Experiment 2

Date: 14.12.23

Aim:
To perform the following operations on an array

a) Concatenation
b) Indexing
c) Shifting
d) Sorting
e) Reshaping
f) Flipping
g) Relational Operations like <,>,>=, ~=
h) Logic Operations

Theory:

a) Concatenation: Concatenating in MATLAB involves combining arrays either horizontally or vertically. It is done using **square brackets** ([]). Horizontal concatenation combines arrays side by side while vertical concatenation stacks arrays on top of each other.

b) Indexing: Indexing in MATLB refers to accessing specific elements of an array. MATLAB uses 1-based indexing, meaning the first element has an index of 1. Elements can be accessed using parentheses and the element's index.

c) Shifting: Shifting an array involves moving its elements to the left or right. This can be achieved by indexing and concatenating.

d) Sorting: Sorting in MATLAB involves arranging the elements of an array in ascending or descending order. The **'sort'** function is used for this purpose.

e) Reshaping: Flipping an array in MATLAB means reversing the order of its elements. The **'flip'** function is used for this purpose.

f) Relational Operations: Relational operations in MATLAB involves comparing elements of two arrays using operators like '<', '>', '<=' and '~='. These operators result in logical arrays.

g) Flipping: Flipping an array in MATLAB means reversing the order of its elements. 'flip' function is used for this purpose.

h) Logic operations: Combining logical values using operators like '&', '|' and '~'. These operations are performed element wise resulting in a logical array.

Input:

```
clc
%Creating a 3*3 array%
disp('Array is 3x3:')
A=[1 2 3; 4 5 6; 7 8 9]
%Conactenation%
disp('Concatenation:')
B=[A, A+1; A+5,A+3]
%Indexing Operation%
disp('Indexing:')
A(5)
A(2,3)
%Sorting%
disp('Sorting the Matrix')
sort(A,1)
sort(A,2,'descend')
sort(A,'descend')
sort(A, "ascend")
%Shift Operations%
disp("Shift:")
circshift(A,[1,0])
circshift(A,[0,1])
%Reshaping%
disp('reshaping')
reshape(A,1,9)
%Flipping%
disp("Flipping the matrix about verticle")
fliplr(A)
disp("flipping the matric about horizontal")
flipud(A)
flipud(B)
%Creating the array X and Y%
disp('array X')
X=[1,2,3];
disp('array Y')
Y=[89 88 90];
disp('Equal Operation')
X=Y
X<Y
X>Y
X~=Y
disp('Logical Operation AND')
and(X,Y)
disp('Logical Operation OR')
or(X,Y)
disp('Logical Operation NOT')
not(X)
```

```
disp('Logical Operation XOR')
xor(X, Y)
```

Output:

```
Array is 3x3:
A =
     1     2     3
     4     5     6
     7     8     9

Concatenation:
B =

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

Indexing:
ans =

     5
ans =

     6

Sorting the Matrix
ans =

     1     2     3
     4     5     6
     7     8     9
ans =

     3     2     1
     6     5     4
     9     8     7
ans =

     7     8     9
     4     5     6
     1     2     3
ans =

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

```
Shift:
ans =

     7     8     9
     1     2     3
     4     5     6
ans =

     3     1     2
     6     4     5
     9     7     8

reshaping
ans =
    1     4     7     2     5     8     3     6     9

Flipping the matrix about verticle
ans =
     3     2     1
     6     5     4
     9     8     7

flipping the matric about horizontal
ans =
     7     8     9
     4     5     6
     1     2     3
ans =
    12    13    14    10    11    12
     9    10    11     7     8     9
     6     7     8     4     5     6
     7     8     9     8     9    10
     4     5     6     5     6     7
     1     2     3     2     3     4
array X
array Y
Equal Operation
X =
    89    88    90
ans =
  1×3 logical array

   0   0   0
ans =

  1×3 logical array

   0   0   0
ans =

  1×3 logical array
```

```
   0   0   0
Logical Operation AND

ans =
  1×3 logical array
   1   1   1

Logical Operation OR

ans =
  1×3 logical array
   1   1   1

Logical Operation NOT
ans =

  1×3 logical array
 0   0   0

Logical Operation XOR
ans =
  1×3 logical array

   0   0   0
>>
```

Experiment 1

Date: 21.12.23

Aim:

a) Creating a one-dimensional array (row/column vector)
b) Creating a two-dimensional matrix of given size.
c) Performing arithmetic operations like- addition, subtraction, multiplication and exponentiation.
d) Performing matrix operations- inverse, transpose, rank.

Theory:

MATLAB short for “Matrix Laboratory “is a high-level programming language and interactive environment designed for numerical computing, data analysis, and visualization. It offers built-in functions for matrix manipulation, signal processing, optimization, and various mathematical tasks. Widely used in academia and engineering, MATLAB provides a user-friendly interface for both command-line and graphical interactions, making it suitable for algorithm development and interactive data exploration.

Creating 1D array and 2D matrix:

In MATLAB we can create 1D & 2D array:

- To create a one-dimensional array (vector), use square brackets for a row vector or separate elements with semicolons for a column vector.
- For a two-dimensional matrix, use square brackets to define the matrix and separate rows with semicolons. For example, A = [1 2; 3 4] creates a 2x2 matrix.

Performing arithmetic operations:

MATLAB provides straightforward syntax for performing arithmetic operations on arrays and matrices. Element-wise addition (+), subtraction (-), and multiplication (*) are performed by default. For exponentiation, the ^ operator is used.

Performing inverse, transpose and rank operation:

Inverse: In MATLAB, you can compute the inverse of a matrix using the **‘inv’** function. However, it's essential to check if the matrix is invertible (non-singular) before computing the inverse.

Transpose: Transposing a matrix is achieved using the apostrophe (') or the transpose function. This operation swaps the rows and columns of the matrix.

Rank: MATLAB provides the rank function to determine the rank of a matrix. The rank is the maximum number of linearly independent rows or columns in the matrix.

Input:

