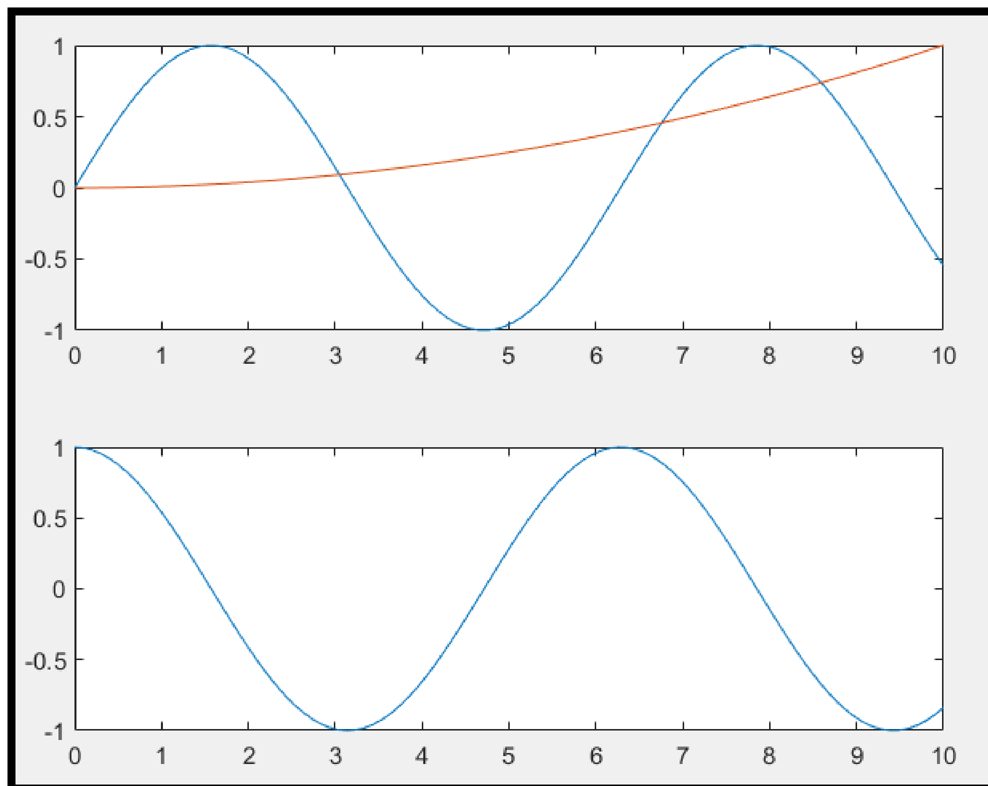
tiledlayout: you can display a tiling of plots using the tiledlayout and nexttile functions. Call the tiledlayout function to create a 2-by-1 tiled chart layout. Call the nexttile function to create the axes objects ax1 and ax2. Plot a sin & cos wave plot in each axis.

```
x = linspace(0,10);
y1 = sin(x);
y2 = cos(x);
tiledlayout(2,1)

% Top plot
ax1 = nexttile;
plot(ax1,x,y1)

% Bottom plot
ax2 = nexttile;
plot(ax2,x,y2)
```



We can add a second graph to any of these axes with using hold on/ hold off command.

```
x = linspace(0,10);
y1 = sin(x);
y2 = cos(x);
tiledlayout(2,1)

% Top plot
ax1 = nexttile;
plot(ax1,x,y1)

% Bottom plot
ax2 = nexttile;
plot(ax2,x,y2)

hold(ax1,'on')
y3 = (x.^2)/100;
plot(ax1,x,y3)
hold(ax1,'off')
```



Session 6

input:

Example 1: Receive length and width of a rectangle from user and calculate the area.

```
3 -    close all
4 -    clear
5 -    clc
6
7 -    i=input ('length of rectangle = ');
8 -    j=input ('width of rectangle= ');
9 -    Area=i*j
```

Example 2: Receive three inputs for grade of a student in midterm #1, midterm #2, and final exam. After that calculate the final grade for him/her in following way: %20 for the 1st midterm, %30 for 2nd midterm and %50 for final exam!

```
3 -    close all
4 -    clear
5 -    clc
6
7 -    N1=input ('Please Enter the 1st midterm exam grade = ');
8 -    N2=input ('Please Enter the 2nd midterm exam grade = ');
9 -    N3=input ('Please Enter the final exam grade = ');
10 -   Final_Grade=(0.2*N1)+(0.3*N2)+(0.5*N3)
```

- str = input (prompt, 's') returns the entered text, without evaluating the input as an expression.

disp: Display value of variable. disp(X) displays the value of variable X without printing the variable name. Another way to display a variable is to type its name, which displays a leading "X =" before the value.

