FAZAIA BILQUIS COLLEGE OF EDUCATION FOR WOMEN PAF BASE NUR KHAN

COMPUTER SCIENCE DEPARTMENT

NUMERICAL ANALYSIS AND COMPUTATION

MANUAL

SUBMITED TO

Ms. Saadia Noor

SUBMITTED BY

- **Aafreen Zahra Kazmi (37)**
- **Fatima Tariq (55)**
- **Alia Mahreez (52)**
- **Arooj Khan (81)**
- **Laiba Fatima (62)**
- **Rubina Shaheen (45)**
- **Laraib Noor (48)**
- **Esha Irfan (47)**
- **Nimra Mumtaz (77)**
- **Khadija Riaz (40)**
- **Hafsa Fatima (59)**

SEMESTER / SECTION

- **5th B**

SUBMISSION DATE

- **6th- JAN-2025**

Contents

WEEK#01

1. INTRODUCTION TO MATLAB:

INTRODUCTION TO MATLAB

MATLAB (short for **Matrix Laboratory**) is a high-level programming language and environment developed by Math Works. It is widely used for numerical computing, data analysis, algorithm development, and visualization. MATLAB's user-friendly interface and robust toolbox ecosystem make it a popular choice for engineers, scientists, and mathematicians.

KEY FEATURES OF MATLAB

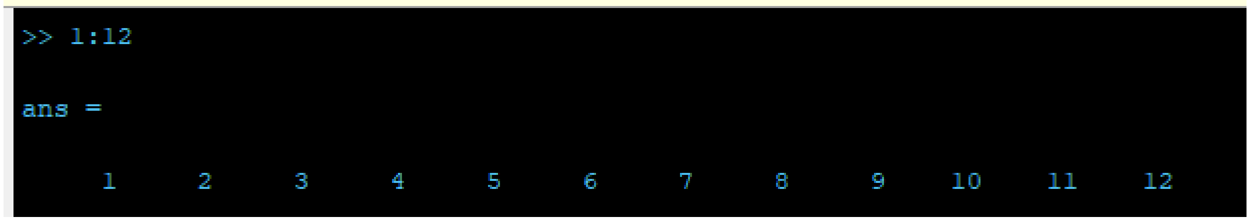
1. **Matrix-Based Computing**:
   - MATLAB is built around the concept of matrices, making it ideal for linear algebra and numerical computation.
2. **Built-in Functions and Toolboxes**:
   - Offers a vast library of built-in functions for various tasks, such as signal processing, image processing, optimization, and machine learning.
   - Specialized toolboxes extend its capabilities to specific fields like control systems, bioinformatics, and finance.
3. **Interactive Environment**:
   - Provides an intuitive environment for running commands, visualizing results, and experimenting with data interactively.
4. **Visualization Capabilities**:
   - Powerful tools for 2D and 3D data visualization, making it easier to analyze and present data.
5. **Integration with Other Languages**:
   - MATLAB can interface with other programming languages such as Python, C/C++, Java, and Fortran for enhanced functionality.
6. **App Designer**:
   - A drag-and-drop tool for creating graphical user interfaces (GUIs) within MATLAB.
7. **Simulink**:
   - An add-on tool for modeling, simulating, and analyzing dynamic systems.
8. **Cross-Platform Support**:
   - Available on Windows, macOS, and Linux, ensuring wide accessibility.
9. **Code Generation**:
   - Supports automatic C and HDL code generation, allowing deployment on embedded systems.
10. **Community and Documentation**:
    - Extensive official documentation and an active user community for support and collaboration.

1.1. UNDERSTANDING OF MATLAB WITH THE HELP OF BASIC COMMANDS:

Here are some executed Command:

1.1.1. Array Declare

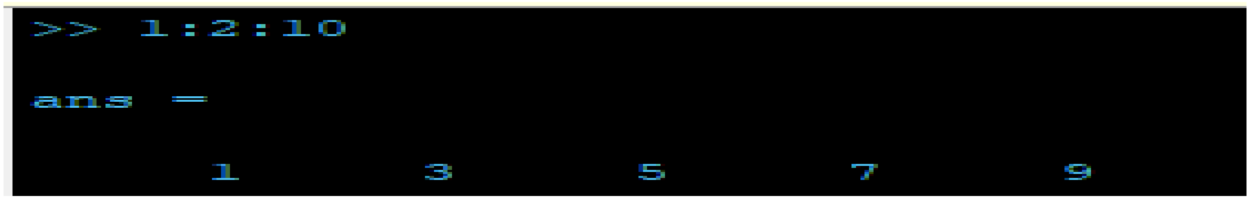
- Arrays are declared by assigning values enclosed in square brackets `[]`, where elements are separated by spaces or commas for rows, and semicolons; for new rows.



```
>> 1:12

ans =

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

1.1.2. Array Declaration with difference of 2 numbers

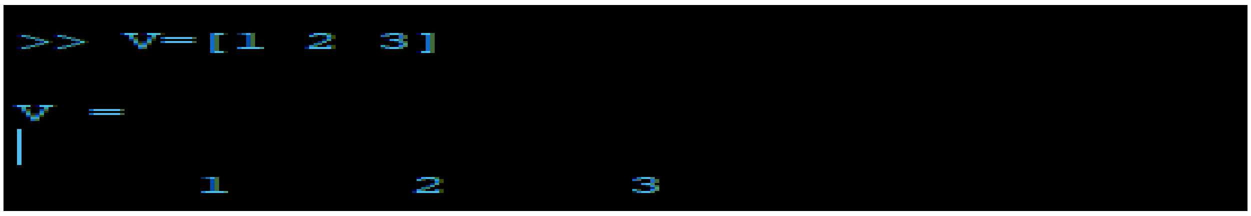
- An array with a constant difference between elements is declared using the colon operator (start:step:end), where step defines the difference.



```
>> 1:2:10

ans =

     1     3     5     7     9
```

1.1.3. Define Vectors

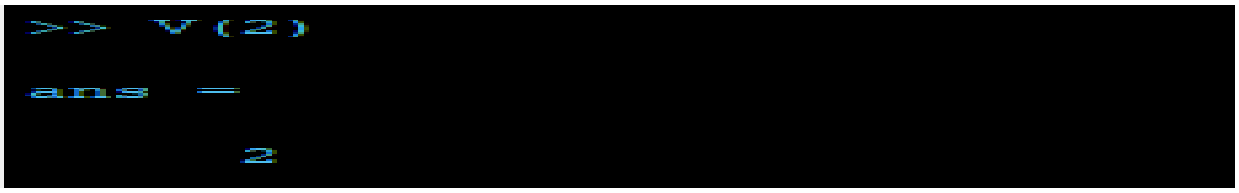
- Vectors are one-dimensional arrays defined using square brackets [] with row vectors using space/comma-separated values and column vectors using semicolons.



```
>> v=[1 2 3]

v =

     1     2     3
```

1.1.4. Find values of vectors by their position.

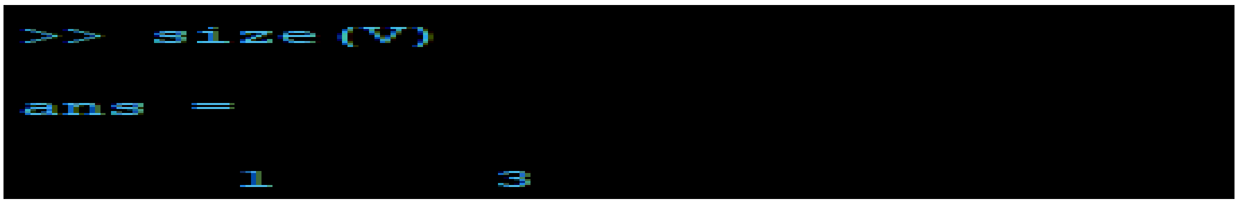
- Values of vectors can be accessed by their position using parentheses (index) where index specifies the position.



```
>> v(2)

ans =

     2
```

1.1.5. Size of vector.

- The size of a vector in MATLAB is obtained using the size function, which returns the number of rows and columns.

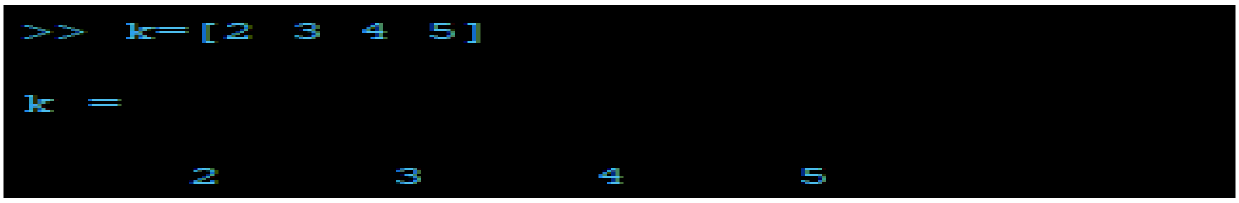


```
>> size(V)

ans =

     1     3
```

Here 1 represent (row) and 3 represent (column).

1.1.6. Define a matrix

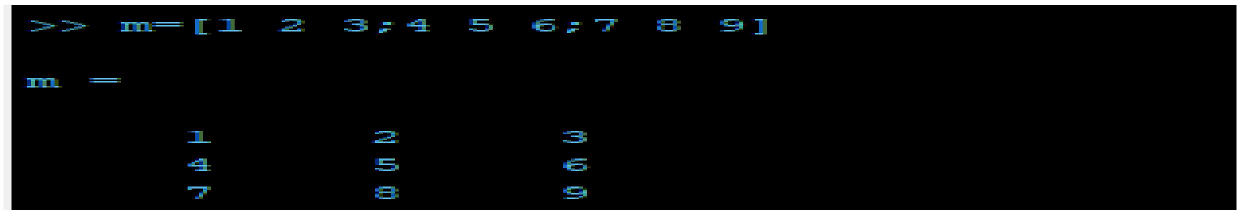
- A matrix is defined using square brackets [], with elements in rows separated by spaces or commas and rows separated by semicolons.



```
>> k=[2 3 4 5]

k =

     2     3     4     5
```

1.1.7. Define Multiple row matrix.

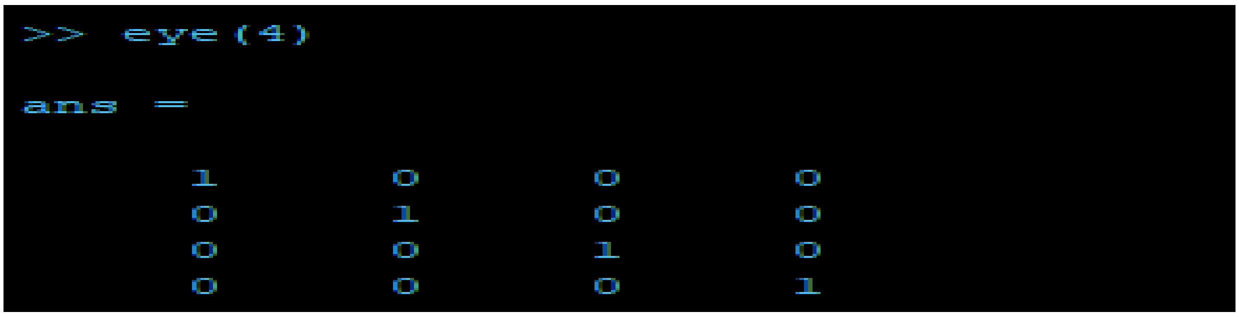
- A multiple-row matrix is defined using square brackets [], separating rows with semicolons ;.



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

m =

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

1.1.8. Define Identity Matrix.

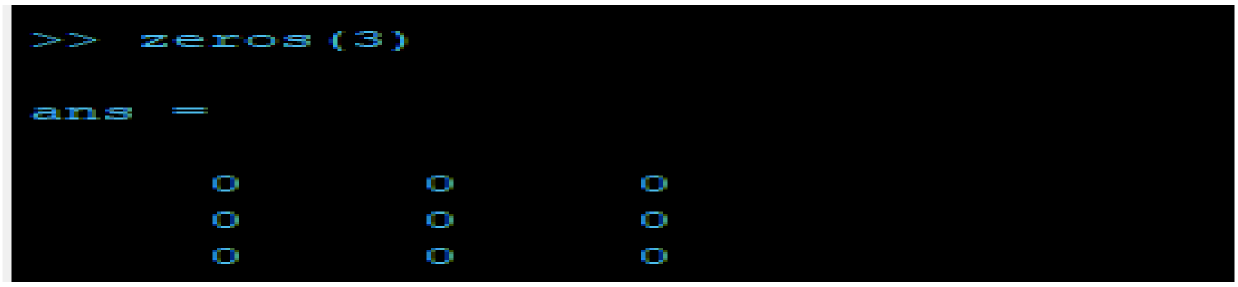
- An identity matrix is defined in MATLAB using eye(n) where n is the size of the matrix.



```
>> eye(4)

ans =

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

1.1.9. Define Zero Matrix.

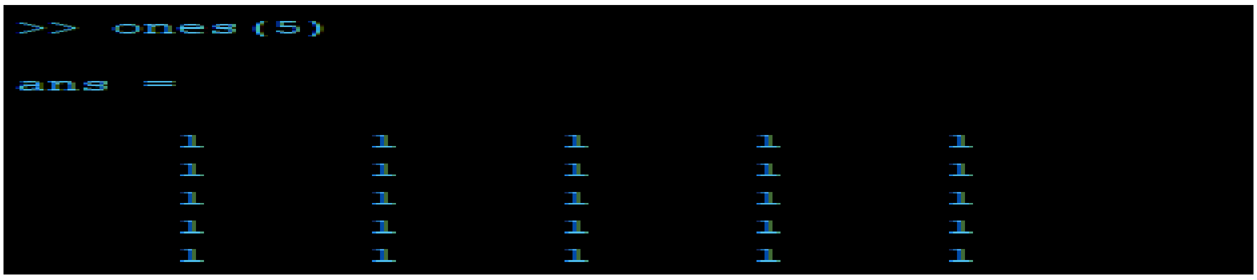
- A zero matrix is defined in MATLAB using zeros(m, n) where m is the number of rows and n is the number of columns.



```
>> zeros(3)

ans =

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

1.1.10. Define Ones Matrix.

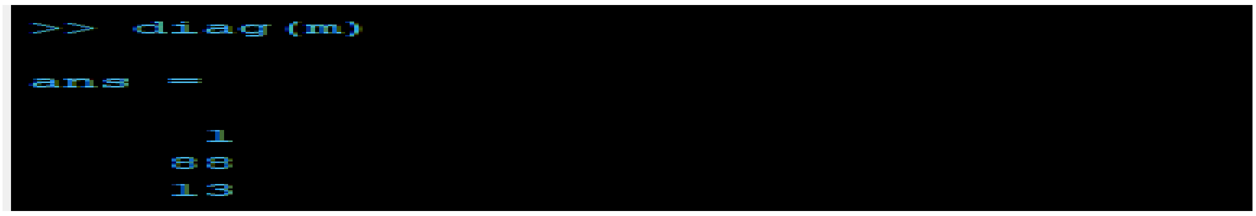
- A ones matrix is defined in MATLAB using ones(m, n) where m is the number of rows and n is the number of columns.