```
%creating a one-dimensional array%
a = [1; 2; 3; 4; 5]
x = [3 4 5 6 7]
b = [1, 2, 3, 4, 5]
%creating a multidimensional array%
c = [7, 5, 1; 7, 3, 9; 10, 1, 12]
%performing arithmetic operations%
display('Addition of matrices')
f = x + b
display('Subtraction of matrices')
g = b - x
display('Multiplication of matrices')
h = a * b
display('Exponent of matrix')
z = exp(b)
%performing matrix operations%
display('Rank of Matrix')
t = rank(c)
display('Transpose of matrix')
y = transpose(c)
display('Inverse of matrix')
n = inv(c)
display('Determinant of matrix')
u = det(c)
```

Output:

```
a =
     1
     2
     3
     4
     5
x =
     3     4     5     6     7
b =

     1     2     3     4     5
c =

     7     5     1
     7     3     9
    10     1    12
```

```
Addition of matrices


f =
     4     6     8    10    12
Subtraction of matrices
g =
   -2    -2    -2    -2    -2

Multiplication of matrices
h =
     1     2     3     4     5
     2     4     6     8    10
     3     6     9    12    15
     4     8    12    16    20
     5    10    15    20    25
Exponent of matrix
z =
    2.7183    7.3891   20.0855   54.5982  148.4132

Rank of Matrix
t =
     3
Transpose of matrix
y =
     7     7    10
     5     3     1
     1     9    12
Inverse of matrix
n =
    0.1378   -0.3010    0.2143
    0.0306    0.3776   -0.2857
   -0.1173    0.2194   -0.0714

Determinant of matrix
u =
   196
>>
```

EXPERIMENT 3

Date: 04.01.24

Aim: a) To generate random sequence and plot them
b) To calculate sum matrix, cumulative sum matrix and plot the sum matrix.

Theory:

The MATLAB code initializes a matrix A and performs column-wise and row-wise summation, as well as cumulative summation using sum and cumsum functions. Additionally, a random sequence x is generated using randn and plotted using plot, with appropriate title, x-axis label, and y-axis label for visualization. These functions are essential tools for matrix operations, random number generation, and data visualization in MATLAB.

Column-wise and Row-wise Summation:

- S1=sum(A): Computes the column-wise sum of the elements in matrix A.
- S2=sum(A,2): Computes the row-wise sum of the elements in matrix A.

Column-wise and Row-wise Cumulative Sum:

- C1=cumsum(A): Computes the column-wise cumulative sum of the elements in matrix A.
- C2=cumsum(A,2): Computes the row-wise cumulative sum of the elements in matrix A

Random Sequence Generation:

- x=randn(1,7): Generates a 1x7 vector x of random numbers from a standard normal distribution.

Plotting:

- plot(x): Creates a plot of the random sequence (x).
- title ('random function'): Adds a title to the plot.
- xlabel('random variable'): Adds a label to the x-axis.
- ylabel('f(x)'): Adds a label to the y-axis.

Input:

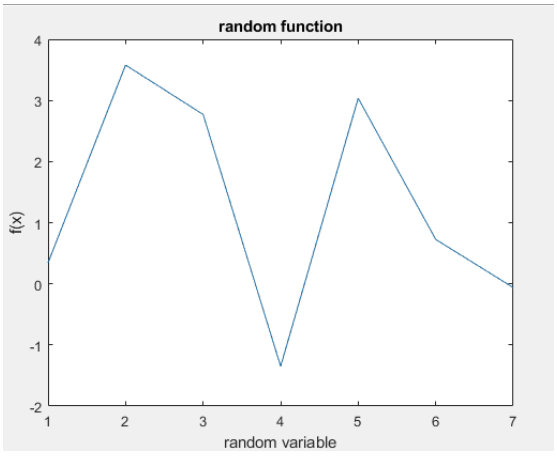
```
%Experiment 3%
%random sequence GP%
clc
A=[1 4 7; 2 5 8; 3 6 9]
%column wise addition%
S1=sum(A)
%row wise%
S2=sum(A,2)
%column wise cumulative sum%
C1=cumsum(A)
```

```
%row wise cumulative sum%
C2=cumsum(A,2)
% a i s%
x=randn(1,7)
plot(x)
title('random function');
xlabel('random variable');
ylabel('f(x)')
```

Output:

```
A =

     1     4     7
     2     5     8
     3     6     9
S1 =

     6    15    24
S2 =

    12
    15
    18
C1 =
     1     4     7
     3     9    15
     6    15    24
C2 =
     1     5    12
     2     7    15
     3     9    18
x =
    0.3426    3.5784    2.7694   -
1.3499    3.0349    0.7254   -0.0631
```

>>



EXPERIMENT 4

Date: 11 Jan 24

Aim: Evaluating a given expression and rounding it to the nearest integer value using round, ceil, floor & fix. Also generate the plots for:
a) trigonometric functions
b) logarithmic and other functions

Theory:

- **ceil (Ceiling Function):** Rounds each element to the smallest integer greater than or equal to it. Useful for rounding up to the next whole number.
- **floor (Floor Function):** Rounds each element to the largest integer less than or equal to it. Useful for rounding down to the previous whole number.
- **round (Round Function):** Rounds each element to the nearest integer. Special case: If the fractional part is exactly 0.5, it rounds to the nearest even integer.
- **fix (Fix Function):** Rounds each element towards zero (truncation), discarding the decimal part. Useful for keeping the integer part and discarding the fractional part.

Trigonometric Functions:

- **Sine (sin(t)), Cosine (cos(t)):** Represent periodic oscillations with a 2π periodicity.
- **Cosecant (csc(t)), Secant (sec(t)), Cotangent (cot(t)):** Reciprocal functions, showing inverses and vertical asymptotes.

Exponential Graph (exp(-T)): Describes exponential decay, where a quantity decreases at a rate proportional to its current value. Commonly used to model natural decay processes, such as radioactive decay.
Logarithmic Graph (log(T)): Represents logarithmic growth, where a quantity increases but with diminishing returns. Useful for describing various natural phenomena, such as population growth.
Square Root Function Graph (sqrt(T)): Represents a square root relationship, indicating that a quantity varies proportionally with the square root of another. Frequently used in physics and engineering to model scaling or growth patterns.

MATLAB's numeric manipulation functions (ceil, floor, round, fix) aid in precise data handling. Trigonometric functions (sin, cos, csc, sec, cot) represent periodic phenomena. Exponential, logarithmic, and square root functions model decay, growth, and scaling. Subplots enhance visual clarity for effective comparison and analysis.

Input:

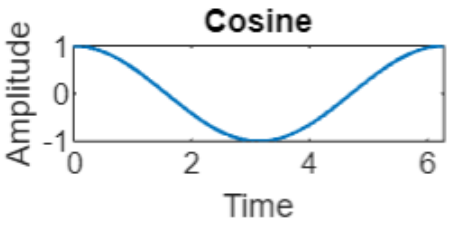
```
%Experiment-4%
clc
A=[-2.7 -0.8 0.7 2.3]
%ceil-rounds the element to the nearest integer greater than
equal to A%
display("Ceil A")
ceil(A)
%floor-rounds the element to the nearest integer less than A%
display("Floor A")
floor(A)
%round-rounds the element normally%
display("Round A")
round(A)
%fix-rounds the element towards zero%
display("Fix A")
fix(A)
%Plot a graph%
t=0:pi/100:2*pi;
y=sin(t);
plot(t,y)
subplot(3,3,1)
xlabel('Time')
ylabel('Amplitude')
title('Sine Graph')
z=cos(t);
subplot(3,3,2)
plot(t,z)
xlabel('Time')
ylabel('Amplitude')
title('Tan Graph')
u=csc(t);
plot(t,u)
subplot(1,3,1)
xlabel('time')
ylabel('Amplitude')
title("Cosec Graph")
v=sec(t);
plot(t,v)
subplot(1,3,2)
xlabel('Time')
ylabel('Amplitude')
title("Secant Graph")
b=cot(t);
plot(t,b)
subplot(1,3,3)
xlabel('Time')
ylabel('Amplitude')
title('Cot Graph')
T=0:0.1:10;
```

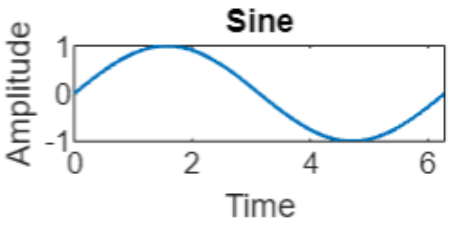
```
P=exp(-T);
subplot(3,3,4)
plot(T,P)
xlabel('Time')
ylabel('Amplitude')
title('Exponential Graph')
q=log(T);
subplot(3,3,5)
plot(T,q)
xlabel("Time")
ylabel("Amplitude")
title("Logarithmic Graph")
r=log10(T);
subplot(3,3,6)
plot(T,r)
xlabel('Time')
ylabel("amplitude")
title("Logarithmic 10 Graph")
s=log(T);
subplot(3,3,7)
plot(T,s)
xlabel("Time")
ylabel("Amplitude")
title("Square root function graph")
```

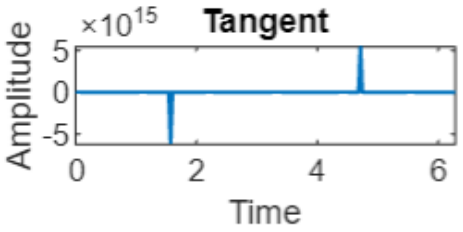
Output:

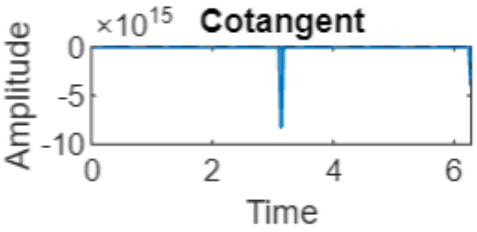
```
A =
   -2.7000   -0.8000    0.7000    2.3000
    "Ceil A"

ans =
    -2     0     1     3
    "Floor A"

ans =
    -3    -1     0     2
    "Round A"

ans =
    -3    -1     1     2
    "Fix A"