Example 1:

```
A = [15 13; 12 4];
B = 'Hello everyone';
disp (A)
disp (B)

    15    13
    12     4

Hello everyone
```

Example 2:

```
close all
clear
clc

name = 'Please write your name: ';
n = input (name, 's');
age = input ('Please write your age: ');
X = [n,' will be ',num2str(age),' this year.'];
disp(X)
```

```
Please write your name: Sara
Please write your age: 21
Sara will be 21 this year.
```

Boolean expressions: True or false (Boolean) conditions

The logical data type represents true or false states using the numbers 1 and 0, respectively. Certain MATLAB® functions and operators return logical values to indicate fulfillment of a condition. You can use those logical values to index into an array or execute conditional code.

```
>> A=2

A =

     2

>> B=4

B =

     4
```

```
>> B>A

ans =

  logical

   1
```

B will be assigned new value.

```
>> B=A*3

B =

     6
```

B & the expression in front of it will be checked if is a correct assumption or not!

```
>> B==A*3

ans =

  logical

   0
```

if, else, elseif:
In this section we will explain the "single-sided if", the "two-sided if (if-else)", and the "cascaded if (if-elseif-else)" structures. All of these are using for the first "programming structures" to allow nonsequential code.

Single-sided if:

```
1 -    close all
2 -    clear
3 -    clc
4 -    Maximum_Weight=150;
5 -    A=input ('Please type your weight: ');
6 -    if     (A < Maximum_Weight)
7 -        disp('You are allowed to use this bicycle ')
8 -    end
```

Answer:

```
Please type your weight: 120
You are allowed to use this bicycle
```

Two-sided if (if-else):

```
 1 -    close all
 2 -    clear
 3 -    clc
 4 -    Maximum_Weight=150;
 5 -    A=input ('Please type your weight: ');
 6 -    if     (A < Maximum_Weight)
 7 -        disp('You are allowed to use this bicycle')
 8 -    else
 9 -        disp('You are too heavy to use this bicycle')
10 -    end
```

Answer:

```
Please type your weight: 132
You are allowed to use this bicycle
Please type your weight: 150
You are too heavy to use this bicycle
```

An expression can include relational operators (such as < or ==) and logical operators (such as &&, ||, or ~). Use the logical operators **and** and **or** to create compound expressions. Within the conditional expression of an if...end block, logical operators & and | behave as short-circuit operators. This behavior is the same as && and ||, respectively. Since && and || consistently short-circuit in conditional expressions and statements, it is good practice to use && and || instead of & and | within the expression.

```
1 -     close all
2 -     clear
3 -     clc
4
5 -     minVal = 21;
6 -     maxVal = 65;
7 -     m = 'Please type your Age: ';
8 -     x = input (m);
9
10 -    if (x <= minVal) || (x >= maxVal)
11 -        disp('Sorry, you are NOT eligible for this membership!')
12 -    else
13 -        disp('Congratulations! You are eligible for this membership.')
14 -    end
```

or: it means if any of these two conditions be satisfied, that would be a correct assumption!

Answer:

```
Please type your Age: 12
Sorry, you are NOT eligible for this membership!
Please type your Age: 70
Sorry, you are NOT eligible for this membership!
Please type your Age: 43
Congratulations! You are eligible for this membership.
```

cascaded if (if-elseif-else):

```
1 -     close all
2 -     clear
3 -     clc
4
5 -     x = input ('Please type your Age: ');
6 -     minVal = 21;
7 -     maxVal = 65;
8
9 -     if (x >= minVal) && (x <= maxVal)
10 -        disp('Congratulations! You are eligible for this membership.')
11 -    elseif (x > maxVal)
12 -        disp('Sorry, your age exceeds maximum age for this membership!')
13 -    else
14 -        disp('Sorry, your age is below minimum age for this membership!')
15 -    end
```

and: it means both conditions must satisfy!

Answer:

```
Please type your Age: 14
Sorry, your age is below minimum age for this membership!
Please type your Age: 34
Congratulations! You are eligible for this membership.
Please type your Age: 72
Sorry, your age exceeds maximum age for this membership!
```

error: Throw error and display message, **error (msg).**

```
1 -    close all
2 -    clear
3 -    clc
4
5 -    minVal = 21;
6 -    maxVal = 65;
7 -    m = 'Please type your Age: ';
8 -    x = input (m);
9 -    if  x<0
10 -       error('Error. The age must be a positive number!')
11 -   elseif (x >= minVal) && (x <= maxVal)
12 -       disp('Congratulations! You are eligible for this membership.')
13 -   else
14 -       disp('Sorry, you are NOT eligible for this membership!')
15 -   end
```

```
Please type your Age: -3
Error using if_code (line 10)
Error. The age must be a positive number!
```

- **If any error happened and you want to stop running program you can use (Ctrl+c)!**

Session 7

switch, case, otherwise: Execute one of several groups of statements.

Example 1: write a code to show different text conditionally, depending on a value entered at the command prompt.

```
1 -    close all
2 -    clear
3 -    clc
4
5 -    n = input('Enter a number: ');
6
7 -    switch n
8 -        case -1
9 -            disp('negative one')
10 -       case 0
11 -           disp('zero')
12 -       case 1
13 -           disp('positive one')
14 -       otherwise
15 -           disp('other value')
16 -   end
```

Results:

```
Enter a number: -1
negative one
Enter a number: 1
positive one
Enter a number: 0
zero
Enter a number: 5
other value
```