```
>> ones(5)

ans =

     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
```

1.1.11. Define Diagonal Matrix.

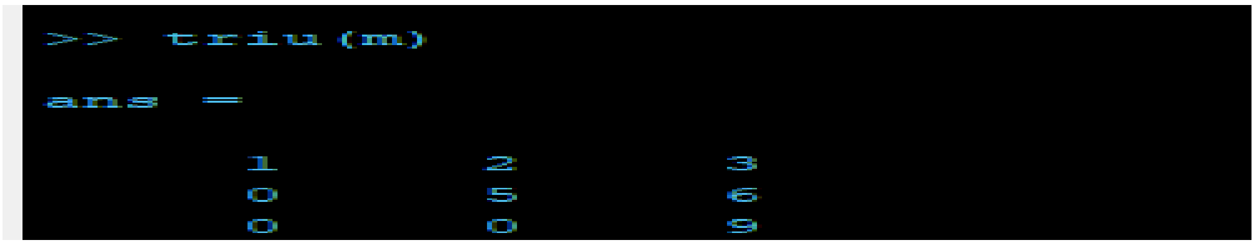
- A diagonal matrix is defined in MATLAB using diag(vector) where vector contains the diagonal elements.



```
>> diag(m)

ans =

     1
    88
    13
```

1.1.12. Define Upper Triangle Matrix.

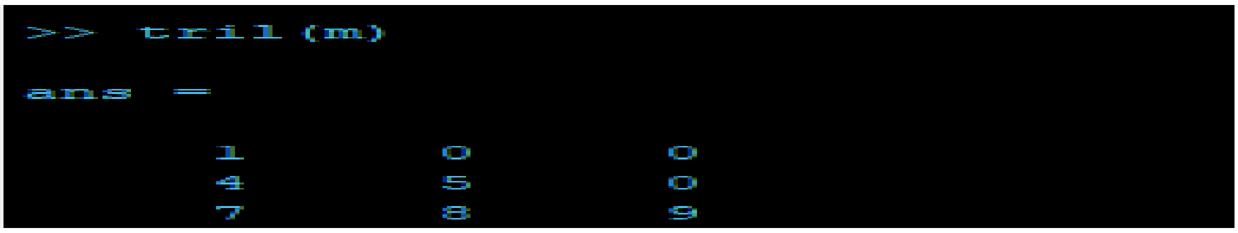
- An upper triangular matrix is defined in MATLAB using triu(matrix) which extracts the upper triangular part.



```
>> triu(m)

ans =

     1     2     3
     0     5     6
     0     0     9
```

1.1.13. Define Lower Triangle Matrix.

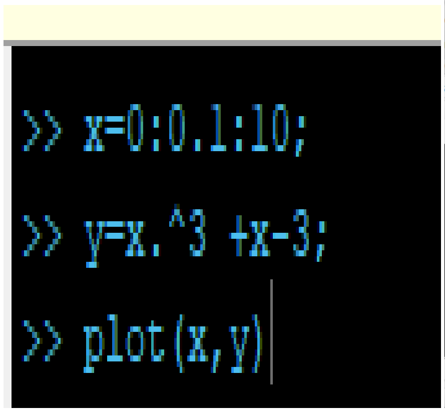
- A lower triangular matrix is defined in MATLAB using tril(matrix) which extracts the lower triangular part.



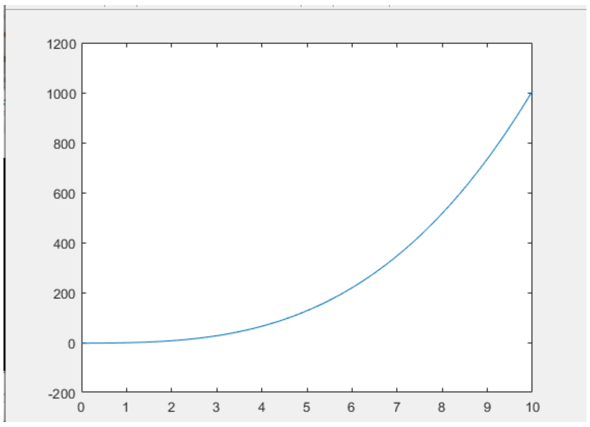
```
>> tril(m)

ans =

     1     0     0
     4     5     0
     7     8     9
```

1.1.14. Draw Function with a single line.

- A function plot is created in MATLAB using plot(x, y) where x and y are the input and output values, respectively.



```
>> x=0:0.1:10;
>> y=x.^3 +x-3;
>> plot(x,y)
```



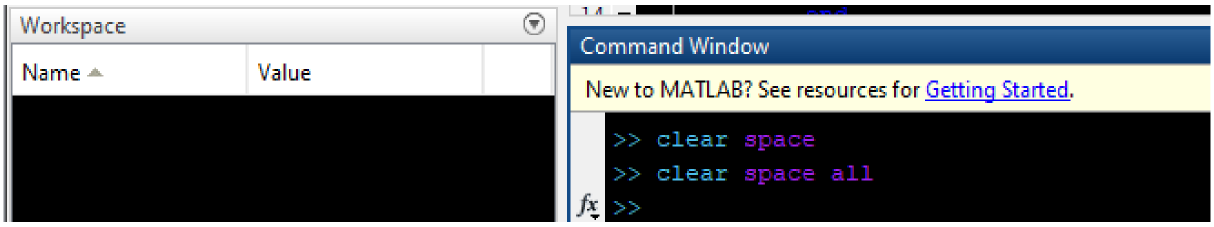
1.1.15. Clear Command Window.

- The command to clear the Command Window in MATLAB is: **clc**

```
>>
```

1.1.16. Clear Workspace.

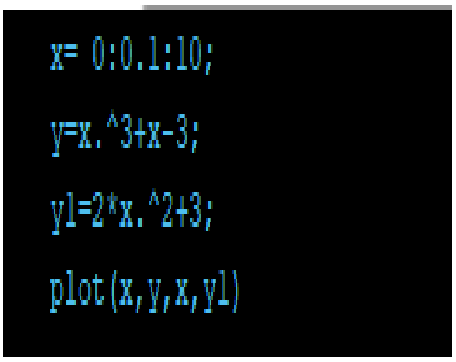
- The command to clear the workspace in MATLAB is: clear



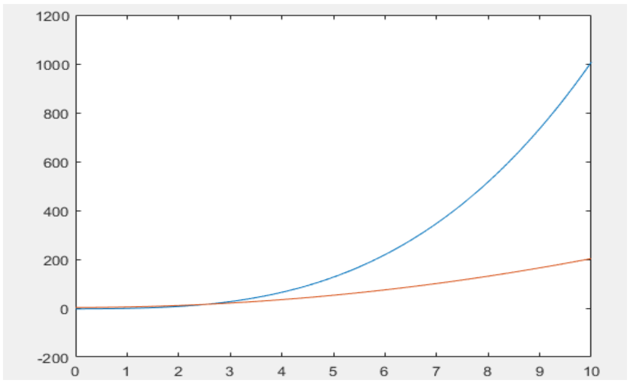
```
>> clear space
>> clear space all
>>
```

1.1.17. Plot two line on same palne.

- To plot two lines on the same plane in MATLAB, use the plot function with different data arrays.

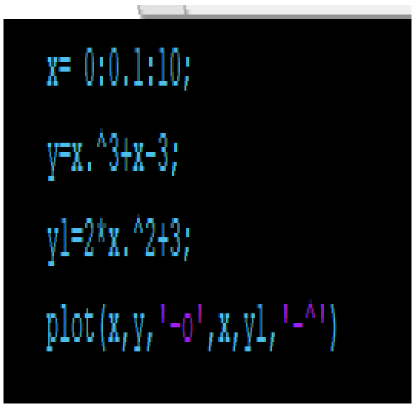


```
x= 0:0.1:10;
y=x.^3+x-3;
y1=2*x.^2+3;
plot(x,y,x,y1)
```

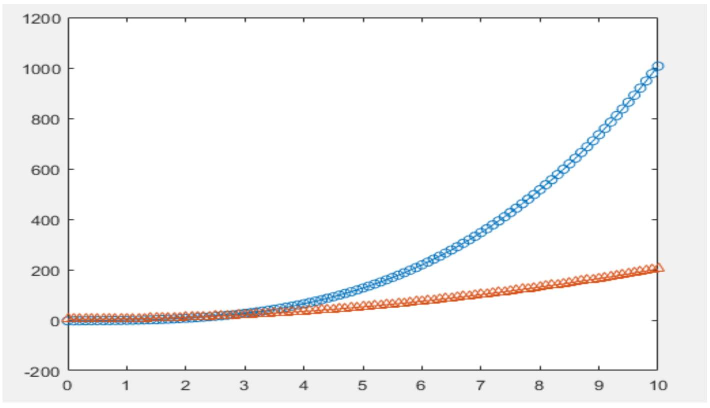


1.1.18. Colorlines graph.

- A **color lines graph** is a variation of a line plot where the color of the lines varies according to a third variable, typically related to data values.
- In MATLAB, the colorline function is used to plot lines with varying colors.

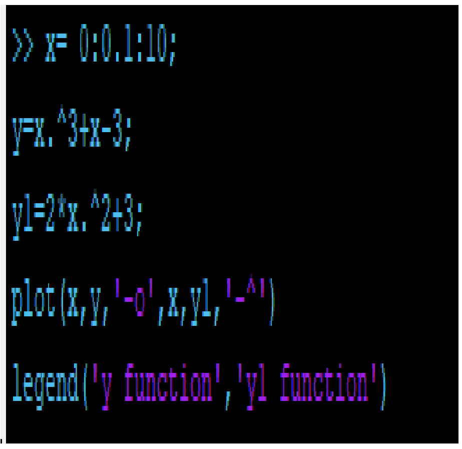


```
x= 0:0.1:10;
y=x.^3+x-3;
y1=2*x.^2+3;
plot(x,y,'-o',x,y1,'-^')
```

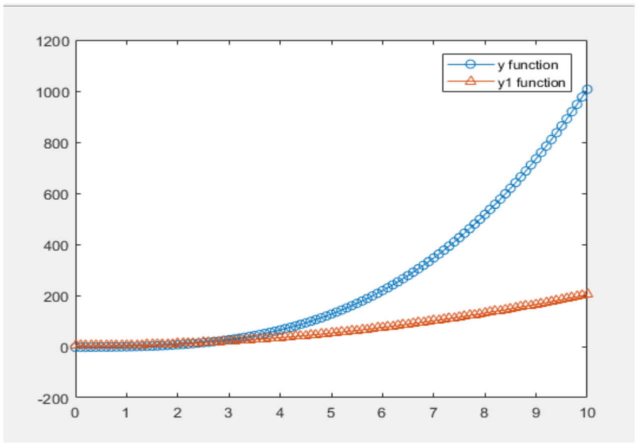


1.1.19. Mention each line function.

- The **legend function** is used to add a legend to a plot, which helps to label and distinguish multiple data series or plot elements.

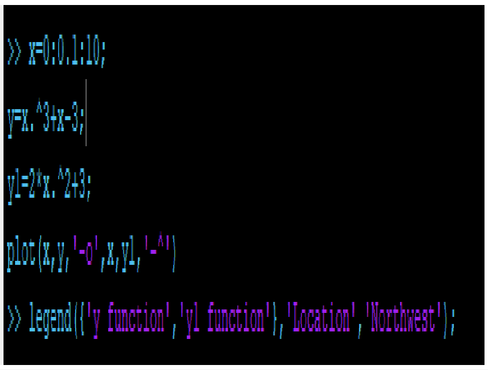


```
>> x= 0:0.1:10;
y=x.^3+x-3;
y1=2*x.^2+3;
plot(x,y,'-o',x,y1,'-^')
legend('y function','y1 function')
```

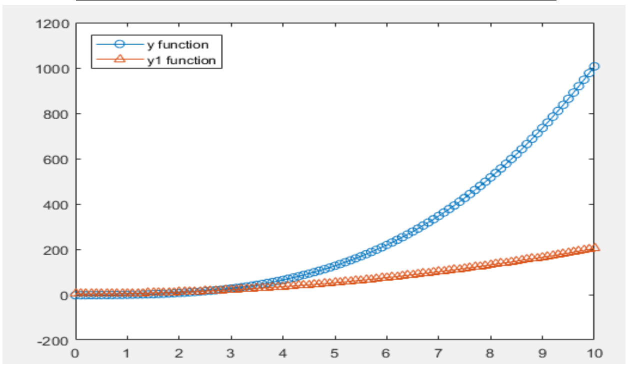


1.1.20. Change Location.

- Change the location of the legend using the legend function with additional parameters that specify the desired location. Legend('location', 'location_name')

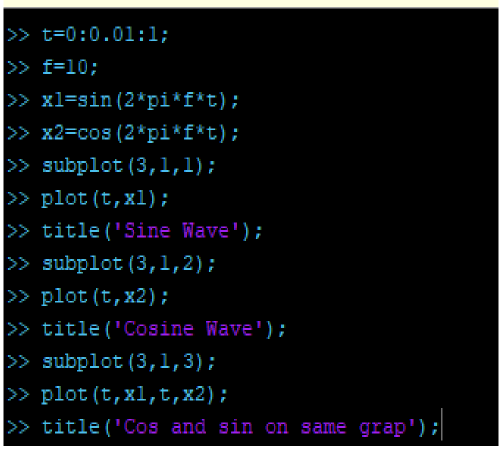


```
>> x=0:0.1:10;
y=x.^3+x-3;
y1=2*x.^2+3;
plot(x,y,'-o',x,y1,'-^')
>> legend({'y function','y1 function'},'Location','Northwest');
```

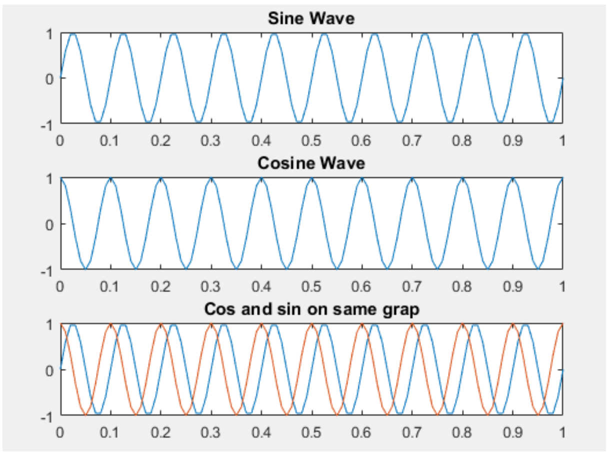


1.1.21. Sub plotting.

- **Subplots** allow you to create multiple plots within the same figure, organized into a grid layout.