ans =
    -2     0     0     2

>>
```

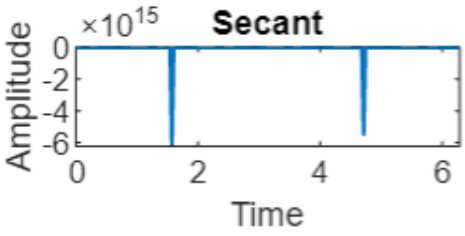
Plots:

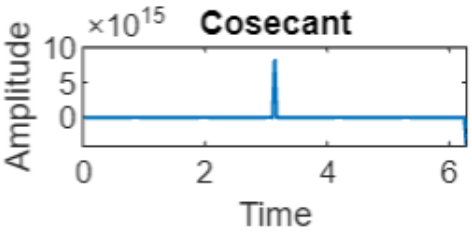

Cosine
Amplitude
Time

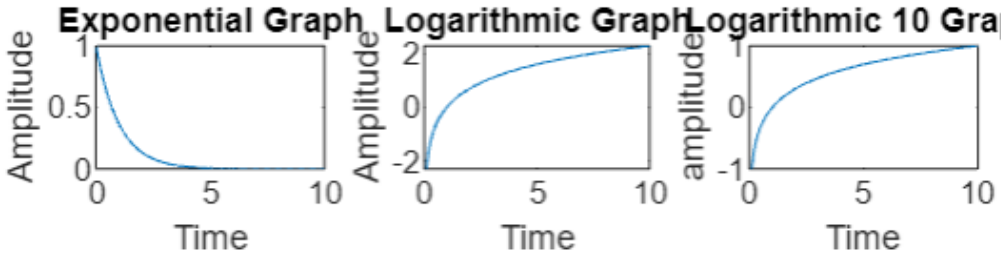

Sine
Amplitude
Time

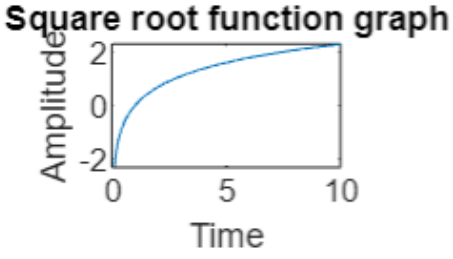

Tangent
×10^15
Amplitude
Time


Cotangent
×10^15
Amplitude
Time


Secant
×10^15
Amplitude
Time


Cosecant
×10^15
Amplitude
Time


Exponential Graph
Logarithmic Graph
Logarithmic 10 Graph
Amplitude
amplitude
Time


Square root function graph
Amplitude
Time

EXPERIMENT 5

Date: 18 Jan 24

Aim: Creating a vector x with elements x_n=(-1) n+1/(2n-1) upto 100 elements of the vector x and plotting the functions x^2,x^3,ex, exp(x2) over the interval 0<x<4 (by choosing appropriate mesh values for x to obtain smooth curves) on a rectangular plot.

Theory:

In MATLAB, when applying the exponential function (exp) to a vector or matrix, the dot (.) is used to ensure element-wise exponentiation. Without the dot, the operation would be interpreted as matrix exponentiation, which is different from exponentiating each element individually. This distinction is vital for accurate calculations when dealing with arrays of values. Element-wise operations are fundamental for treating each element independently, providing flexibility and efficiency in numerical computations.

The dot notation allows for a seamless integration of mathematical functions into array operations, offering both flexibility and computational efficiency. This becomes particularly essential when working with multidimensional arrays, ensuring precise numerical computations across diverse datasets.

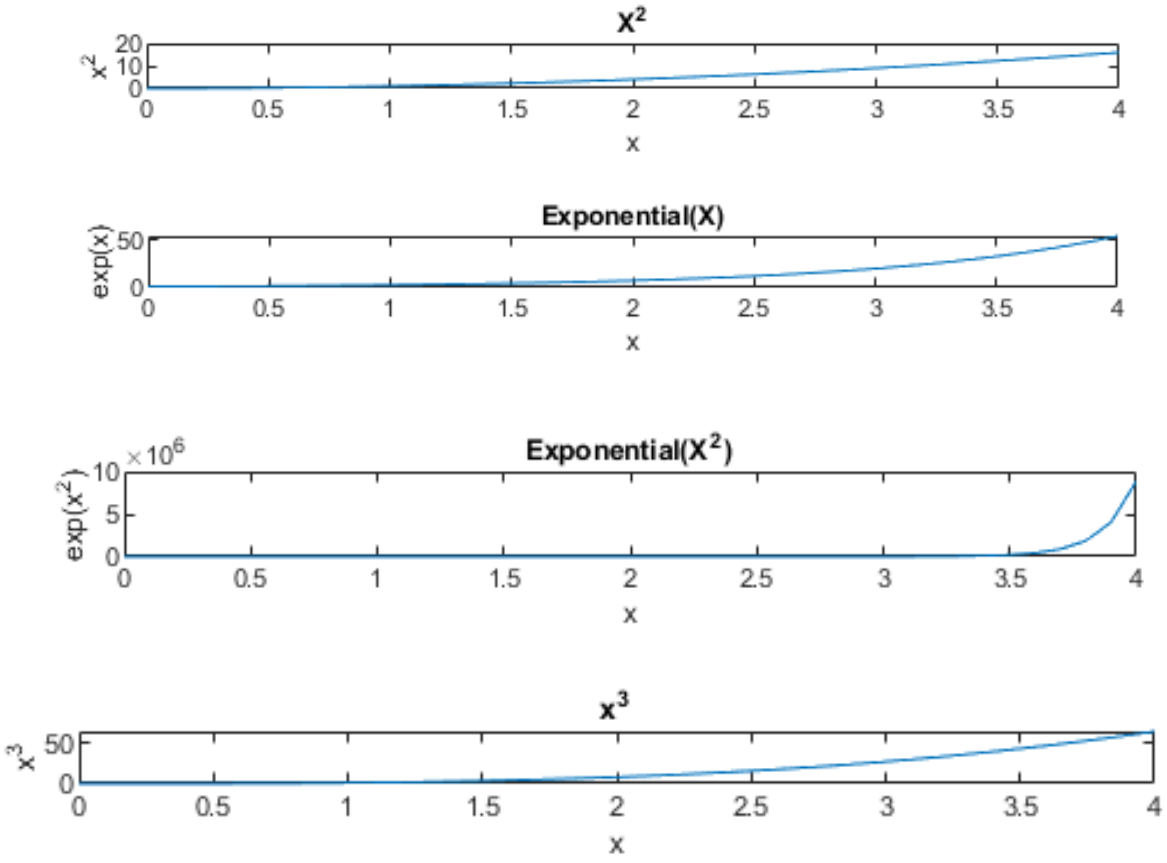
Input:

```
clc
%declaring n%
n=1:100;
%declaring Xn%
Xn=((-1).^(n+1)./(2*n-1));
plot(n,Xn)
subplot(5,1,1)
title('Xn')
xlabel('n')
ylabel('Xn')
sum(Xn)
%declaring x%
x=0:0.1:4;
%declaring function%
Z1=x.^2;
plot(x,Z1)
subplot(5,1,2)
title('X^2')
xlabel('x')
ylabel('x^2')
Z2=x.^3;
plot(x,Z2)
subplot(5,1,3)
```

```
title('x^3')
xlabel('x')
ylabel('x^3')
Z3=exp(x);

plot(x,Z3)
subplot(5,1,4)
title('Exponential(X)')
xlabel('x')
ylabel('exp(x)')
Z4=exp(Z1);
plot(x,Z4)
subplot(5,1,5)
title('Exponential(X^2)')
xlabel('x')
ylabel('exp(x^2)')
```

Output:

```
Sum is:
ans =
    0.7829

>>
```



EXPERIMENT 6

Date: 25 Jan 24

Aim: Generating a Sinusoidal Signal of a given frequency (say,100Hz) and plotting with Graphical Enhancements - titling, labelling, adding text, adding legends, adding new plots to existing plot, printing text in Greek letters, plotting as multiple and subplot.

Theory:

The sin function operates element-wise on arrays. The function's domains and ranges include complex values. All angles are in radians.

Y = sin(X) returns the circular sine of the elements of X.

MATLAB allows you to add title, labels along the x-axis and y-axis, grid lines and also to adjust the axes to spruce up the graph.

1. The x label and y label commands generate labels along x-axis and y-axis.

2. The title command allows you to put a title on the graph

3. The grid on command allows you to put the grid lines on the graph.

4. The axis equal command allows generating the plot with the same scale factors and the spaces on both axes.

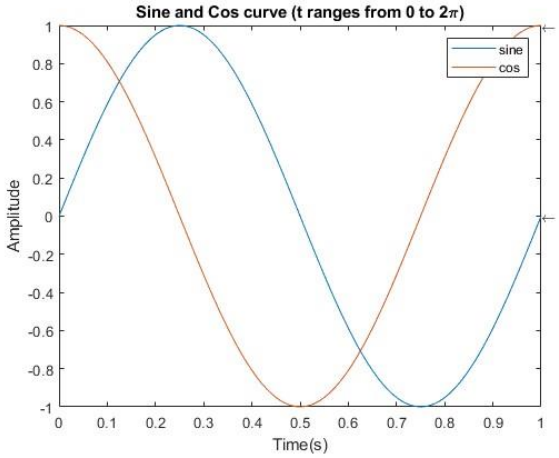
5. The axis square command generates a square plot.

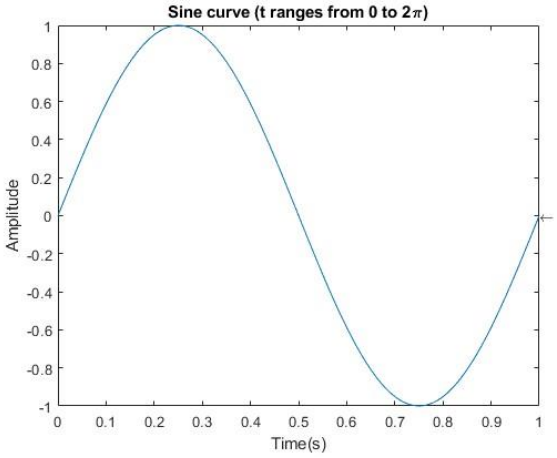
Input:

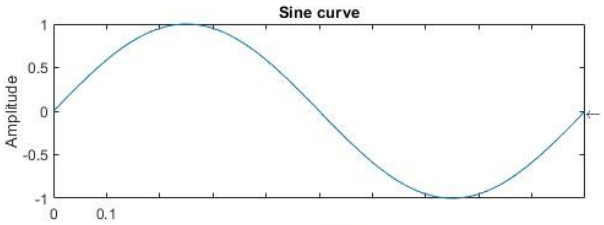
```
%plot sine wave%
t = 0 : 1/100 : 1;
f0 = 1
s1 = sin(2 * pi * f0 * t);
figure;
plot(t, s1);
xlabel('Time(s)');
ylabel('Amplitude');
title('Sine curve (t ranges from 0 to 2\pi)');
text(01, 0, '\leftarrow')
s2 = cos(2 * pi * f0 *
t);
figure;
plot(t, s1, t, s2);
legend('sine','cos');
xlabel('Time(s)');
ylabel('Amplitude');
title('Sine and Cos curve (t ranges from 0 to 2\pi)');
text(01, 0, '\leftarrow');
text(01, 1, '\leftarrow')

figure;
subplot(2, 1, 1); plot(t, s1);
xlabel("Time(s");
```

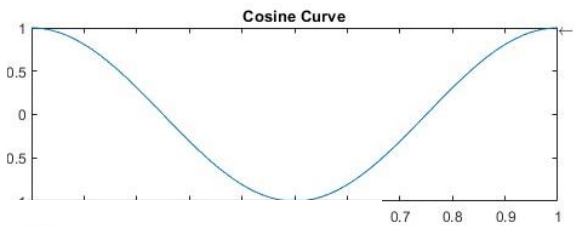
```
ylabel("Amplitude");
title("Sine curve");
text(01, 0, '\leftarrow');

figure;
subplot(2, 1, 2); plot(t, s2);
xlabel("Time(s)");
ylabel("Amplitude");
title("Cosine Curve");
text(01, 1, '\leftarrow');

s3 = sin(2 * pi * f0 * 2 * t);
figure;
subplot(4, 1, 1); plot(t, s3);
xlabel("Time(s)");
ylabel("Amplitude");
text(01, 0, '\leftarrow');

s4 = sin(2 * pi * f0 * 4 * t);
subplot(4, 1, 2); plot(t, s4);
xlabel("Time(s)");
ylabel("Amplitude");
text(01, 0, '\leftarrow');

s5 = sin(2 * pi * f0 * 0.25 * t);
subplot(4, 1, 3); plot(t, s5);
xlabel("Time(s)");
ylabel("Amplitude");
text(01, 1, '\leftarrow');

s6 = sin(2 * pi * f0 * 0.0625 * t);
subplot(4, 1, 4); plot(t, s6);
xlabel("Time(s)");
ylabel("Amplitude");
text(01, 1, '\leftarrow');

%Greek Letters%
t = 1:900;
y = 0.25*exp(-0.005*t);