Example 2: write a code to show we are in which part of the day based on the time.

```
1 -     close all
2 -     clear
3 -     clc
4
5 -     n = input('Enter the time between (0-23): ');
6
7 -     switch n
8 -         case {7,8,9,10,11}
9
10 -            h=[num2str(n),' Am', ' (Morning)'];
11 -            disp(h)
12 -        case 12
13 -            h=[num2str(n) , ' Pm', ' (Noon)'];
14 -            disp(h)
15 -        case {13,14,15,16,17,18,19}
16 -            h=[num2str(n), ' Pm', ' Afternoon'];
17 -            disp(h)
18 -        case {20,21,22,23}
19 -            h=[num2str(n), ' Pm', ' (Night)'];
20 -            disp(h)
21 -        otherwise
22 -            h=[num2str(n), ' Am', ' (Night)'];
23 -            disp(h)
24 -    end
```

Results:

```
Enter the time between (0-23): 8
8 Am (Morning)
Enter the time between (0-23): 12
12 Pm (Noon)
Enter the time between (0-23): 16
16 Pm Afternoon
Enter the time between (0-23): 21
21 Pm (Night)
Enter the time between (0-23): 5
5 Am (Night)
```

while loop: while loop to repeat when **condition is true**.

Example 1: write a code to calculate the factorial for different numbers.

```
1 -    close all
2 -    clear
3 -    clc
4
5 -    n = input ('Please type the number you want to calculate the factorial for it: ');
6 -    f = n;
7 -    while n > 1
8 -        n = n-1;
9 -        f = f*n;
10 -   end
11 -   disp(['n! = ' num2str(f)])
```

Results:

```
Please type the number you want to calculate the factorial for it: 1
n! = 1
Please type the number you want to calculate the factorial for it: 2
n! = 2
Please type the number you want to calculate the factorial for it: 5
n! = 120
Please type the number you want to calculate the factorial for it: 10
n! = 3628800
```

Example 2: write a code to calculate the maximum number of people can using an elevator at the same time with a capacity of 800 lbs.

```
1 -    close all
2 -    clear
3 -    clc
4
5 -    limit = 800;
6 -    wsum = 0;
7 -    i = 0;
8
9 -    while wsum<limit
10 -       s = input(['Please type the weight of the person ', '#', num2str(i+1),': ']);
11 -       wsum = wsum + s;
12
13 -         if wsum > limit
14 -             wsum = wsum - s;
15 -           break
16 -         end
17
18 -       i = i+1;
19 -   end
20 -   disp(['Number of people: ', num2str(i), ', sum of their weight: ' ,num2str(wsum),'lbs '])
```

Results:

```
Please type the weight of the person #1: 100
Please type the weight of the person #2: 200
Please type the weight of the person #3: 220
Please type the weight of the person #4: 115
Please type the weight of the person #5: 130
Please type the weight of the person #6: 170
Number of people: 5, sum of their weight: 765lbs
```

Example 3: Sum a sequence of random numbers until the next random number is greater than an upper limit. Then, exit the loop using a break statement.

```
1 -     close all
2 -     clear
3 -     clc
4 -     limit = 0.8;
5 -     s = 0;
6
7 -     while 1
8 -         tmp = rand;
9 -         if tmp > limit
10 -            break
11 -        end
12 -        s = s + tmp;
13 -    end
14
15 -    disp(['Results= ' num2str(s)])
```

Results:

```
Results= 3.7155
```

Session 8

for: for loop to repeat specified number of times.

```
for index = values
    statements
end
```

Example 1: write a code to create a Hilbert matrix of order 10.

Hilbert matrix:

$$H_n = \begin{pmatrix} 1 & 1/2 & 1/3 & \cdots & 1/n \\ 1/2 & 1/3 & 1/4 & \cdots & 1/n+1 \\ 1/3 & 1/4 & 1/5 & \cdots & 1/n+2 \\ \vdots & \vdots & \vdots & & \vdots \\ 1/n & 1/n+1 & 1/n+2 & \cdots & 1/2n-1 \end{pmatrix}$$

```
1 -    close all
2 -    clear
3 -    clc
4
5 -    format rat
6 -    s = input('Please type the order for Hilbert matrix that you want to create: ');
7 -    H = zeros(s);
8
9 -    for c = 1:s
10 -       for r = 1:s
11 -           H(r,c) = 1/(r+c-1);
12 -       end
13 -   end
14
15 -   disp(H)
```

Rational fraction format

Results:

```
Please type the order for Hilbert matrix that you want to create: 5
        1              1/2            1/3            1/4            1/5
        1/2            1/3            1/4            1/5            1/6
        1/3            1/4            1/5            1/6            1/7
        1/4            1/5            1/6            1/7            1/8
        1/5            1/6            1/7            1/8            1/9
```

Example 2: Step by increments of `-0.2` and display the values.

```
1 -    close all
2 -    clear
3 -    clc
4
5 -    n = input('Please type the Maximum number: ');
6
7 -    for v = n:-0.2:0.0
8 -        disp(v)
9 -    end
```

Results:

```
Please type the Maximum number: 1
     1

    0.8000

    0.6000

    0.4000

    0.2000

     0
```

Example 3: Execute statements for specified values

```
1 -    close all
2 -    clear
3 -    clc
4
5 -    for x = [1 4 6 8 90]
6 -    disp(x)
7 -    end
```