```
>> t=0:0.01:1;
>> f=10;
>> x1=sin(2*pi*f*t);
>> x2=cos(2*pi*f*t);
>> subplot(3,1,1);
>> plot(t,x1);
>> title('Sine Wave');
>> subplot(3,1,2);
>> plot(t,x2);
>> title('Cosine Wave');
>> subplot(3,1,3);
>> plot(t,x1,t,x2);
>> title('Cos and sin on same grap');
```



WEEK#02

2. BISECTION METHOD

The bisection method is a root-finding algorithm that relies on the principle of dividing an interval into two equal parts and successively narrowing down the interval where the root lies. It is based on the Intermediate Value Theorem, which states that if a continuous function changes sign over an interval, at least one root exists within that interval. The bisection method involves repeatedly dividing the interval into two halves, calculating the midpoint, and selecting the subinterval that contains the root. The method converges linearly to the root, provided the function is continuous and the initial interval brackets the root. The formula for the bisection method is:

$$x_{new} = \frac{a+b}{2}$$

where a and b are the current lower and upper bounds of the interval, respectively, and x_{new} is the midpoint used to refine the interval.

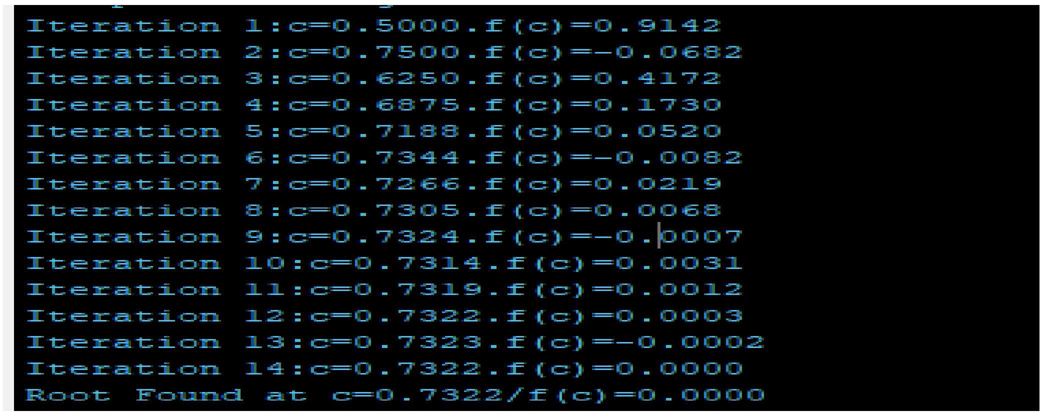
QUESTION:

2.1. Implementation of Bisection Method.

CODE:

```
disp('Script is running...')
f=@(x)2^x-5*x+2;
a=0;
b=1;
n=30;
e=0.0001;
if f(a)*f(b)< 0
    for i=1:n
        c=(a+b)/2;
        fc=f(c);
        fprintf('Iteration %d:c=%.4f.f(c)=%.4f\n',i,c,fc);
        if abs(fc)<e||abs(b-a)<e
            fprintf('Root Found at c=%.4f/f(c)=%.4f\n',c,fc);
        break;
        end
        if f(a)*fc<0
            b=c;
        else
            a=c;
        end
    end
else
    fprintf('No root lies between the interval[%f,%f]\n',a,b);
end
```

OUTPUT



```
Iteration 1:c=0.5000.f(c)=0.9142
Iteration 2:c=0.7500.f(c)=-0.0682
Iteration 3:c=0.6250.f(c)=0.4172
Iteration 4:c=0.6875.f(c)=0.1730
Iteration 5:c=0.7188.f(c)=0.0520
Iteration 6:c=0.7344.f(c)=-0.0082
Iteration 7:c=0.7266.f(c)=0.0219
Iteration 8:c=0.7305.f(c)=0.0068
Iteration 9:c=0.7324.f(c)=-0.0007
Iteration 10:c=0.7314.f(c)=0.0031
Iteration 11:c=0.7319.f(c)=0.0012
Iteration 12:c=0.7322.f(c)=0.0003
Iteration 13:c=0.7323.f(c)=-0.0002
Iteration 14:c=0.7322.f(c)=0.0000
Root Found at c=0.7322/f(c)=0.0000
```

WEEK#03

3. SECANT METHOD

The secant method is a numerical root-finding algorithm that approximates the root of a nonlinear equation by using the concept of secant lines. Unlike the bisection method, which requires repeated interval halving, the secant method uses two initial guesses and iteratively calculates new approximations by drawing secant lines to estimate the root. The method converges faster than the bisection method, especially when the initial guesses are close to the root. The formula for the secant method is:

$$x_{n+1} = x_n - \frac{f(x_n)(x_n - x_{n-1})}{f(x_n) - f(x_{n-1})}$$

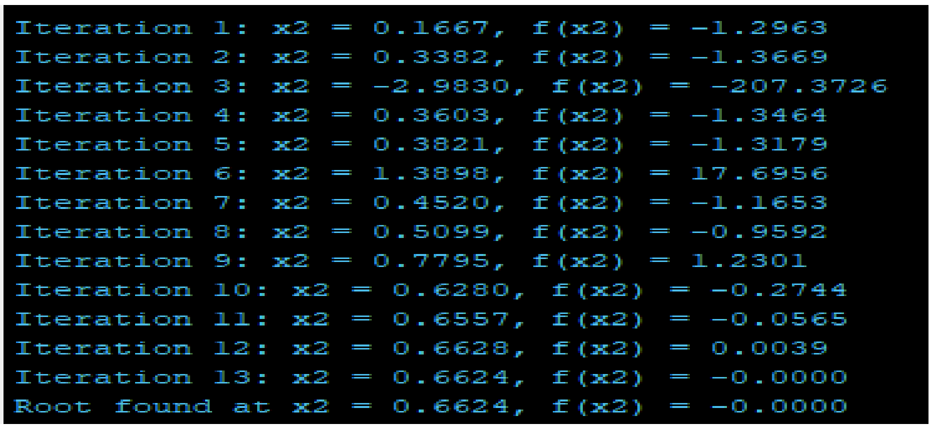
QUESTION:

3.1. Implementation of Secant Method.

CODE:

```
f = @(x) 8*x^3 - 2*x - 1; x0 = 0;
x1 = 1;
n = 30;
e = 0.0001;
for i = 1:n
if f(x1) - f(x0) == 0
fprintf('Division by zero error. f(x1) and f(x0) must not be equal.\n');
break;
end
x2 = x1 - f(x1) * (x1 - x0) / (f(x1) - f(x0));
fprintf('Iteration %d: x2 = %.4f, f(x2) = %.4f\n', i, x2, f(x2)); if abs(f(x2)) < e || abs(x2 - x1) < e
fprintf('Root found at x2 = %.4f, f(x2) = %.4f\n', x2, f(x2));
break;
end
x0 = x1;
x1 = x2;
end
if i == n
fprintf('Maximum iterations reached. Last approximation x2 = %.4f, f(x2) = %.4f\n', x2, f(x2));
end
```

OUTPUT:



```
Iteration 1: x2 = 0.1667, f(x2) = -1.2963
Iteration 2: x2 = 0.3382, f(x2) = -1.3669
Iteration 3: x2 = -2.9830, f(x2) = -207.3726
Iteration 4: x2 = 0.3603, f(x2) = -1.3464
Iteration 5: x2 = 0.3821, f(x2) = -1.3179
Iteration 6: x2 = 1.3898, f(x2) = 17.6956
Iteration 7: x2 = 0.4520, f(x2) = -1.1653
Iteration 8: x2 = 0.5099, f(x2) = -0.9592
Iteration 9: x2 = 0.7795, f(x2) = 1.2301
Iteration 10: x2 = 0.6280, f(x2) = -0.2744
Iteration 11: x2 = 0.6557, f(x2) = -0.0565
Iteration 12: x2 = 0.6628, f(x2) = 0.0039
Iteration 13: x2 = 0.6624, f(x2) = -0.0000
Root found at x2 = 0.6624, f(x2) = -0.0000
```

WEEK#04

4. N-R SYSTEM OF LINEAR EQUATION IMPLEMENTATION

The Newton-Raphson (N-R) method is an iterative technique used to find the roots of nonlinear equations. It relies on the concept of tangents, where a tangent is drawn to the curve of the function, and the intersection of this tangent with the x-axis gives the next approximation of the root. The method is efficient and converges quadratically if the initial guess is close to the actual root. The Newton-Raphson formula is derived from the linear approximation of a function:

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

The variables represent:

- $x(n)$: $Current\ approximation$

- $f(x(n))$: $Function\ value\ at\ the\ current\ approximation$

- $x(n-1)$: $Previous\ approximation$

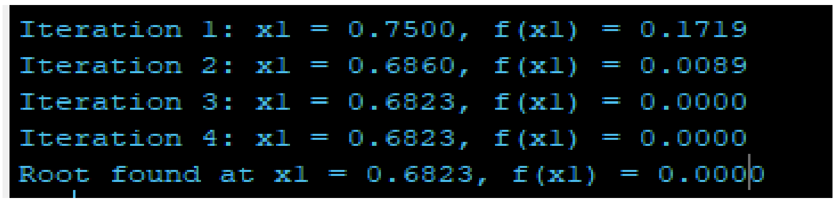
QUESTION:

4.1. Implementation of Newton Raphson(N-R) Method.

CODE:

```
f = @(x) x^3 + x-1;
df = @(x) 3*x^2+1;
x0 = 1;
n = 30;
e = 0.0000001;
for i = 1:n
if df(x0) == 0
fprintf('Division by zero error. Derivative must not be zero.\n');
break;
end
x1 = x0 - f(x0) / df(x0);
fprintf('Iteration %d: x1 = %.4f, f(x1) = %.4f\n', i, x1, f(x1));
if abs(f(x1)) < e || abs(x1 - x0) < e
fprintf('Root found at x1 = %.4f, f(x1) = %.4f\n', x1, f(x1));
break;
end
x0 = x1;
end
if i == n
fprintf('Maximum iterations reached. Last approximation x1 = %.4f, f(x1) = %.4f\n', x1, f(x1));
end
```

OUTPUT



```
Iteration 1: x1 = 0.7500, f(x1) = 0.1719
Iteration 2: x1 = 0.6860, f(x1) = 0.0089
Iteration 3: x1 = 0.6823, f(x1) = 0.0000
Iteration 4: x1 = 0.6823, f(x1) = 0.0000
Root found at x1 = 0.6823, f(x1) = 0.0000
```

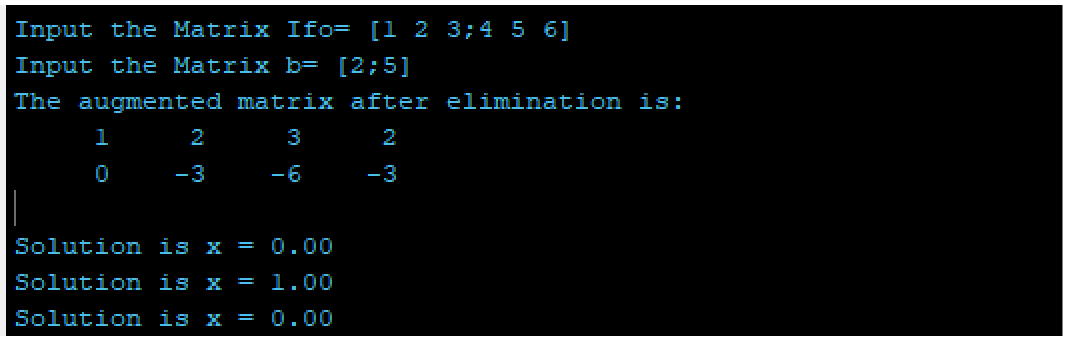
QUESTION

4.2. Implementation of System of linear Equations.

CODE:

```
Ifo = input('Input the Matrix Ifo= ');
b = input('Input the Matrix b= ');
A = [Ifo b];
n = size(A, 1);
for i = 1:n
    for j = i+1:n
        key = A(j, i) / A(i, i);
        A(j, :) = A(j, :) - key * A(i, :);
    end
end
 disp('The augmented matrix after elimination is:');
disp(A);
x = zeros(1, size(Ifo, 2));
for i = n:-1:1
    hg = sum(A(i, i+1:end-1) .* x(i+1:end));
    x(i) = (A(i, end) - hg) / A(i, i);
end
fprintf('Solution is x = %0.2f \n', x);
```

OUTPUT:



```
Input the Matrix Ifo= [1 2 3;4 5 6]
Input the Matrix b= [2;5]
The augmented matrix after elimination is:
     1     2     3     2
     0    -3    -6    -3


Solution is x = 0.00
Solution is x = 1.00
Solution is x = 0.00
```

WEEK#05

5. LU RECOMPOSITION METHOD

The LU decomposition method is a numerical technique used to solve systems of linear equations efficiently. It decomposes a square matrix A into the product of a lower triangular matrix L The LU decomposition of a matrix A is expressed as:

$$\boldsymbol{A} = \boldsymbol{LU}$$

where:

$$L = Lower\ triangular\ matrix$$

$$U = Upper\ triangular\ matrix$$

To solve the linear system $Ax = b$, we can use the LU decomposition to simplify the equation:

$$LUx = b$$

This equation can be broken down into two triangular systems:

$$Ly = b\ (forward\ substitution)$$

$$Ux = y\ (backward\ substitution)$$

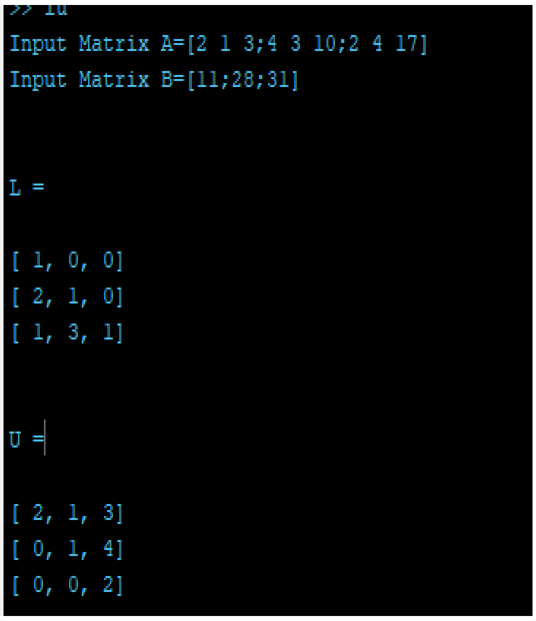
<u>QUESTION:</u>

5.1. Implementation of LU using Dolittle's Decomposition Method.

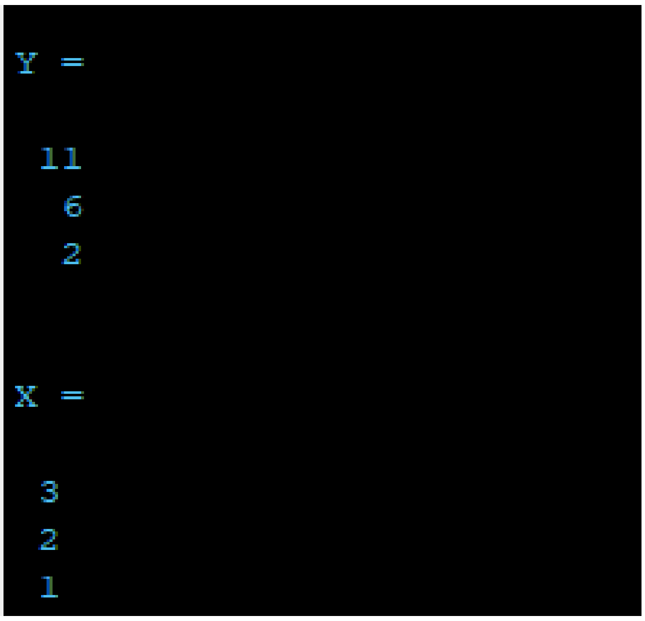
<u>CODE:</u>