figure
plot(t,y)
title('Ae^{\alphat} for A = 0.25 and \alpha = -0.0005')
xlabel('Time')
ylabel('Amplitude')
```

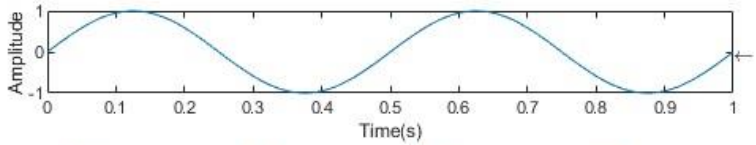
Output:

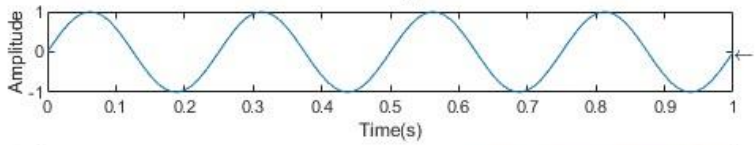

Sine and Cos curve (t ranges from 0 to 2π)
sine
cos
Amplitude
Time(s)

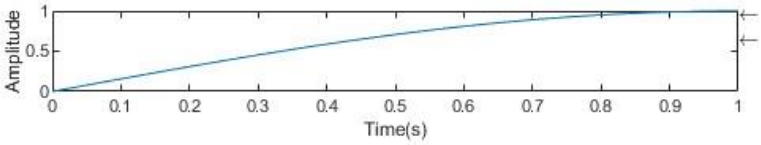

Sine curve (t ranges from 0 to 2π)
Amplitude
Time(s)

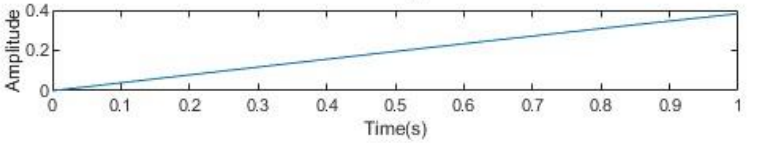

Sine curve
Amplitude

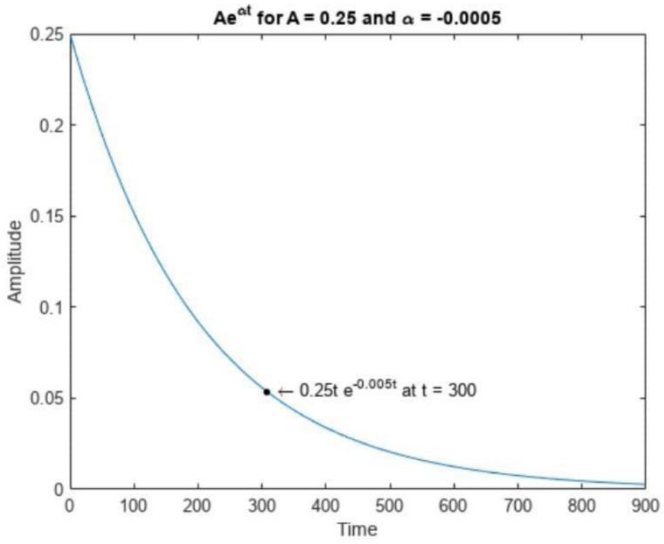

Cosine Curve


Amplitude
Time(s)


Amplitude
Time(s)


Amplitude
Time(s)


Amplitude
Time(s)


Ae^αt for A = 0.25 and α = -0.0005
Amplitude
Time
← 0.25t e^-0.005t at t = 300
0.25
0.2
0.15
0.1
0.05
0
0
100
200
300
400
500
600
700
800
900

EXPERIMENT 7

Date: 01 Feb 24

Aim: To generate square waves from a sum of sine waves.

Theory:

Generating a square wave form as the sum of sine waves involves utilizing the concept of Fourier series. The Fourier series states that any periodic waveform can be expressed as the sum of sinusoidal functions (sine and cosine waves) of different frequencies, each with its own amplitude and phase.

The general form of a Fourier series representation for a periodic function *f*(*t*) with period *T* is given by:

$$f(t) = \frac{a_0}{2} + \sum_{n=1}^{\infty} \left(a_n \cos\left(\frac{2\pi nt}{T}\right) + b_n \sin\left(\frac{2\pi nt}{T}\right) \right)$$

Where:

- a_0 is the DC (direct current) component or average value of the function.
- a_n and b_n are the coefficients of the cosine and sine terms respectively, representing the amplitudes of the respective frequency components.

For a square wave, which is a periodic waveform with a duty cycle of 50%, the Fourier series simplifies to include only odd harmonics. The amplitude of each harmonic decreases as the frequency increases, with the fundamental frequency (the frequency of the square wave) having the largest amplitude.

To generate a square wave as the sum of sine waves, follow these steps:

- Determine the Frequencies: Decide the frequencies of the sine waves to be included in the Fourier series. For a square wave, typically only odd harmonics are included, with the fundamental frequency being the lowest.
- Calculate the Amplitudes: Determine the amplitudes of each sine wave component. The amplitude of each harmonic is inversely proportional to its frequency.
- Sum the Sine Waves: Add up the sine waves with their respective frequencies and amplitudes according to the Fourier series equation.
- Normalize: Normalize the resultant waveform to ensure it falls within the desired range, typically between -1 and 1 for a normalized square wave.

- Plotting: Utilize a plotting library such as Matplotlib to visualize the generated square wave.

By summing sine waves of appropriate frequencies and amplitudes, the resulting waveform will approximate a square wave. This approach demonstrates the power of Fourier series in decomposing complex signals into simpler sinusoidal components, enabling the synthesis of various waveforms for practical applications in signal processing and communication systems.

Input:

```
clc
theta= 0: pi/20 : 2*pi;
sine=sin(theta)
sq=sine;
for n=3:2:100
sq=sq+sin(n*theta)./n;
end
figure;
plot(theta,sine,theta,sq);
xlabel("t");
ylabel("amplitude");
legend("sine","square")
```

Output:

EXPERIMENT 8

Date: 08 Feb 24

Aim: Calculate the value of the function (h(T)) based on the user-input value of (T).

Theory:

Program files can be scripts that simply execute a series of MATLAB statements or they can be functions that also accept input arguments and produce output. Both scripts with a.m extension. However, functions are more flexible and more easily extensible.

MATLAB expressions usually consist of variables or smaller expressions joined by relational operators (eg count <limit), or logical functions (eg, is real(A)). Simple expressions can be combined by logical operators (&&, ||,~) into compound expressions such as the following. MATLAB evaluates compound expressions from left to right, adhering to operator precedence rules. (count < limit) && ((height offset) > 0).

Nested if statements must each be paired with a matching end.

The if function can be used alone or with the else and elseif functions. When using elseif and/or else within an if statement, the general form of the statement is-

if expression1 statements1

elseif expression2 statements2

else statement3

end

Input:

```
clc
T = input("Enter the value of T for the function h(T): ");
if (0<T && T < 100)
   disp("Value of h")
   h = T - 10
else
    (T > 100);
   disp("Value of h")
   h = 0.45 * T + 900
end
```

Output:

```
Enter the value of T for the fucntion h(T): 15
Value of h

h =

     5
```

Experiment 9

Date: 22 Feb 24

Aim: Basic 2D and 3D plots polygon with vertices, 3D contours lines, pie and bar charts.

Theory:

Basic 2D and 3D plots can compare sets of data, track changes in data error time, or show data distribution for polygon. We use patch function for pie graph we use pie(p)function for bar graph we used bar(s) for basis 2D OR 3D we use simple logics.

ezplot(fun) plots the expression fun(x) over the default domain -2\pi<x<2\pi. where f~un(x) is an explicit function of only x.

fun can be a function handle, a character vector, or a string

ezsurf(fun) creates a graph of fun (x,y) using the surf [surf X,Y,Z] creates a three-dimensional surface plot, which is a three-dimensional surface that has solid edge colors and solid face colors. The function plots the values in matrix Z as heights above a grid in the x-y plane defined by X and Y. The color of the surface varies according to the heights specified by Z. function), fun is plotted over the default domain: -2\pi<x<2\pi, -2\pi<y<2\pi

fun can be a function handle, a character vector, or a string

[X,Y] =meshgrid(x, y) returns 2-D grid coordinates based on the coordinates contained in vectors x and y. X is a matrix where each row is a copy of x, and Y is a matrix where each column is a copy of y. The grid represented by the coordinates X and Y has length(y) rows and length(x) columns.

surfc(Z) creates a surface and contour plot and uses the column and row indices of the elements in Z as the x and y-coordinates

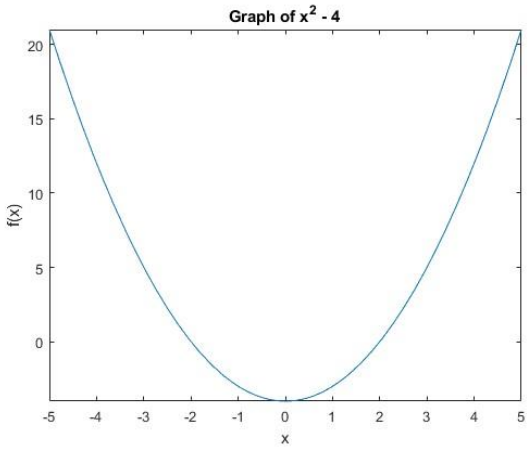
contour (Z) creates a contour plot containing the isolines of matrix Z. where Z contains height values on the x-y plane MATLAB automatically selects the contour lines to display. The column and row indices of Z are the x and y coordinates in the plane, respectively.

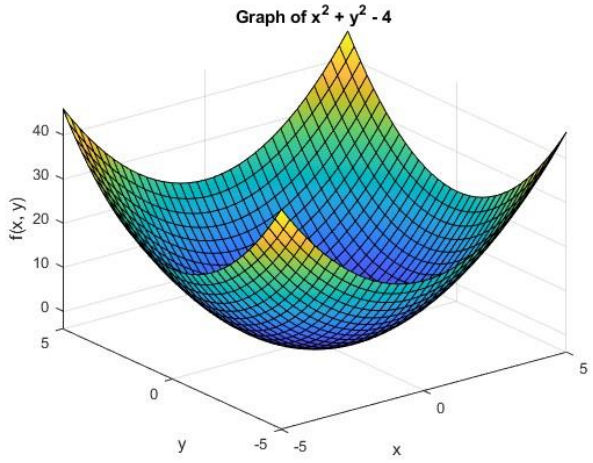
Input:

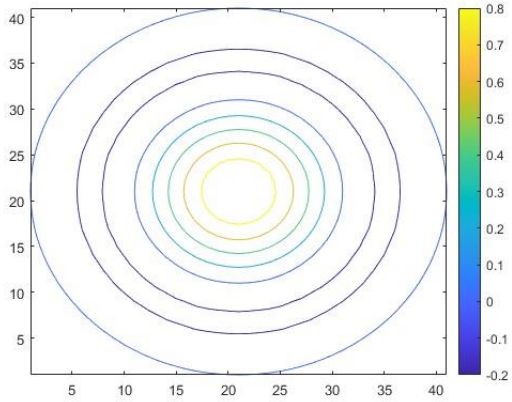
```
%parametric curve%
fplot(x^2 - 4)
title('Graph of x^2 - 4')
xlabel('x')
ylabel('f(x)')

%parametric surface%
fsurf(x^2 + y^2 - 4)
title('Graph of x^2 + y^2 - 4')
xlabel('x')
ylabel('y')
zlabel('f(x, y)')

%2D polygons with vertices%
x1 = -pi/6 : 2*pi/3 : 2*pi-pi/6
plot(cos(x1), sin(x1))
```

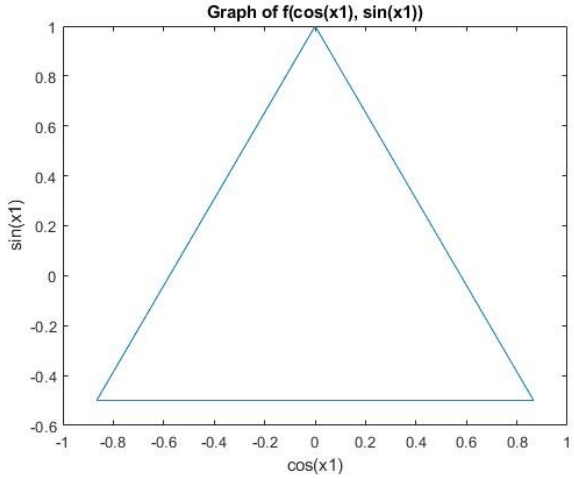
```
title('Graph of f(cos(x1), sin(x1))')
xlabel('cos(x1)')
ylabel('sin(x1)')

x2 = -pi/4 : 2*pi/4 : 2*pi-pi/4
plot(cos(x2), sin(x2))
title('Graph of f(cos(x2), sin(x2))')
xlabel('cos(x2)')
ylabel('sin(x2)')

x3 = 0 : 2*pi/6 : 2*pi
plot(cos(x3), sin(x3))
title('Graph of f(cos(x3), sin(x3))')
xlabel('cos(x3)')
ylabel('sin(x3)')