Results:

```
     1

     4

     6

     8

    90
```

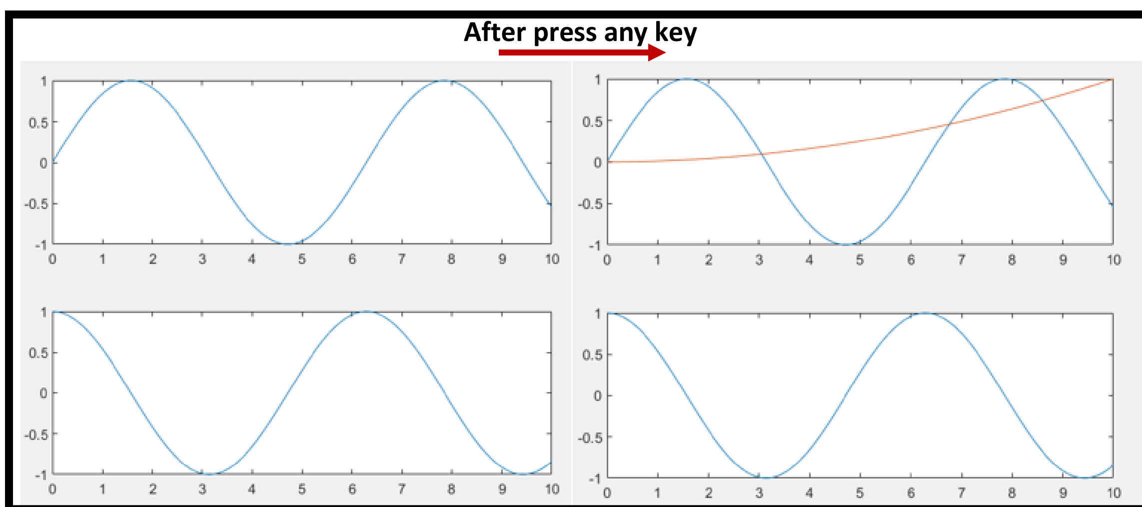
Note: A "for" loop can usually be avoided in MATLAB programming: if the computation being done can be vectorized, no need for the loop. Only when current computation depends on previously computed values does a for loop need to be used. If an array of results is being computed, make sure the array's memory is "pre-allocated", e.g., A = zeros(N,1) like the first example.

pause: We can use it in different form for temporarily stops MATLAB execution.

pause: temporarily stops MATLAB execution and waits for the user to press any key.
pause(n): pauses execution for n seconds before continuing. Pausing must be enabled for this call to take effect.

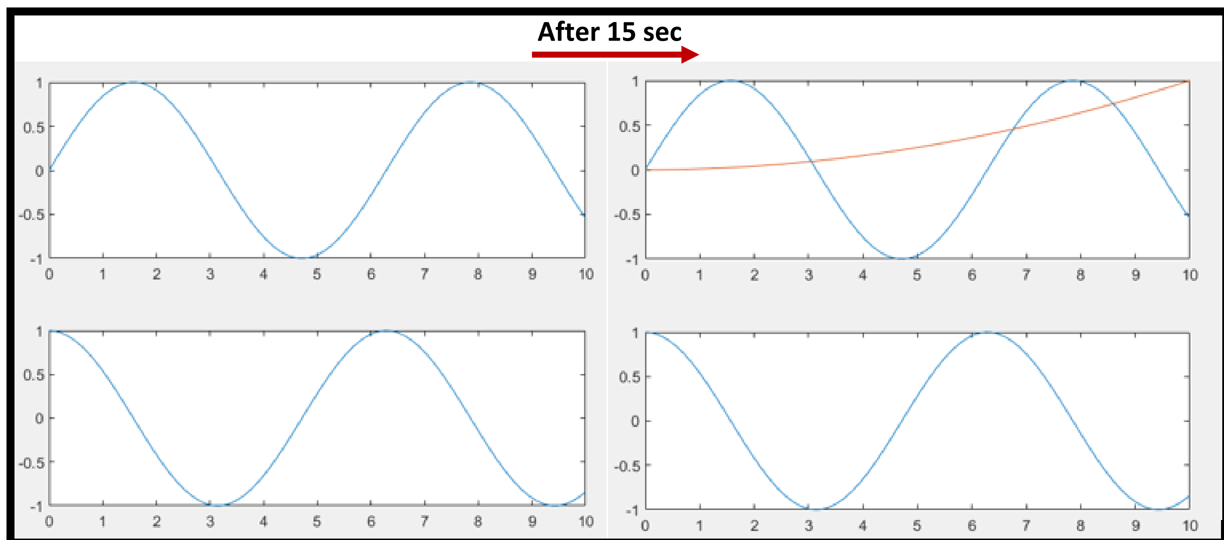
Example 1: pause until user want with press any keys then display another function in final graph

```
x = linspace(0,10);
y1 = sin(x);
y2 = cos(x);
tiledlayout(2,1)

% Top plot
ax1 = nexttile;
plot(ax1,x,y1)

% Bottom plot
ax2 = nexttile;
plot(ax2,x,y2)


pause % pause and wait until press a key then will be added this function to final graph
hold(ax1,'on')
y3 = (x.^2)/100;
plot(ax1,x,y3)
hold(ax1,'off')
```



Example 2: pause for 15 sec and then display another function in final graph

```
x = linspace(0,10);
y1 = sin(x);
y2 = cos(x);
tiledlayout(2,1)

% Top plot
ax1 = nexttile;
plot(ax1,x,y1)

% Bottom plot
ax2 = nexttile;
plot(ax2,x,y2)

n = 15;
pause(n) % 15 Sec pause and then add to final graph
hold(ax1,'on')
y3 = (x.^2)/100;
plot(ax1,x,y3)
hold(ax1,'off')
```



Reading data from files and saving data to files: A) the binary .mat versions of load and save B) the ascii versions.