```
A=input ( 'Input Matrix A=' );
b=input ( 'Input Matrix B=' );
[L U]=lu(sym (A))
adjoint(L);
det(L);
inverseL =adjoint(L)/det(L);
Y=inverseL*b
adjoint(U);
det(U);
inverseU=adjoint(U)/det(U);
X=inverseU*Y
```

OUTPUT:



```
>> lu
Input Matrix A=[2 1 3;4 3 10;2 4 17]
Input Matrix B=[11;28;31]


L =

[ 1, 0, 0]
[ 2, 1, 0]
[ 1, 3, 1]


U =

[ 2, 1, 3]
[ 0, 1, 4]
[ 0, 0, 2]
```



```
Y =

  11
   6
   2


X =

  3
  2
  1
```

QUESTION:

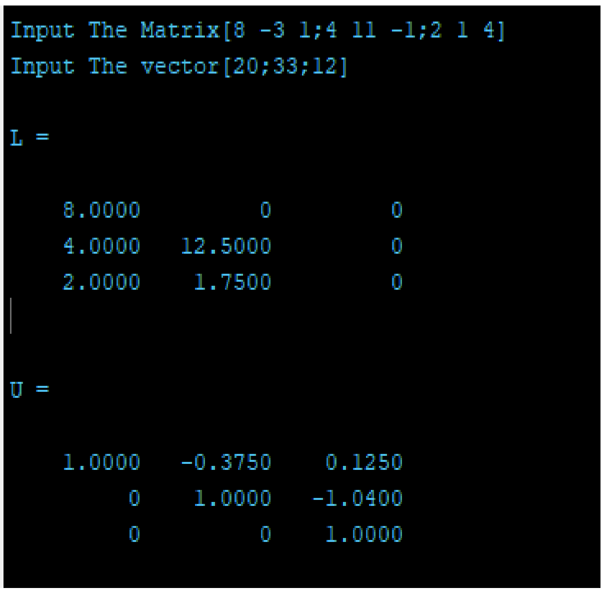
5.2. Implementation of LU using Crout's Decomposition Method.

CODE:

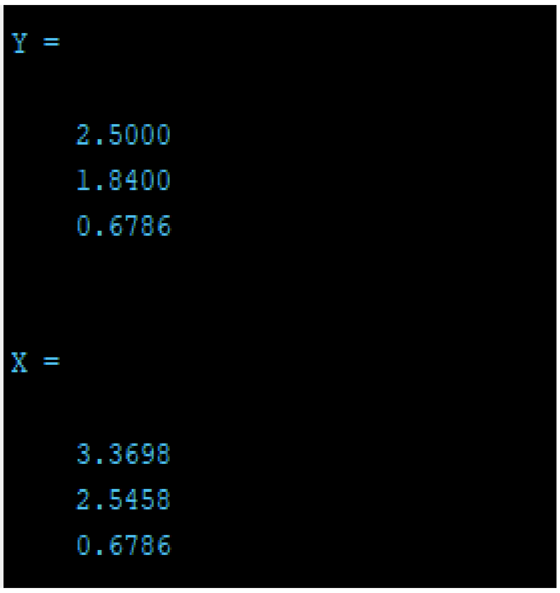
```
A=input('Input The Matrix');
b=input('Input The vector');
N=length(b);
L=zeros(N,N);
U=zeros(N,N);
for a=1:N
   U(a,a)=1;
end
L(:,1)=A(:,1);
U(1,:)=A(1,:)/L(1,1);
for i=2:N
   for k=i:N
      L(k,i)=A(k,i) - L(k,1:i-1)*U(1:i-1,i);
   end
   for j=i+1:N
      U(i,j)=A(i,j) - L(i,1:i-1)*U(1:i-1,j)/L(i,i);
   end
   L,U
   Y=zeros(N,1);
   Y(1)=b(1)/L(1,1);
   for k=2:N
      Y(k)=(b(k) - L(k,1:k-1)*Y(1:k-1))/L(k,k);
   end
   Y
   X=zeros(N,1);
   X(N)=Y(N)/U(N,N);
   for k=N-1:-1:1
      X(k)=(Y(k) - U(k,k+1:N)*X(k+1:N))/U(k,k);
```

```
    end
    X
end
```

OUTPUT:



```
Input The Matrix[8 -3 1;4 11 -1;2 1 4]
Input The vector[20;33;12]

L =

    8.0000         0         0
    4.0000   12.5000         0
    2.0000    1.7500         0


U =

    1.0000   -0.3750    0.1250
         0    1.0000   -1.0400
         0         0    1.0000
```



```
Y =

    2.5000
    1.8400
    0.6786


X =

    3.3698
    2.5458
    0.6786
```

WEEK#06

6. GAUSS JACOBI

The Gauss-Jacobi method is an iterative technique for solving systems of linear equations of the form Ax = b, where A is a square matrix and b is a constant vector.

Key Principles

1. Decompose the system into simpler equations.
2. Update each variable successively using previously computed values.

Gauss-Jacobi Formula

Given the system of linear equations Ax = b, the Gauss-Jacobi iteration formula is:

$$xi(k+1) \;=\; 1/Aii \;*\; (bi \;-\; \sum(j \neq i)\, Aij \;*\; xj(k))$$

Variables

- $xi(k+1)$: $New\ estimate\ of\ the\ ith\ variable.$

$-\ Aii$: $Diagonal\ element\ of\ matrix\ A.$

$-\ Aij$: $Off - diagonal\ elements\ of\ matrix\ A.$

$-\ xj(k)$: $Values\ of\ other\ variables\ from\ the\ previous\ iteration.$

$-\ bi$: $ith\ element\ of\ vector\ b.$

This method requires an initial guess for the solution and iteratively updates each variable until convergence.

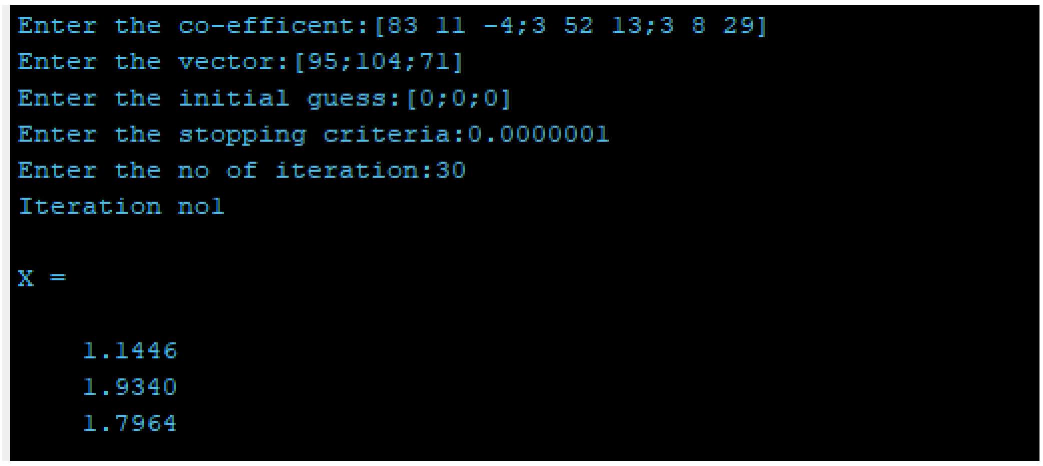
QUESTION:

6.1. Implementation of Gauss Jacobi Method.

CODE:

```
A=input('Enter the co-efficent:');
B=input('Enter the vector:');
P=input('Enter the initial guess:');
e=input('Enter the stopping criteria:');
n=input('Enter the no of iteration:');
N=length(B);
X=zeros(N,1);
for j=1:n
   for i=1:N
      X(i)=(B(i)/A(i,i))-(A(i,[1:i-1,i+1:N]))*P([1:i-1,i+1:N])/A(i,i);
      P(i)=X(i);
   end
   fprintf('Iteration no%d \n',j);
   X
   if abs(X-P)<e
      break
   end
   P=X;
end
```

OUTPUT:



```
Enter the co-efficent:[83 11 -4;3 52 13;3 8 29]
Enter the vector:[95;104;71]
Enter the initial guess:[0;0;0]
Enter the stopping criteria:0.0000001
Enter the no of iteration:30
Iteration no1

X =

    1.1446
    1.9340
    1.7964
```

WEEK#07

7. GAUSS SEIDLE

The Gauss-Seidel method is an iterative technique for solving systems of linear equations of the form Ax = b, where A is a square matrix and b is a constant vector. This method is an improvement over the Gauss-Jacobi method, as it uses updated values from the current iteration to compute subsequent values, leading to faster convergence.

Given the system of linear equations Ax = b, the Gauss-Seidel iteration formula is:

$$xi(k+1) = 1/Aii * (bi - \sum(j = 1\, to\, i-1)\, Aij * xj(k+1) - \sum(j = i+1\, to\, n)\, Aij * xj(k))$$

Variables

$- xi(k+1)$: $Updated\ value\ of\ the\ ith\ variable\ at\ iteration\ k+1.$

$- Aii$: $Diagonal\ element\ of\ matrix\ A.$

$- \ Aij$: $Off - diagonal\ elements\ of\ matrix\ A.$

$- \ xj(k)\ and\ xj(k+1)$: $Current\ and\ updated\ values\ of\ other\ variables.$

$- bi$: $ith\ element\ of\ vector\ b.$

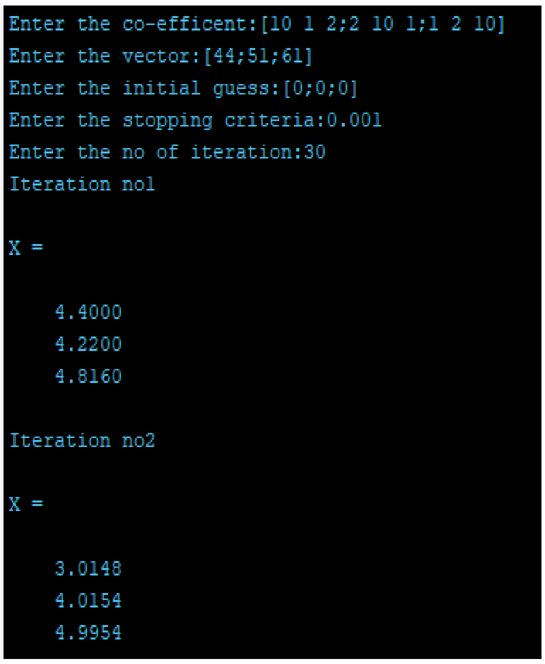
QUESTION:

7.1. Implementation of Gauss Seidle Method.

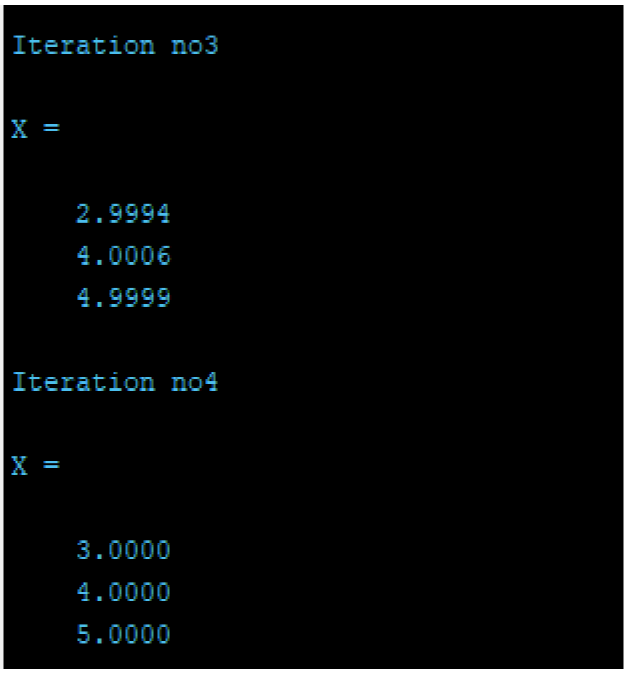
CODE:

```
A=input('Enter the co-efficent:');
B=input('Enter the vector:');
P=input('Enter the initial guess:');
e=input('Enter the stopping criteria:');
n=input('Enter the no of iteration:');
N=length(B);
X=zeros(N,1);
Y=zeros(N,1);
for j=1:n
   for i=1:N
      X(i)=(B(i)/A(i,i))-(A(i,[1:i-1,i+1:N]))*P([1:i-1,i+1:N])/A(i,i);
      P(i)=X(i);
   end
   fprintf('Iteration no%d \n',j);
   X
   if abs(Y-X)<e
      break
   end
   Y=X;
end
```

OUTPUT:

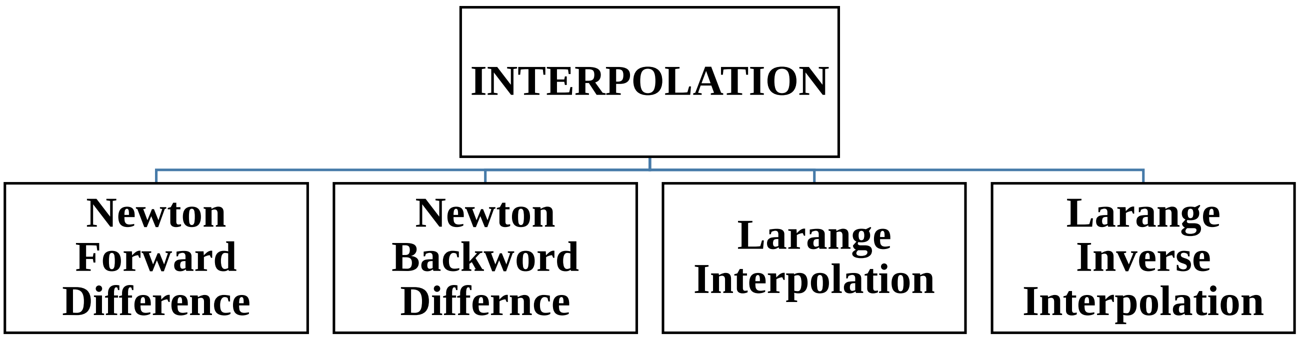


```
Enter the co-efficent:[10 1 2;2 10 1;1 2 10]
Enter the vector:[44;51;61]
Enter the initial guess:[0;0;0]
Enter the stopping criteria:0.001
Enter the no of iteration:30
Iteration no1