%surface, contour plots%
y1 = -2*pi : pi/10 : 2*pi
y2 = -2*pi : pi/10 : 2*pi
[X, Y] = meshgrid(y1, y2);
R = sqrt(X.^2 + Y.^2);
clear Z
Z = sin(R) ./ R;
surfc(Z)
xlabel('X')
ylabel('Y')
zlabel('Z')
contour(Z)
colorbar

%pie and bar-charts%
z1 = [1, 2, 4, 1, 2]
pie(z1)
bar(z1)
```

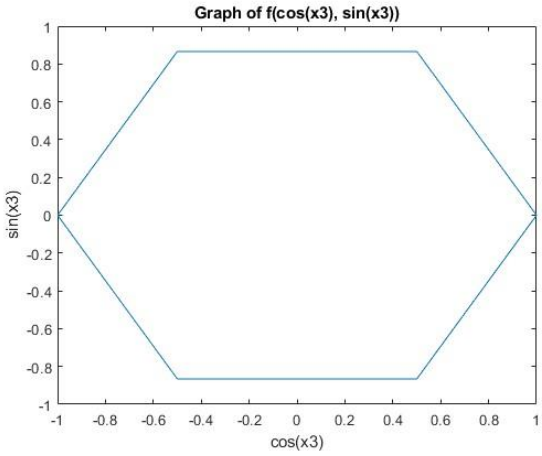
Output:

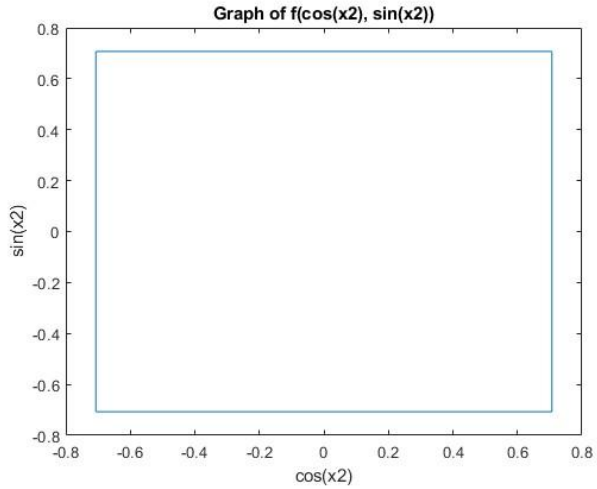


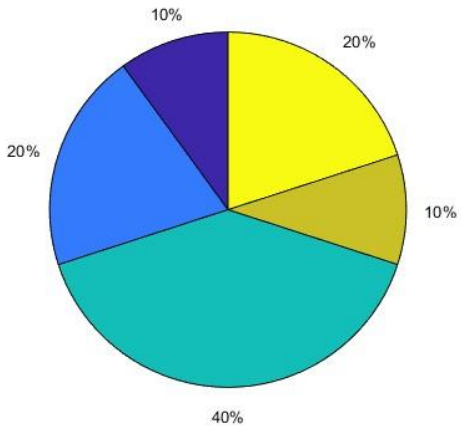

Graph of $x^2 + y^2 - 4$
f(x, y)
y
x

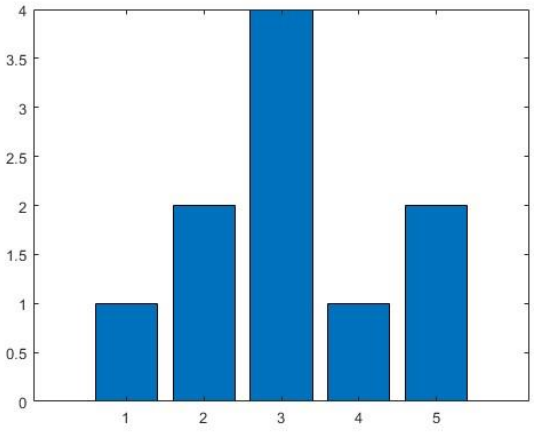



Graph of f(cos(x1), sin(x1))
sin(x1)
cos(x1)


Graph of f(cos(x3), sin(x3))
sin(x3)
cos(x3)


Graph of f(cos(x2), sin(x2))
sin(x2)
cos(x2)


10%
20%
20%
10%
40%


4
3.5
3
2.5
2
1.5
1
0.5
0
1
2
3
4
5

EXPERIMENT 10

Date: 29 Feb 24

Aim: Solving first order and second order ordinary differential equations using built in functions

Theory:

To solve first and second-order ordinary differential equations (ODEs) in MATLAB, we can leverage its symbolic computation capabilities.

For first-order ODEs, we define symbolic variables using the `syms` function, then express the ODE using symbolic derivatives (`diff` function). The `dsolve` function is then employed to solve the ODE symbolically. Optionally, we can provide initial conditions using symbolic expressions if required for specific problems.

Similarly, for second-order ODEs, we follow a similar procedure. We define symbolic variables, express the ODE using symbolic derivatives (including second derivatives), and then use `dsolve` to solve the ODE symbolically. Additionally, we may need to provide initial conditions for both the function and its derivative, specifying them along with the differential equation when using `dsolve`.

Once we have the solutions obtained symbolically, we can display them using `disp` or any other appropriate MATLAB functions for displaying results.

In summary, MATLAB provides a powerful symbolic computation toolbox for solving both first and second-order ODEs. By leveraging symbolic variables and functions, along with the `dsolve` function, you can efficiently tackle a wide range of differential equations, making it a valuable tool for scientific computing and engineering applications.

Code:

```
clc
syms y(t)
ode = diff(y,t) == t*y;
ode(t) = diff(y(t),t) == t*y(t);
ysol(t) = dsolve(ode);
disp(ysol);

syms y(x)
Dy = diff(y);
ode = diff(y,x,2) == cos(2*x) - y;
cond1 = y(0) == 1;
cond2 = Dy(0) == 0;
conds = [cond1 cond2];
ysol(x) = dsolve(ode,conds);
ysol = simplify(ysol);
disp(ysol);
```

Output:

```
C1*exp(t^2/2)
symbolic function inputs: t

1 - (8*sin(x/2)^4)/3
symbolic function inputs: x
```