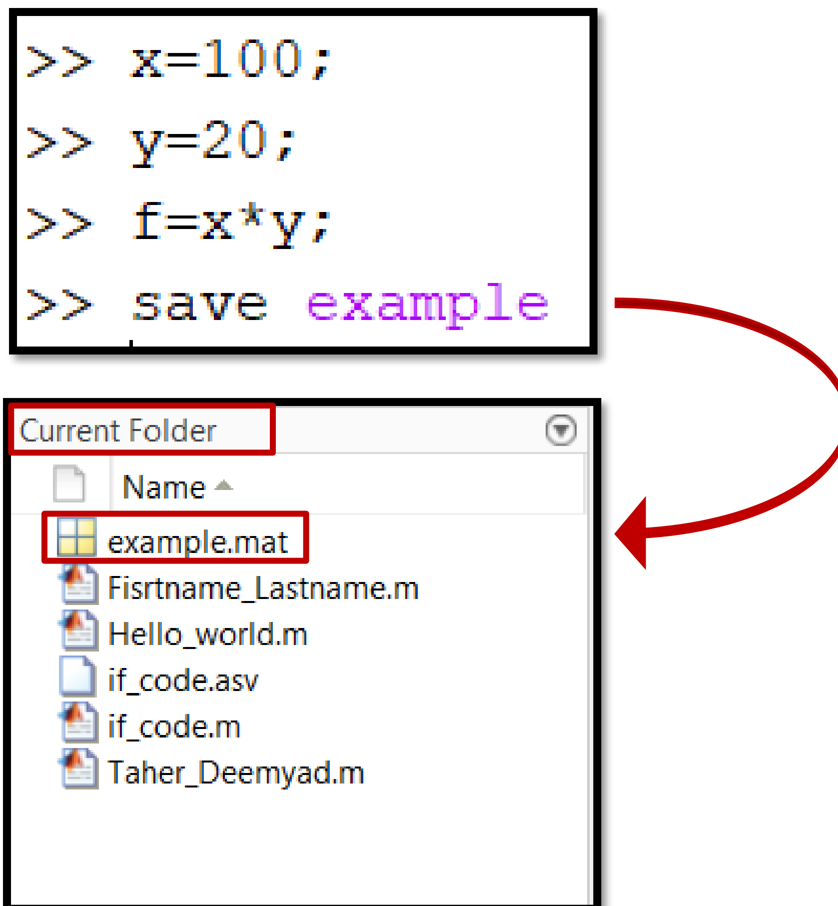
- MATLAB code is in files with extension .M.
- MATLAB data is in files with extension .MAT.

Thus, you can't always convert **.M** files to **.MAT**, but if your MATLAB code just contains variable assignments, then you can convert it.

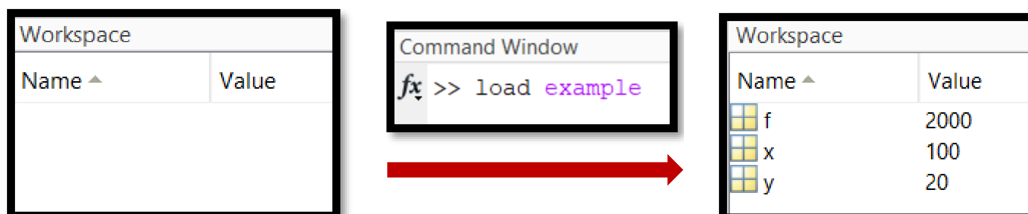
Data are stored in workspace temporary and if we use "clear" all of them will be erased. Thus, it would be better to store them in a file for future.

For creating a .MAT file or update it and add some new variables to it, we can use "save file-name" like the following example. In this way you can save all variables from the workspace in a binary MAT-file.

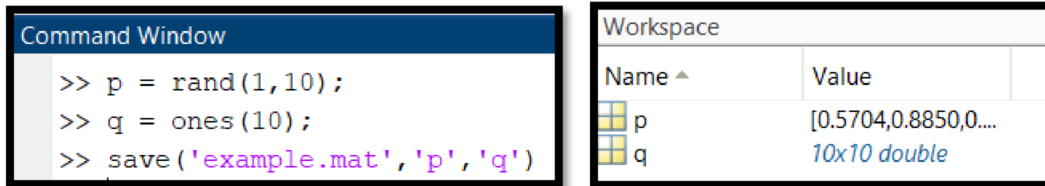
Example:



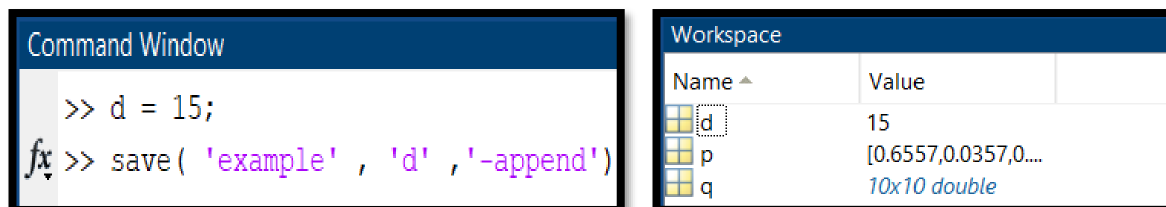
For call and reading the file, we can use "load file-name" like the following example.