X =

    4.4000
    4.2200
    4.8160

Iteration no2

X =

    3.0148
    4.0154
    4.9954
```



```
Iteration no3

X =

    2.9994
    4.0006
    4.9999

Iteration no4

X =

    3.0000
    4.0000
    5.0000
```

WEEK#08


INTERPOLATION
Newton Forward Difference
Newton Backword Differnce
Larange Interpolation
Larange Inverse Interpolation

8. NEWTON FORWARD DIFFERENCE

The Newton Forward Difference Method is a numerical technique used to estimate the value of a function at a point using forward differences. This method is particularly useful for solving problems involving interpolation:

The formula for Newton Forward Difference is:

$$f(x) = f(x_0) + (x - x_0) * f'(x_0) + ((x - x_0) * (x - x_1)) \,/\, 2! * f''(x_0) + ((x - x_0) * (x - x_1) * (x - x_2)) \,/\, 3! * f'''(x_0) + \ldots$$

Variables

$- f(x)\ is\ the\ estimated\ value\ of\ the\ function\ at\ x.$

$- x_0, x_1, x_2, \ldots\ are\ the\ given\ data\ points.$

$- f'(x_0), f''(x_0), \ldots\ are\ the\ forward\ differences\ of\ the\ function.$

Newton Forward Difference Formula

$$f(x) = f(x_0) + \Delta x * f'(x_0) + (\Delta x^2) \,/\, 2! * f''(x_0) + (\Delta x^3) \,/\, 3! * f'''(x_0) + \ldots$$

Variables

$-\Delta x = x - x_0\ is\ the\ step\ size\ between\ consecutive\ points.$

$- f'(x_0), f''(x_0), f'''(x_0), \ldots\ are\ computed\ using\ forward\ differences.$

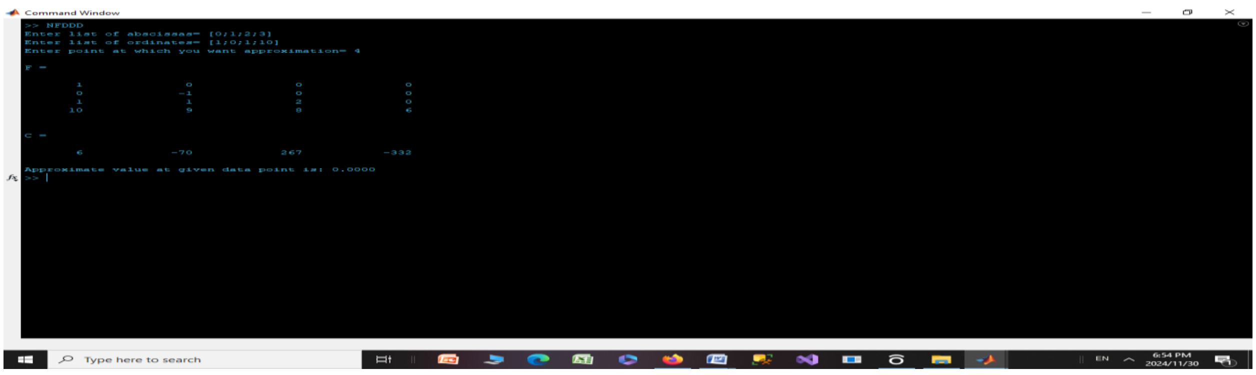
QUESTION:

8.1. Implementation of Newton Forward Difference.

CODE:

```
x = input('Enter list of abscissas= ');
y = input('Enter list of ordinates= ');
PO = input('Enter point at which you want approximation= ');
n = length(x);
h = x(2) - x(1);
F = zeros(n, n);
F(:, 1) = y;
for j = 2:n
   for i = j:n
      F(i, j) = F(i, j-1) - F(i-1, j-1);
   end
end
F
C = F(n, n);
for k = n-1:-1:1
   P = poly(x(n))/h;
   P(2) = P(2) - (k-1);
   C = conv(C, P) ;
   m = length(C);
   C(m) = C(m) + F(k, k);
end
C
A = polyval(C, PO);
fprintf('Approximate value at given data point is: %.4f\n', A);
X = linspace(x(1), x(n), 100);
Y = polyval(C, X);
plot(X, Y, 'r');
hold on;
plot(x, y, 'o');
```

OUTPUT:



```
>> NFDDD
Enter list of abscissas= [0;1;2;3]
Enter list of ordinates= [1;0;1;10]
Enter point at which you want approximation= 4

F =

     1            0            0            0
     0           -1            0            0
     1            1            2            0
    10            9            8            6


C =

     6          -70          267         -332

Approximate value at given data point is: 0.0000
>> 
```

WEEK#9

9. NEWTON BACKWARD

The Newton Backward Difference Method is a numerical technique used to estimate the value of a function at a point, especially when data points are provided at equally spaced intervals. This method is useful when the data is available only at the end of the interval, and we want to predict values at points closer to the start.

The Newton Backward Difference formula is based on the principle of backward differences:

$$f(x) = f(xn) + (x - xn) / 1! * f'(xn) + (x - xn)(x - xn - 1) / 2! * f''(xn) + (x - xn)(x - xn - 1)(x - xn - 2) / 3! * f'''(xn) + \ldots$$

Variables

$- f(x) \text{ is the estimated value at } x.$

$- xn \text{ is the last data point.}$

$- xn - 1, xn - 2, \ldots \text{ are earlier data points.}$

$- f'(xn), f''(xn), f'''(xn) \text{ are the backward differences.}$

Newton Backward Difference Formula

$$f(x) = f(xn) + \Delta x * [f(xn) - f(xn - 1)]/1! + (x - xn)(x - xn - 1)/2! * [f(xn) - 2f(xn - 1) + f(xn - 2)]/2! + \ldots$$

Variables

$- \Delta x = x - xn \text{ is the step size between consecutive points.}$

$- f(xn), f(xn - 1), f(xn - 2) \text{ are the function values at those points.}$

QUESTION:

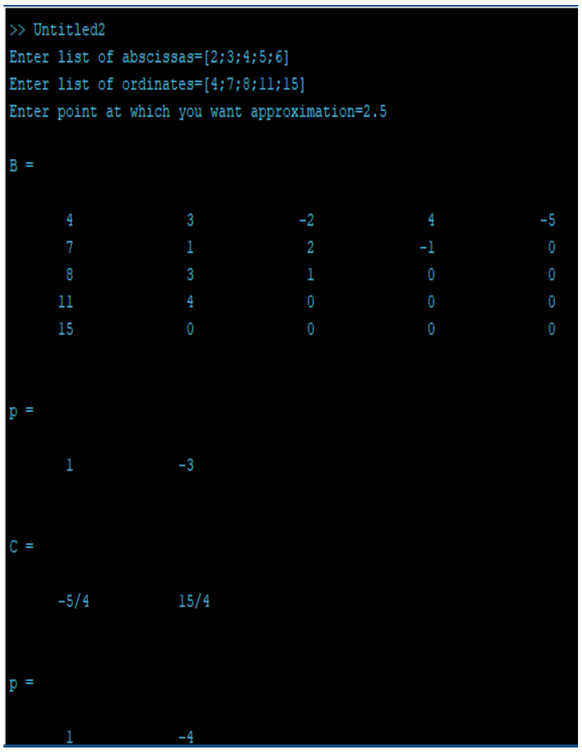
9.1. Implementation of Newton Backward Difference.

CODE:

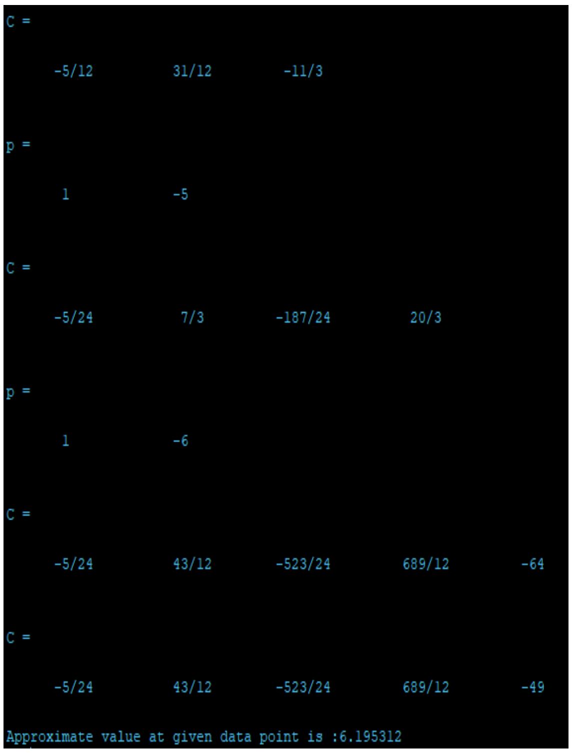
```
x =input('Enter list of abscissas=');
y =input('Enter list of ordinates=');
PO=input('Enter point at which you want approximation=');
n=length(x);
h=x(2)-x(1);
B=zeros(n,n);
B(:,1)=y;
for j=2:n
   for i=1:n-j+1
      B(i,j)=B(i+1,j-1)-B(i,j-1);
   end
end
```

```
B
C=B(1,n);
for k=n-1:-1:1
    p=poly(x(n))/h;
    p(2) = p(2)+ (k-1);
    p
   C= conv(C,p)/k
    m=length(C);
    C(m)=C(m)+B(n-k+1,k);
end
C
A=polyval(C,PO);
fprintf('Approximate value at given data point is :%4f\n',A);
x=linspace(x(1),x(n),100);
y=polyval(C,x);
plot(x,y,'r')
hold on
plot(x,y,'o')
```

OUTPUT:



```
>> Untitled2
Enter list of abscissas=[2;3;4;5;6]
Enter list of ordinates=[4;7;8;11;15]
Enter point at which you want approximation=2.5

B =

      4          3          -2          4         -5
      7          1           2         -1          0
      8          3           1          0          0
     11          4           0          0          0
     15          0           0          0          0


p =

      1         -3


C =

     -5/4       15/4


p =

      1         -4
```



```
C =

     -5/12        31/12        -11/3


p =

      1           -5


C =

     -5/24         7/3        -187/24        20/3


p =

      1           -6


C =

     -5/24        43/12       -523/24       689/12       -64


C =

     -5/24        43/12       -523/24       689/12       -49

Approximate value at given data point is :6.195312
```

WEEK#10

10.LAGRANGE INTERPOLATION

Lagrange Interpolation is a polynomial-based method used to estimate values of a function at points where direct computation is difficult or unknown. It relies on constructing a polynomial that exactly passes through a set of given data points. The Lagrange polynomial is formed by using the concept of Lagrange basis functions, which ensures that each data point contributes to the polynomial without affecting the others.

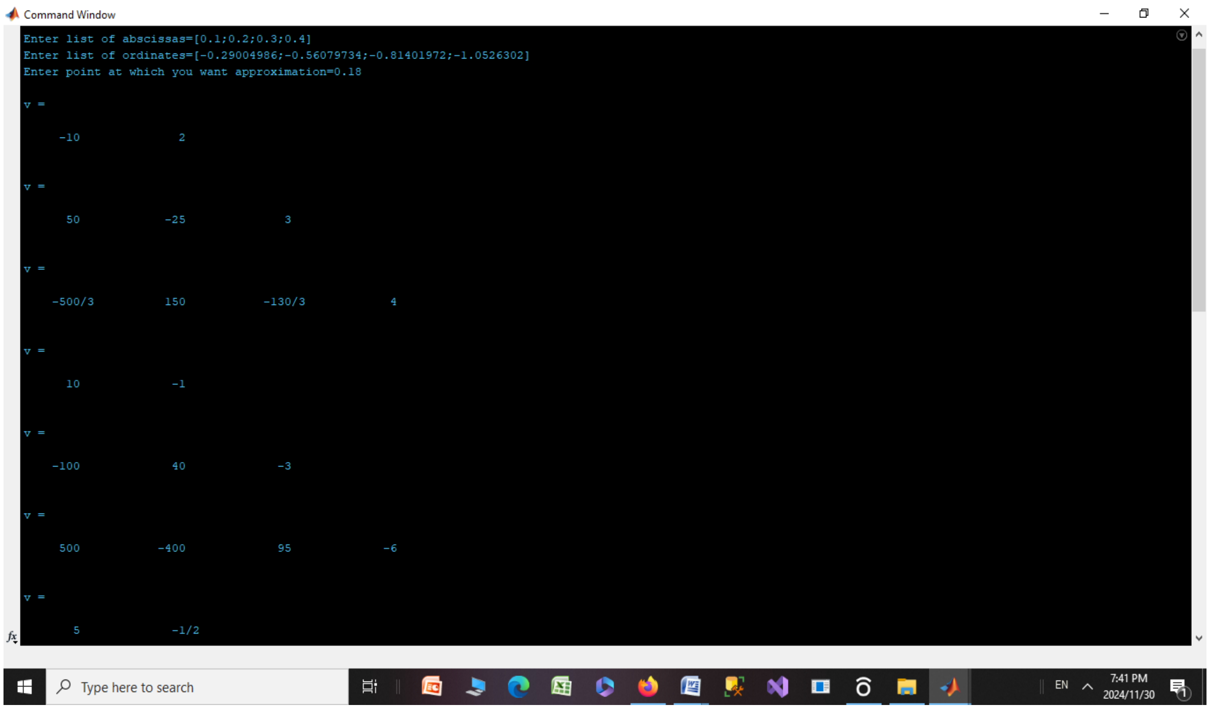
QUESTION:

10.1. Implementation of Lagrange’s Interpolation Method.

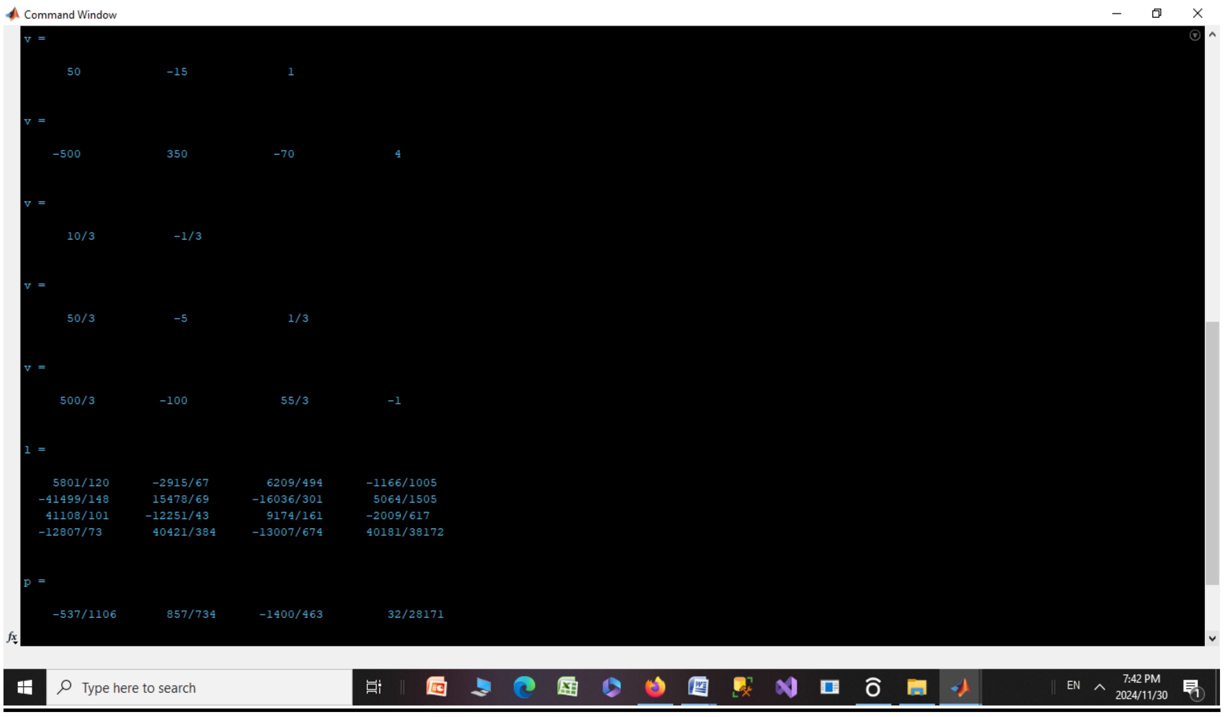
CODE:

```
x =input('Enter list of abscissas=');
y =input('Enter list of ordinates=');
PO=input('Enter point at which you want approximation=');
n=length(x);
l=zeros(n,n);
for i=1:n
   v=1;
   for j=1:n
      if i~=j
         v=conv(v,poly(x(j)))/(x(i)-x(j))
      end
   end
   l(i,:)=v*y(i);
end
l
p=sum(l)
a=polyval(p,PO)
fprintf('Approximate value of givven data poinr is =%.4f\n',a);
x=linspace(x(1),x(n),100);
y=polyval(p,x);
plot(x,y,'r')
hold on
plot(x,y,'o')
hold on
plot(PO,a,'g*')
```

OUTPUT:



```
Enter list of abscissas=[0.1;0.2;0.3;0.4]
Enter list of ordinates=[-0.29004986;-0.56079734;-0.81401972;-1.0526302]
Enter point at which you want approximation=0.18

v =

    -10              2


v =

     50            -25               3


v =

   -500/3          150            -130/3            4


v =

     10             -1


v =

   -100             40              -3


v =

    500           -400              95             -6


v =

      5             -1/2
```



```
v =

     50            -15               1


v =

   -500            350             -70              4


v =

     10/3           -1/3


v =

     50/3           -5               1/3


v =

    500/3         -100              55/3            -1


l =

     5801/120      -2915/67        6209/494      -1166/1005
   -41499/148      15478/69      -16036/301       5064/1505
    41108/101     -12251/43        9174/161      -2009/617
   -12807/73       40421/384     -13007/674      40181/38172


p =

    -537/1106       857/734       -1400/463         32/28171
```

11.LAGRANGE INVERSE INTERPOLATION

QUESTION:

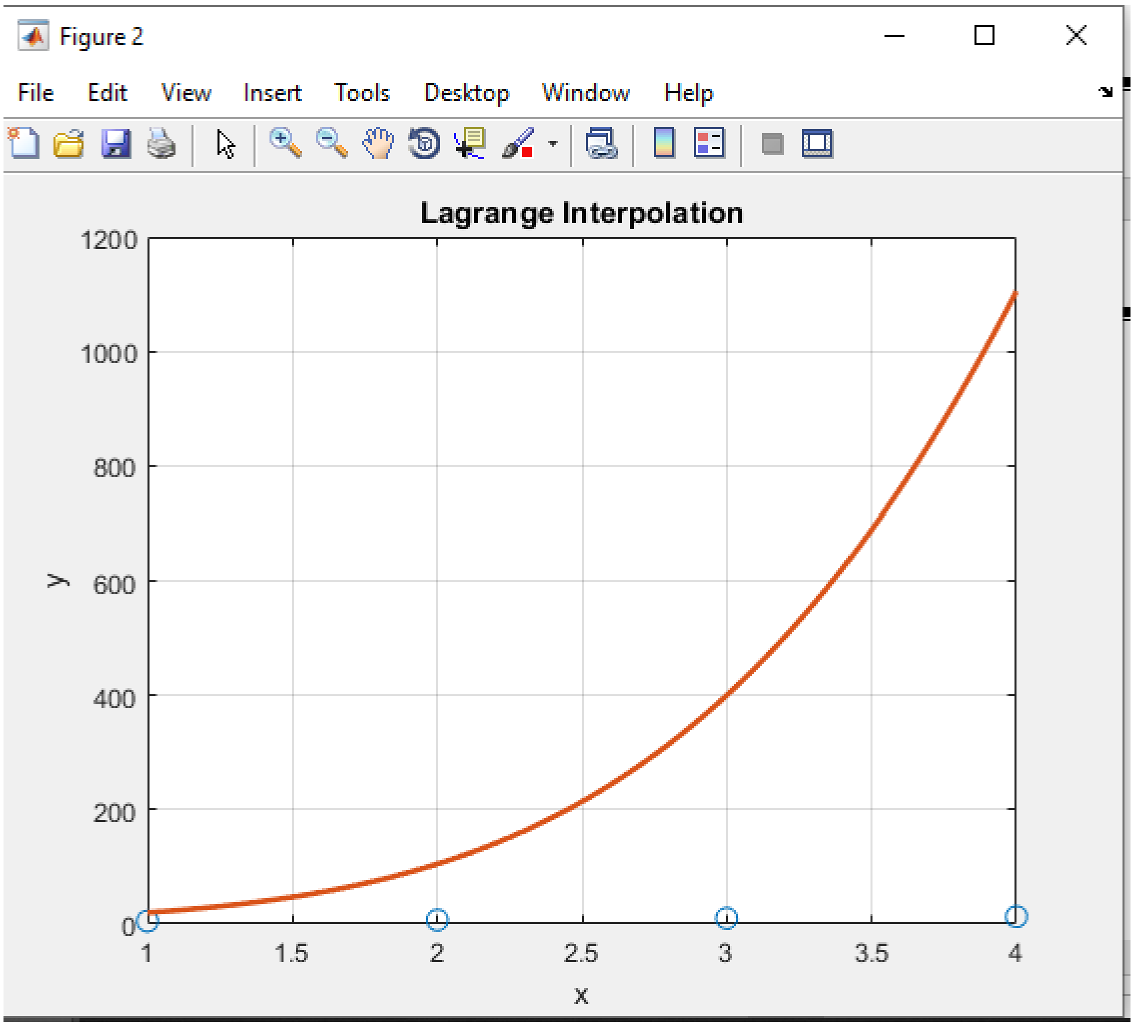
11.1. Implementation of Lagrange Inverse Interpolation.

CODE:

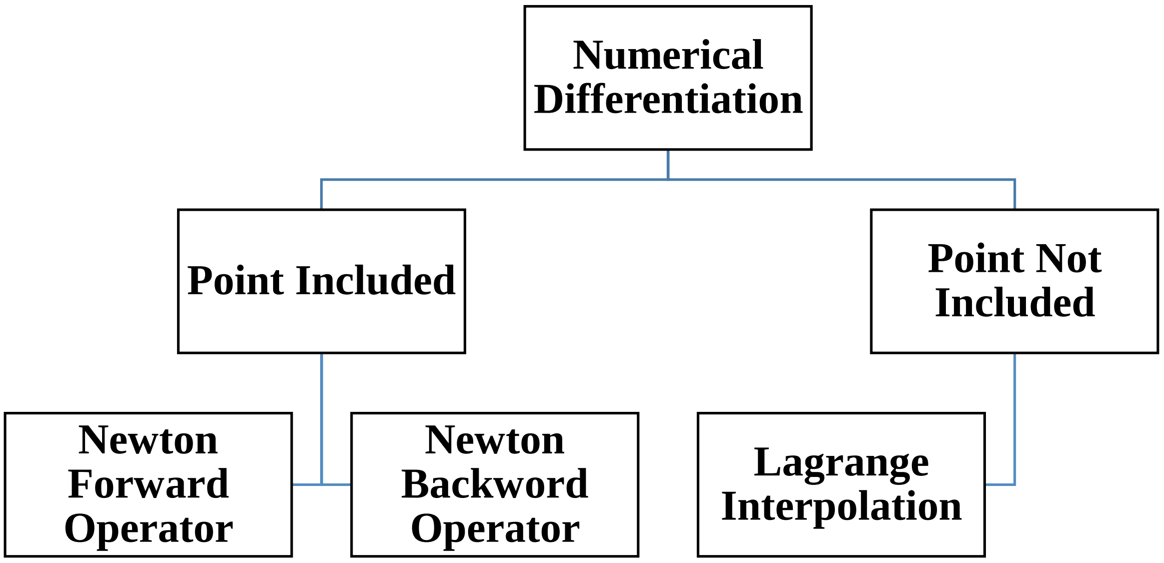
```
% Define the data points (x and y)
x = [1, 2, 3, 4];    % x-coordinates of the data points
y = [5, 7, 10, 13];  % y-coordinates of the data points

% Function for Lagrange Interpolation
n = length(x);        % Number of points
L = zeros(1, n);       % Initialize Lagrange basis polynomial
interp_poly = 0;       % Initialize the interpolating polynomial

% Lagrange basis polynomial
for i = 1:n
    L = ones(1, n);
    for j = 1:n
        if i ~= j
            L = conv(L, poly(x(j))) / (x(i) - x(j));
        end
    end
    interp_poly = interp_poly + y(i) * L;
end

% Display the interpolating polynomial
disp('Interpolating polynomial:')
disp(poly2sym(interp_poly))

% Evaluate the interpolation at some points
xx = linspace(min(x), max(x), 100);  % Generate more points for smooth plot
yy = polyval(interp_poly, xx);        % Interpolated values

% Plot the original data points and the interpolating polynomial
figure;
plot(x, y, 'o', 'MarkerSize', 8, 'DisplayName', 'Data Points');
hold on;
plot(xx, yy, '-', 'LineWidth', 2, 'DisplayName', 'Lagrange Interpolation');
legend;
xlabel('x');
ylabel('y');
title('Lagrange Interpolation');
grid on;
```

OUTPUT:

```
>> inverse
Interpolating polynomial:
- x^6/6 + (4*x^5)/3 + 5*x^3 + (31*x^2)/6 + (11*x)/3 + 5
```



WEEK#11



12.NEWTON FORWARD OPERATOR

The Newton Forward Operator is a method used for interpolation, particularly when dealing with equally spaced data points. It is commonly employed to estimate the values of a function at points within the interval of known data points. The method is based on polynomial interpolation, specifically using the divided differences concept.

Formula:

$$y(x) = y_0 + (x - x_0) / 1! * f'(x_0) + (x - x_0)(x - x_1) / 2! * f''(x_0) + \ldots + (x - x_0)(x - x_1) \ldots (x - x_{n-1}) / n! * f^{(n)}(x_0)$$

Variables:

$- x$: *Point at which interpolation is desired*

$- x_0, x_1, \ldots, x_n$: *Equally spaced data points*

$- y(x)$: *Interpolated value of the function at x*

$- f'(x_0), f''(x_0), \ldots, f^{(n)}(x_0)$: *Forward differences of the function*

<u>QUESTION:</u>

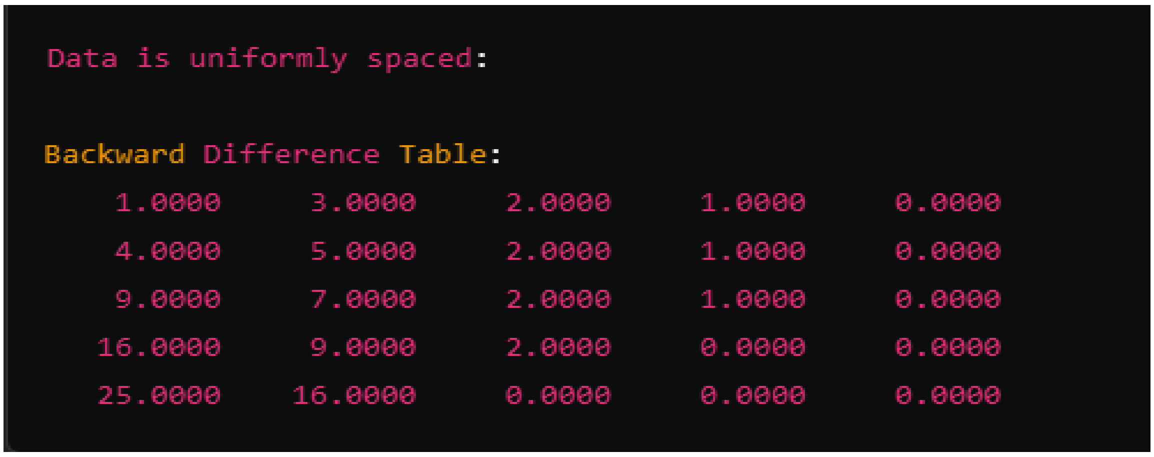
12.1. Implementation of Newton Backward Operator.

<u>CODE:</u>

```
% Input abscissas (x) and ordinates (y)
x = input('Enter list of abscissas: ');
```

```
y = input('Enter list of ordinates: ');

% Number of data points
n = length(x);

% Check if data is uniformly spaced
h = x(2) - x(1);
count = 0;

for a = 2 : n - 1
    b = x(a + 1) - x(a);
    if b ~= h
        count = count + 1;
    end
end

if count == 0
    disp('Data is uniformly spaced:');
else
    disp('Data is not uniformly spaced:');
end

% Initialize Backward Difference Table
B = zeros(n, n);
B(:, 1) = y;

% Compute Backward Differences
for j = 2 : n
    for i = 1 : n - j + 1
        B(i, j) = B(i + 1, j - 1) - B(i, j - 1);
    end
end

% Display the Backward Difference Table
disp('Backward Difference Table:');
disp(B);
```

OUTPUT:



```
Data is uniformly spaced:

Backward Difference Table:
      1.0000      3.0000      2.0000      1.0000      0.0000
      4.0000      5.0000      2.0000      1.0000      0.0000
      9.0000      7.0000      2.0000      1.0000      0.0000
     16.0000      9.0000      2.0000      0.0000      0.0000
     25.0000     16.0000      0.0000      0.0000      0.0000
```

13.NEWTON BACKWARD OPERATOR

The **Newton Backward Operator** is used to interpolate values of a function at points that are not available but lie within a set of equally spaced data points. It is particularly useful when the data points are given in reverse order or when interpolation is required in the backward direction.

The Newton Backward Difference formula is a numerical method for interpolating a function at a given point x, based on equally spaced data points $(x_0, y_0), (x_1, y_1), \ldots, (x_n, y_n)$.

Formula:

$$y(x) = y_n + (x - x_n) / 1! * f_n' + (x - x_n)(x - x_{n-1}) / 2! * f_n'' + \ldots + (x - x_n)(x - x_{n-1}) \ldots (x - x_{n-m+1}) / m! * f_n^{(m)}$$

Variables:

$-\, x$: *Point at which interpolation is required*

$-\, x_0, x_1, \ldots, x_n$: *Equally spaced data points*

$-\, y(x)$: *Interpolated value of the function at x*

QUESTION:

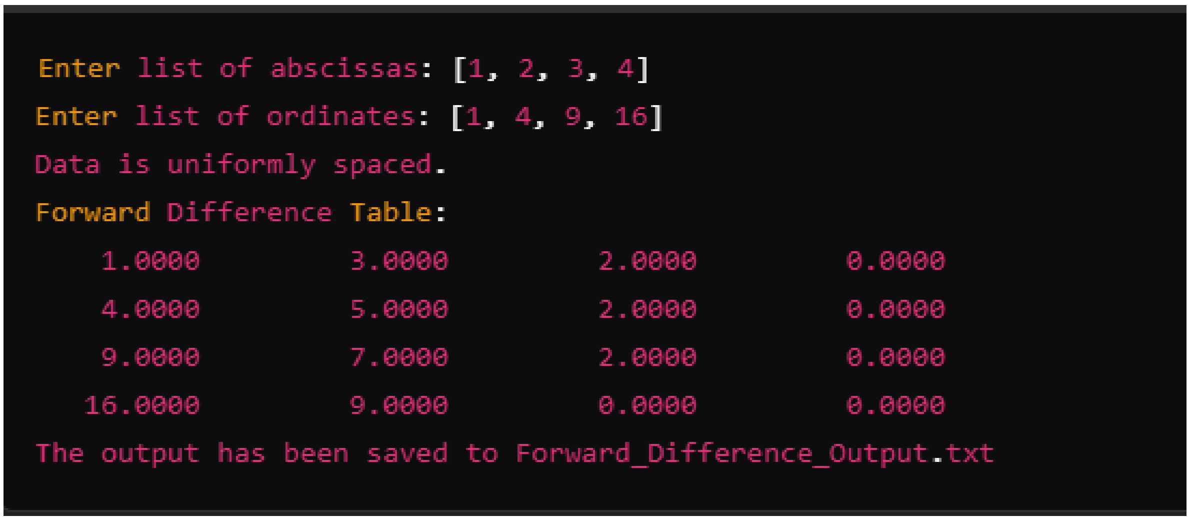
13.1. Implementation of Newton Forward Operator.

CODE:

```
% Input abscissas (x) and ordinates (y)
x = input('Enter list of abscissas: ');
y = input('Enter list of ordinates: ');

% Number of data points
n = length(x);

% Open the file to write the output
outputFile = 'Forward_Difference_Output.txt';
fileID = fopen(outputFile, 'w'); % Open for writing
if fileID == -1
    error('Unable to open the file for writing. Check file permissions or
path.');
end

% Check if data is uniformly spaced
h = x(2) - x(1);
count = 0;

for a = 2 : n - 1
    b = x(a + 1) - x(a);
    if b ~= h
        count = count + 1;
    end
end

if count == 0
```

```
    uniformityMessage = 'Data is uniformly spaced.';
else
    uniformityMessage = 'Data is not uniformly spaced.';
end

% Display and write uniformity check
disp(uniformityMessage);
fprintf(fileID, '%s\n', uniformityMessage);

% Initialize Forward Difference Table
F = zeros(n, n);
F(:, 1) = y;

% Compute Forward Differences
for j = 2 : n
    for i = 1 : n - j + 1
        F(i, j) = F(i + 1, j - 1) - F(i, j - 1);
    end
end

% Display the Forward Difference Table
disp('Forward Difference Table:');
disp(F);

% Write Forward Difference Table to file
fprintf(fileID, '\nForward Difference Table:\n');
for i = 1 : n
    for j = 1 : n
        if j <= n - i + 1 % Print only the relevant part of the table
            fprintf(fileID, '%10.4f\t', F(i, j));
        else
            fprintf(fileID, ' \t'); % Blank spaces for the upper triangular
part
        end
    end
    fprintf(fileID, '\n');
end

% Close the file
fclose(fileID);

disp(['The output has been saved to ', outputFile]);
```

OUTPUT:



```
Enter list of abscissas: [1, 2, 3, 4]
Enter list of ordinates: [1, 4, 9, 16]
Data is uniformly spaced.
Forward Difference Table:
    1.0000         3.0000         2.0000         0.0000
    4.0000         5.0000         2.0000         0.0000
    9.0000         7.0000         2.0000         0.0000
   16.0000         9.0000         0.0000         0.0000
The output has been saved to Forward_Difference_Output.txt
```

14.NUMERICAL DIFFERENTIATION

Numerical differentiation is a method used to approximate the derivative of a function at a given point. It is a fundamental concept in numerical analysis and is widely used in various fields such as physics, engineering, and economics.

Formula for Numerical Differentiation

There are several formulas for numerical differentiation, including:

Forward Difference Formula

$$f'(x) \approx (f(x+h) - f(x))\,/\,h$$

Backward Difference Formula

$$f'(x) \approx (f(x) - f(x-h))\,/\,h$$

Central Difference Formula

$$f'(x) \approx (f(x+h) - f(x-h))\,/\,(2h)$$

where:

$- f(x)\ is\ the\ function\ to\ be\ differentiated$

$- h\ is\ the\ step\ size$

$- f'(x)\ is\ the\ approximate\ derivative\ of\ f(x)\ at\ x$

QUESTION:

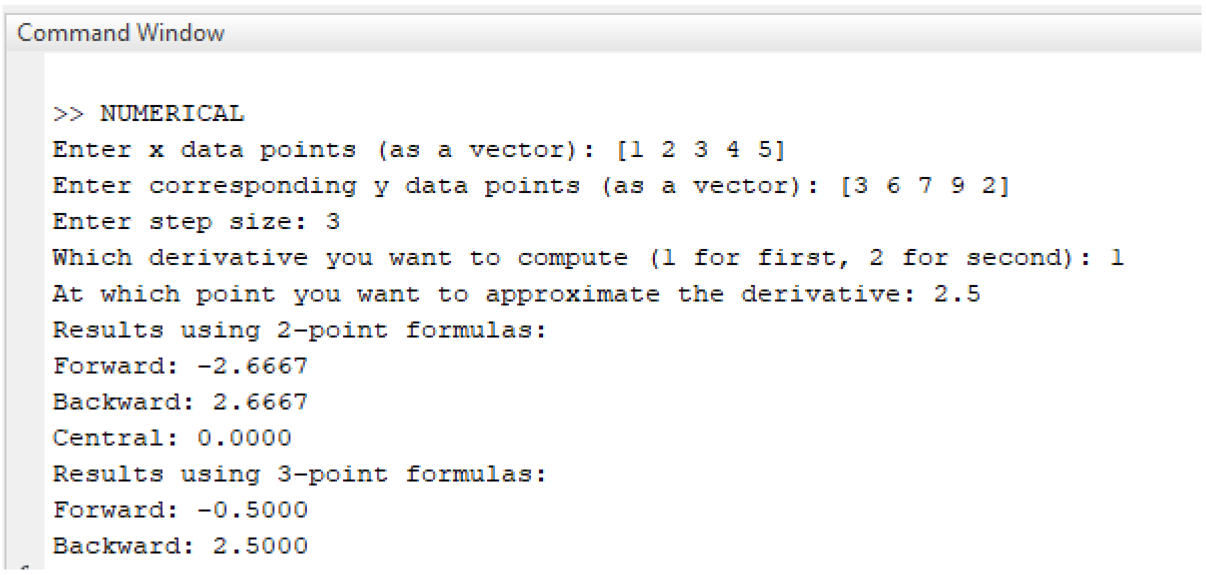
14.1. Implementation of Numerical Differentiation.

CODE:

```
% Input data points from the user
x_data = input('Enter x data points (as a vector): ');
y_data = input('Enter corresponding y data points (as a vector): ');
% Step size for the approximation
h = input('Enter step size: ');
% Which derivative to compute
d = input('Which derivative you want to compute (1 for first, 2 for
second): ');
% The point at which to approximate the derivative
x_point = input('At which point you want to approximate the derivative: ');
% Use interpolation to get the function value at a point if necessary
f = @(x) interp1(x_data, y_data, x, 'linear', 'extrap'); % Linear
interpolation
if d == 1
% First derivative using 2-point formulas
    Fd2 = (f(x_point + h) - f(x_point)) / h;     % Forward difference
    Bd2 = (f(x_point) - f(x_point - h)) / h;     % Backward difference
Cd2 = (f(x_point + h) - f(x_point - h)) / (2 * h); % Central difference
% First derivative using 3-point formulas
Fd3 = (4 * f(x_point + h) - f(x_point + 2 * h) - 3 * f(x_point)) / (2 * h);
% Forward difference (3-point)
```

```
Bd3 = (-4 * f(x_point - h) + f(x_point - 2 * h) + 3 * f(x_point)) / (2 *
h);     % Backwarddifference (3-point)
disp('Results using 2-point formulas:')
fprintf('Forward: %.4f\n', Fd2)
fprintf('Backward: %.4f\n', Bd2)
fprintf('Central: %.4f\n', Cd2)
disp('Results using 3-point formulas:')
fprintf('Forward: %.4f\n', Fd3)
fprintf('Backward: %.4f\n', Bd3)
elseif d == 2
% Second derivative using 3-point formulas
Fd2 = (f(x_point + h) - 2 * f(x_point) + f(x_point - h)) / (h^2); % Central
difference (2nd derivative)
Bd2 = (f(x_point + 2 * h) - 2 * f(x_point + h) + f(x_point)) / (h^2); %
Forward difference\(2nd derivative)
Cd2 = (f(x_point - 2 * h) - 2 * f(x_point - h) + f(x_point)) / (h^2); %
Backward difference(2nd derivative)
disp('Results using 3-point formulas for second derivative:')
fprintf('Forward: %.4f\n', Fd2)
fprintf('Backward: %.4f\n', Bd2)
fprintf('Central: %.4f\n', Cd2)
else
    disp('*Formula not available*')
end
```

OUTPUT:



```
>> NUMERICAL
Enter x data points (as a vector): [1 2 3 4 5]
Enter corresponding y data points (as a vector): [3 6 7 9 2]
Enter step size: 3
Which derivative you want to compute (1 for first, 2 for second): 1
At which point you want to approximate the derivative: 2.5
Results using 2-point formulas:
Forward: -2.6667
Backward: 2.6667
Central: 0.0000
Results using 3-point formulas:
Forward: -0.5000
Backward: 2.5000
```

WEEK#12

15.RECTANGULAR RULE

QUESTION:

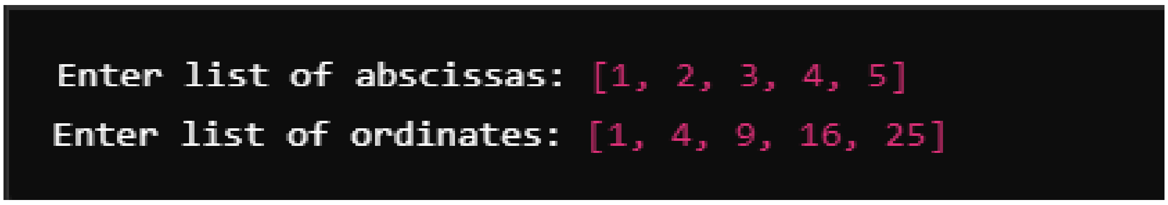
15.1. Implementation of Rectangular rule.

CODE:

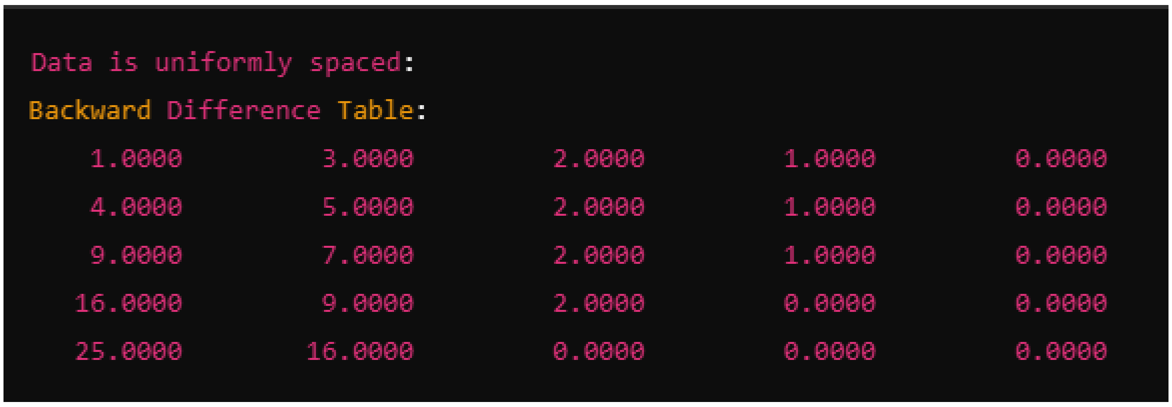
```
% Input abscissas (x) and ordinates (y)

x = input('Enter list of abscissas: ');

y = input('Enter list of ordinates: ');


% Number of data points

n = length(x);


% Check if data is uniformly spaced

h = x(2) - x(1);

count = 0;


for a = 2 : n - 1

   b = x(a + 1) - x(a);

   if b ~= h

      count = count + 1;

   end

end


if count == 0

   disp('Data is uniformly spaced:');

else

   disp('Data is not uniformly spaced:');

end
```

```
% Initialize Backward Difference Table

B = zeros(n, n);

B(:, 1) = y;


% Compute Backward Differences

for j = 2 : n

   for i = 1 : n - j + 1

       B(i, j) = B(i + 1, j - 1) - B(i, j - 1);

   end

end


% Display the Backward Difference Table

disp('Backward Difference Table:');

disp(B);
```

OUTPUT



```
Enter list of abscissas: [1, 2, 3, 4, 5]
Enter list of ordinates: [1, 4, 9, 16, 25]
```