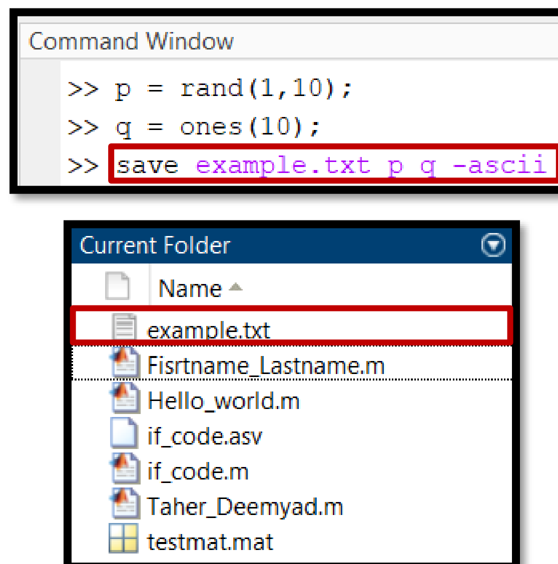
If we want to just save some specific variables to MAT-file **NOT** all variables in workspace, we can use following method.



For adding more new variables to a saved file, we can use **save** ('filename', '**variables**', '-**append**') line of code.



Also, we can save the Data to ASCII File (text format).



If you specify a text format and any variable is a two-dimensional character array, then MATLAB translates characters to their corresponding internal ASCII codes. For example, 'abc' appears in a text file as:

9.7000000e+001 9.8000000e+001 9.9000000e+001

Session 9

fprintf: Write data to text file.

Syntax:

fprintf(fileID,formatSpec,A1,...,An)

fprintf(formatSpec,A1,...,An)

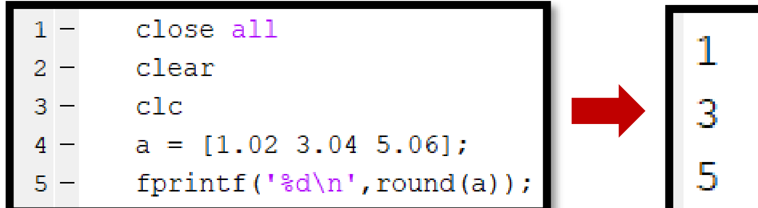
Example 1: Print multiple numeric values and literal text to the screen.

```
1 -    close all
2 -    clear
3 -    clc
4 -    A1 = [9.9, 9900];
5 -    A2 = [8.8,  7.7 ; 8800, 7700];
6 -    formatSpec = 'X is %4.2f meters or %8.3f mm\n';
7 -    fprintf(formatSpec,A1,A2)
```

```
X is 9.90 meters or 9900.000 mm
X is 8.80 meters or 8800.000 mm
X is 7.70 meters or 7700.000 mm
```

- **%4.2f** in the formatSpec input specifies that the first value in each line of output is a floating-point number with a field width of four digits, including two digits after the decimal point. **%8.3f** in the formatSpec input specifies that the second value in each line of output is a floating-point number with a field width of eight digits, including three digits after the decimal point. \n is a control character that starts a new line.

  Example 2: Explicitly convert double-precision values with fractions to integer values.



- **single precision**: $\log_{10}(2^{24})$, which is about 7~8 decimal digits
- **double precision**: $\log_{10}(2^{53})$, which is about 15~16 decimal digits

Write tabular data to text file: Write a short table of the exponential function to a text file called exp.txt.

```
1 -     close all
2 -     clear
3 -     clc
4 -     x = 0:.1:1;
5 -     A = [x; exp(x)];
6
7 -     fileID = fopen('exp.txt','w');
8 -     fprintf(fileID,'%6s %12s\n','x','exp(x)');
9 -     fprintf(fileID,'%6.2f %12.8f\n',A);
10 -    fclose(fileID);
```

Open a new file.

Write in this file.

String array

```
>> type exp.txt

     x        exp(x)
  0.00    1.00000000
  0.10    1.10517092
  0.20    1.22140276
  0.30    1.34985881
  0.40    1.49182470
  0.50    1.64872127
  0.60    1.82211880
  0.70    2.01375271
  0.80    2.22554093
  0.90    2.45960311
  1.00    2.71828183
```

Value Type	Conversion	Details
Integer, signed	%d or %i	Base 10
Integer, unsigned	%u	Base 10
	%o	Base 8 (octal)
	%x	Base 16 (hexadecimal), lowercase letters a–f
	%X	Same as %x, uppercase letters A–F
Floating-point number	%f	Fixed-point notation (Use a precision operator to specify the number of digits after the decimal point.)
	%e	Exponential notation, such as 3.141593e+00 (Use a precision operator to specify the number of digits after the decimal point.)
	%E	Same as %e, but uppercase, such as 3.141593E+00 (Use a precision operator to specify the number of digits after the decimal point.)
	%g	The more compact of %e or %f, with no trailing zeros (Use a precision operator to specify the number of significant digits.)
	%G	The more compact of %E or %f, with no trailing zeros (Use a precision operator to specify the number of significant digits.)
Characters or strings	%c	Single character
	%s	Character vector or string array. The type of the output text is the same as the type of formatSpec.

Function in MATLAB:

User-defined functions: usually engineers will need only to encode a particular computation for use with an 'ode solver' or 'integration' function already provided by MATLAB. For writing a user-defined MATLAB function, we will need a specific format and several tips must be considered.

Format of the function file:

The function Prototype or Declaration: The very first thing you see in every function file (after the comments) is the line with the keyword function on it. This line is called the function "prototype" or "declaration".

```
1      % This is a function for adding two input values to eachother!
2      function sum = addto(a,b)      → Prototype or Declaration
3 -    sum=a+b;
4 -    disp(sum);
5 -    end
```

It is important to know that this line Doesn't Do Anything! In other words, this line is a syntactic pattern that tells MATLAB important information about the function. The line is never executed in your program.

Thus, the function prototype is a means for telling MATLAB that the **M file contains a function** (not a script), **what the name of the function** is, **what its return variable** is called, and **what parameters it takes**. When the function is called (run by the computer), the computer starts at the first line of the codes right after the prototype.
In above example, adding two numbers is the first actual line of the function is:

```
1      % This is a function for adding two input values to eachother!
2      % sum = addto(a,b)
3      function sum = addto(a,b)
4 -    sum=a+b;      → First actual line of function
5 -    disp(sum);
6 -    end
```

The structure of a function is shown in following table.

Function		Description
Prototype/ Declaration Line	Return value	The goal of this function or the final value that is supposed to find by this function (In above example “sum”)
	=	Showing relation between the return value and parameters
	Function name	Right after “=” we **must** use the same name we want to save the function with that name later!
	‘(‘	Open parenthesis
	Parameters	Write all variables and parameters which will be use in function and have relation with return value
	‘)’	Close parenthesis
Codes		Write all the code after that
end		Finally use “end” command to close function

Note 1: The “return value” must be the last line of code and calculations in function.
Note 2: The name of the file must match the name of the function.

H-1 help section: The first line of help text, often called the H1 line, typically contains a brief description of the function. For above example:

```
>> help addto
 This is a function for adding two input values to eachother!
 sum = addto(a,b)
```

H-1 help section

Function with multiple return values:

Example:

```
1      % This function can be used for calculation of the base area of a cubic shape countainer
2      % plus volume and mass of liquid inside of it.
3      % [area, volume, mass] = simple_cal(W,L,H,den)
4      % w: width, L: lenght, H: hight, den: density of liquid
5
6      function [area, volume, mass] = measurement(W,L,H,den)
7 -    area = W*L;
8 -    volume = area*H;
9 -    mass = volume*den;
10 -   n = ['Arae: ' num2str(area) '  Volume: ' num2str(volume) '  Mass: ' num2str(mass)];
11 -   disp (n);
12 -   end
```

```
>> measurement(2,3,4,7);
Arae:  6   Volume: 24   Mass: 168
```

Session 10

function_handle:

A typical use of function handles is to pass a function to another function. For example, you can use function handles as input arguments to functions that evaluate mathematical expressions over a range of values.

Named function handles: represent functions in existing program files, including functions that are part of MATLAB and functions that you create using the function keyword.

Example 1:

```
>> f = @sin;
>> m = fminbnd(f,0,2*pi)

m =

    4.7124
```

- **fminbnd:** Function to find minimum of single-variable function on fixed interval.

Example 2:

```
function y = Poly_eq(x)
a=2;
b=5;
y = a*x.^3 + x/b + 1;
end
```

```
>> q = integral(@Poly_eq,2,3)

q =

   34.0000
```

Anonymous function handles: (often called *anonymous functions*) represent single inline executable expressions that return one output.

Example 3:

```
>> f = @(x,y) (x.^2 - y.^2);
>> f(2,3)

ans =

    -5
```

Example 4:

```
>> a=2;b=5;
>> f = @(x)a*x.^3 + x/b + 1;
>> q = integral(f,2,3)

q =

   34.0000
```

fsolve: Solve system of nonlinear equations.

Example 5: Solve the following two nonlinear equations in two variables.

$$(x_1)^2 + sin(x_2) = 3 * x_2 + x_{1/cos(x_2)}$$
$$e^{x_2} + 7 * x_2 = cos(x_1)$$

Convert the equations to the form F(x)=0.

$$(x_1)^2 + sin(x_2) - 3 * x_2 - (x_1)/cos(x_2) = 0$$

$$e^{x_2} + 7 * x_2 - cos(x_1) = 0$$

Base on the above equations, write a function that computes the left-hand side of these two equations.

```
function F = root2d(x)
F(1) = (x(1).^2)+ sin(x(2))-(3*x(2))-(x(1).^3);
F(2) = exp(x(2))+(7*x(2)) - 5*((x(1)).^3);
end
```

Rather than x1, x2, … we can define just one x in and then in equations show with x(1), x(2),…!

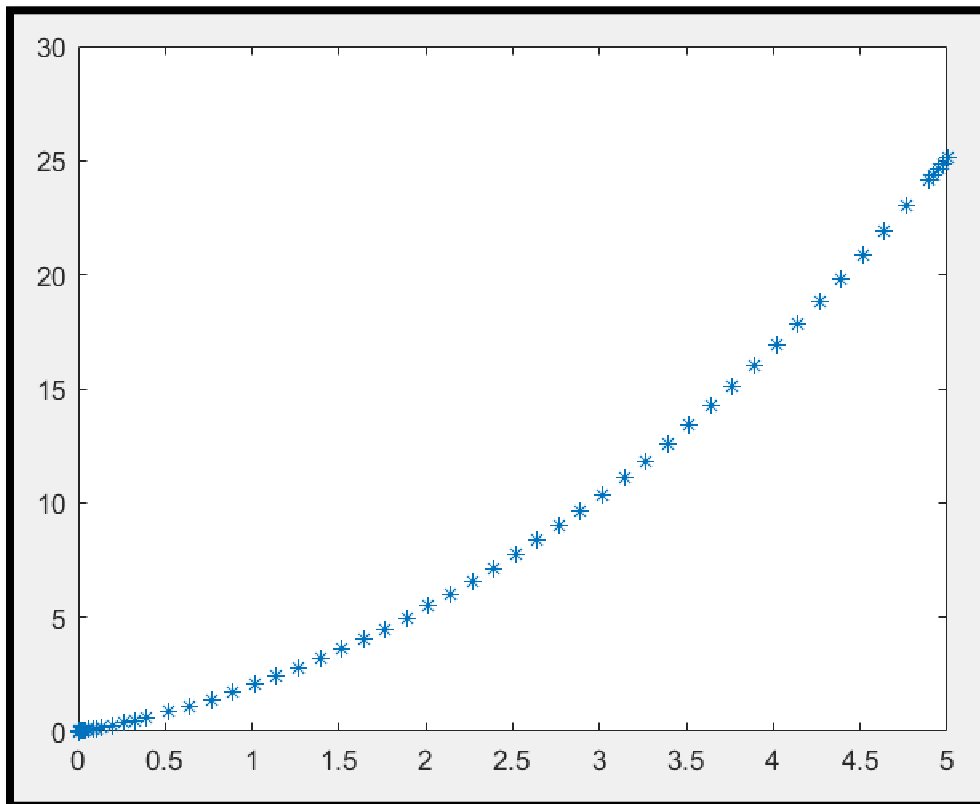
```
>> fun = @root2d;
>> x0 = [0,0];
>> x = fsolve(fun,x0)

No solution found.

fsolve stopped because the problem appears regular as measured by the gradient,
but the vector of function values is not near zero as measured by the
value of the function tolerance.

<stopping criteria details>

x =

    0.0000   -0.1183
```

ode45: Solver for differential equations — medium order method.

Simple ODEs can be used for a single solution component like following example.

Example 6: Solve $y' = e^{x^{(1/2)}}$ with use ode45 for x interval of [0,5] and the initial condition y0 = 0 and plot it.

```
>> xspan = [0 5];
>> y0 = 0;
>> [x,y] = ode45(@(x,y) exp(x.^(1/2)), xspan, y0);
>> plot(x,y,'*')
```



local and global function variables: Ordinarily, each MATLAB function has its own local variables, which are separate from those of other functions and from those of the base workspace. However, if several functions all declare a particular variable name as global, then they all share a single copy of that variable. Any change of value to that variable, in any function, is visible to all the functions that declare it as global.

Example:

Create a function and define x as a global variable.

```
1      function setGlob(value)
2 -    global x
3 -    x = value;
```

Create another function and define and use x as a global variable as well.

```
1      function r = getGlob
2 -    global x
3 -    r = x.^2;
```

These two variables in both functions become link to each other!

```
>> setGlob(2)
>> r = getGlob

r =

     4
```

- **Note 1:** To clear a global variable from all workspaces, use ***clear global variable***.
- **Note 2:** To clear a global variable from the current workspace but NOT other workspaces, use ***clear variable***.

nargin: nargin returns the number of function input arguments given in the call to the currently executing function.

Example: Create a function that accepts up to two inputs. Use nargin in the body of the function to determine the number of inputs.

```
1     function x = addme(a,b)
2 -       switch nargin
3 -           case 2
4 -               x = a + b;
5 -           case 1
6 -               x = a + a;
7 -           otherwise
8 -               x = 0;
9 -       end
10 -  end
```

```
>> addme(1,3)

ans =

     4

>> addme(3)

ans =

     6
```

nargout: nargout returns the number of function output arguments given in the call to the currently executing function.

Example: Create a function that able to calculate subtraction between two numbers and find absolute value between them if the user asking for it.

```
1    function [dif,absdif] = subtract(y,x)
2 -      dif = y-x;
3 -      if nargout > 1
4 -          disp('Calculating absolute value')
5 -          absdif = abs(dif);
6 -      end
7 -  end
```

```
>> dif=subtract(5,12)

dif =

    -7

>> [dif,absdif]=subtract(5,12)
Calculating absolute value

dif =

    -7


absdif =

     7
```