```
Data is uniformly spaced:
Backward Difference Table:
     1.0000      3.0000      2.0000      1.0000      0.0000
     4.0000      5.0000      2.0000      1.0000      0.0000
     9.0000      7.0000      2.0000      1.0000      0.0000
    16.0000      9.0000      2.0000      0.0000      0.0000
    25.0000     16.0000      0.0000      0.0000      0.0000
```

WEEK#13

16.TRAPEZOIDAL RULE

The Trapezoidal Rule is a numerical method used to approximate the integral of a function over a given interval. It is based on the idea of approximating the region under the curve by dividing it into a series of trapezoids and summing their areas. The method provides a way to compute integrals when the exact analytical form of the integral is difficult or impossible to calculate.

The Trapezoidal Rule is a numerical method for approximating the definite integral of a continuous function f(x) over the interval [a, b]. The formula is:

$$\int [a,b]\, f(x)\, dx \approx (b - a)/2 * (f(a) + f(b))$$

Variables:

$- a \text{ and } b\text{: } Limits\ of\ integration$

$- f(a) \text{ and } f(b)\text{: } Function\ values\ at\ the\ endpoints$

$- h = b - a\text{: } Width\ of\ each\ trapezoid$

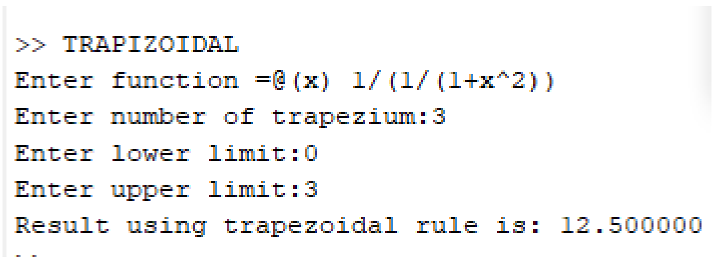
QUESTION:

16.1. Implementation of trapezoidal rule.

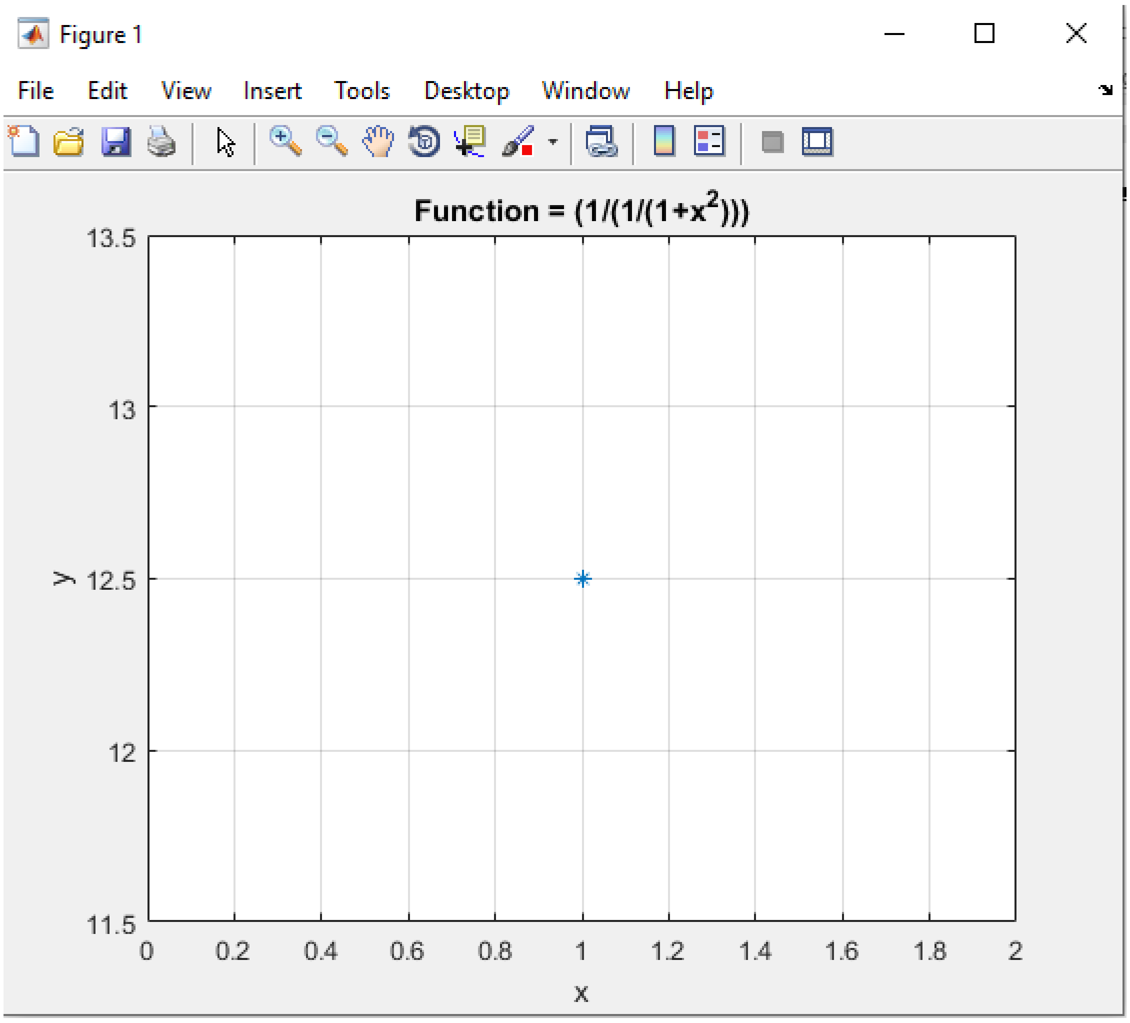
CODE:

```
f= input('Enter function =');
N= input('Enter number of trapezium:');
a= input('Enter lower limit:');
b= input('Enter upper limit:');
h= (b-a)/N;
sum=0;
for i=1:N-1
    sum = sum+f(a+i*h);
end
out = (h/2)*(f(a)+2*sum+f(b));
fprintf('Result using trapezoidal rule is: %.6f\n',out);
plot(out,'*')
grid on
title('Function = (1/(1/(1+x^2)))');
xlabel('x');
ylabel('y');
```

OUTPUT



```
>> TRAPIZOIDAL
Enter function =@(x) 1/(1/(1+x^2))
Enter number of trapezium:3
Enter lower limit:0
Enter upper limit:3
Result using trapezoidal rule is: 12.500000
```



17.SIMPSON'S RULE

The Simpson's Rule is a numerical method used to estimate the integral of a function over a given interval. It provides a more accurate approximation than the Trapezoidal Rule, particularly when the function is smooth and has varying curvature. Simpson's Rule is based on approximating the area under the curve using parabolic segments (quadratic polynomials) rather than straight-line segments (as in the Trapezoidal Rule).

Simpson's Rule is a numerical method for approximating the definite integral of a continuous function f(x) over the interval [a, b]. The formula is:

$$\int [a,b]\, f(x)\, dx \approx (b - a)/6 * [f(a) + 4f((a + b)/2) + f(b)]$$

Variables:

$- a \text{ and } b: Limits\ of\ integration$

$- f(a) \text{ and } f(b): Function\ values\ at\ the\ endpoints$

$- f((a + b)/2): Midpoint\ value$

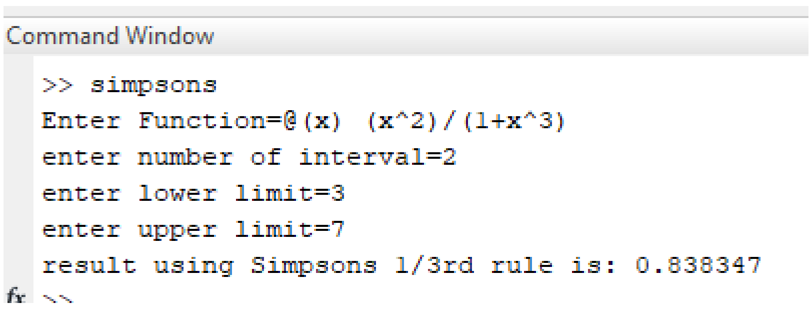
QUESTION:

17.1. Implementation of Simpson's rule.

CODE:

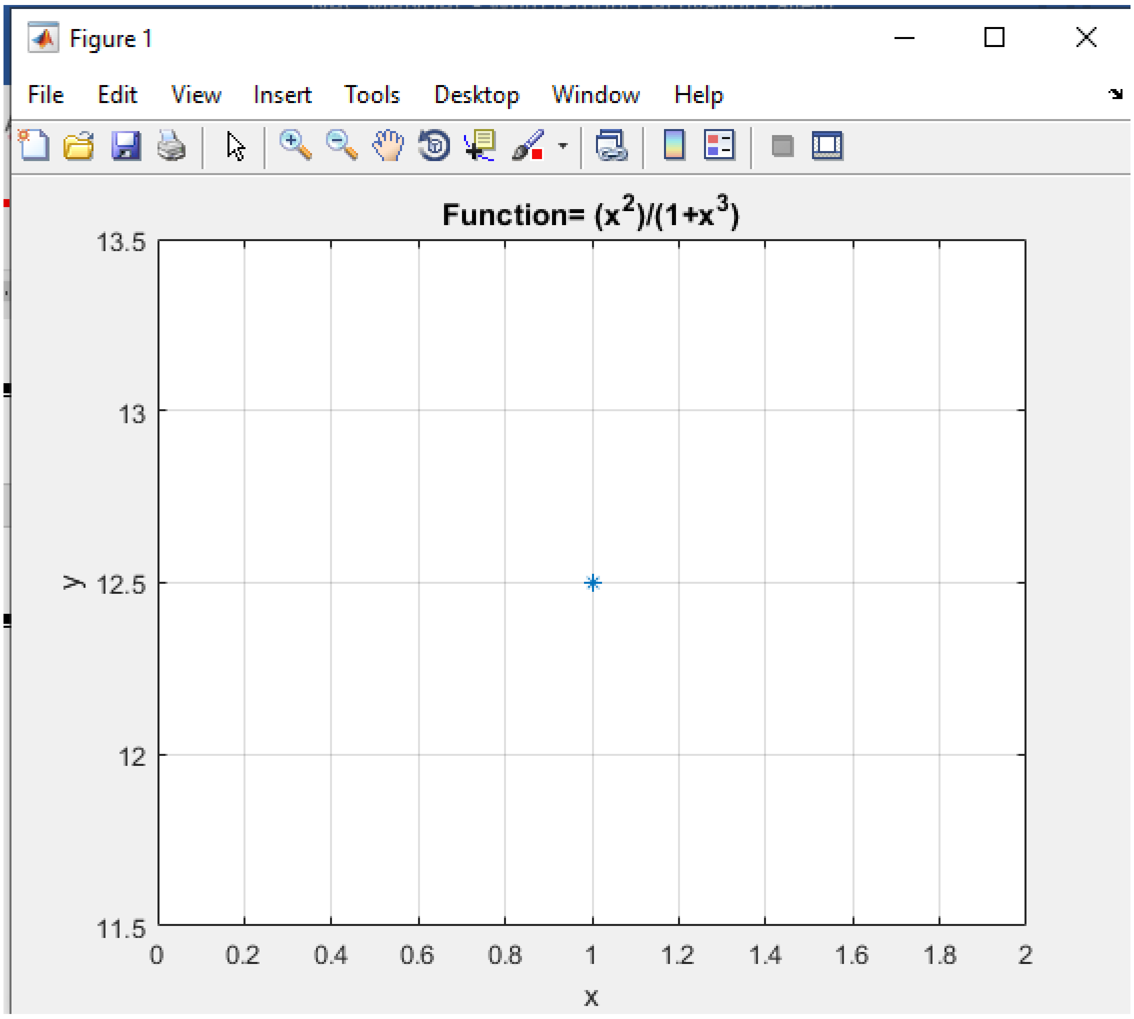
```
f=input('Enter Function=');
N=input('enter number of interval=');
a=input('enter lower limit=');
b=input('enter upper limit=');
h=(b-a)/N;
oddsum=0;
for j=1:2:N-1
    oddsum=oddsum+f(a+j*h);
end
evensum=0;
for k=2:2:N-2
    evensum=evensum+f(a+k*h);
end
Sim=(h/3)*(f(a) + 4*oddsum + 2*evensum + f(b));
fprintf('result using Simpsons 1/3rd rule is: %.6f\n',Sim);
plot(out,'*')
grid on
title('Function= (x^2)/(1+x^3)');

xlabel('x');
ylabel('y');
```

OUTPUT



Command Window

```
>> simpsons
Enter Function=@(x) (x^2)/(1+x^3)
enter number of interval=2
enter lower limit=3
enter upper limit=7
result using Simpsons 1/3rd rule is: 0.838347
fx >>
```



WEEK#14

18.POWER METHOD

The **Power Method** is an iterative algorithm used to compute the dominant (largest) eigenvalue (also known as the eigenvalue with the largest magnitude) of a square matrix. It is commonly used in applications such as solving systems of linear equations, stability analysis, and more generally in numerical methods.

The formula for the Power Method is:

$$X(k+1) = A * X(k)$$

Where:

$- X(k) \text{ is the current estimate of the eigenvector}$

$- X(k+1) \text{ is the updated estimate of the eigenvector}$

$- A \text{ is the given matrix}$

The Power Method can also be normalized to prevent overflow:

$$X(k+1) = A * X(k) \,/\, ||A * X(k)||$$

Where $||.||$ denotes the Euclidean norm.

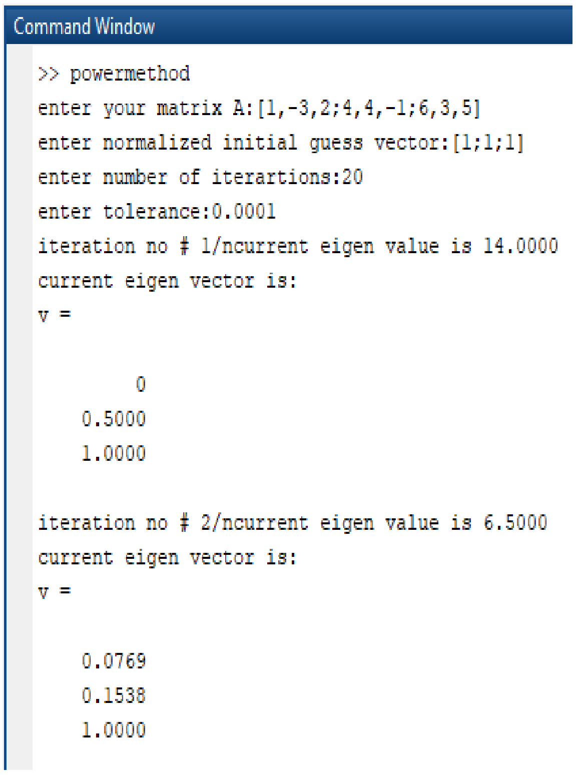
QUESTION:

18.1. Implementation of Power Method.

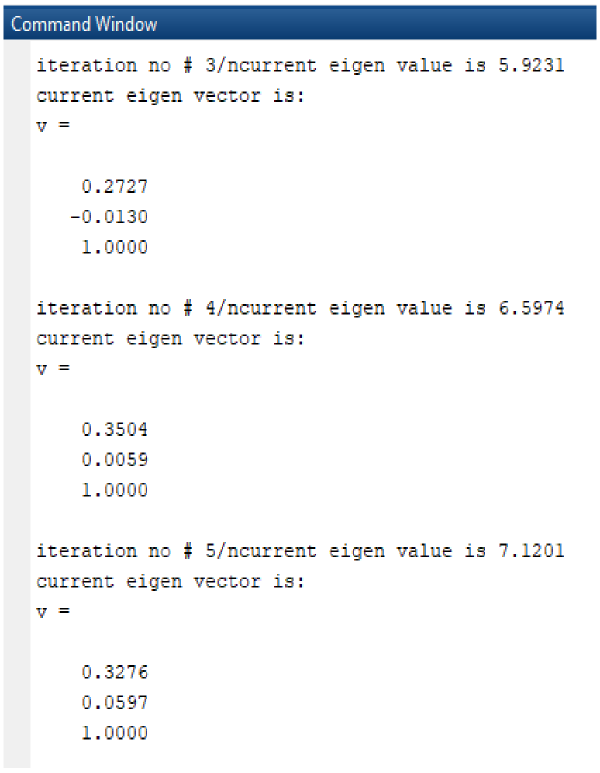
CODE:

```
A=input('enter your matrix A:');
v=input('enter normalized initial guess vector:');
n=input('enter number of iterartions:');
e=input('enter tolerance:');
v0=v;
for i=1:n
    v=A*v0;
    M=max(v);
    v=v/M;
    if abs(v-v0)<e
        break
    end
    v0=v;
    fprintf('iteration no # %d/n',i)
    fprintf('current eigen value is %.4f\n',M)
    fprintf('current eigen vector is:')
    v
end
```

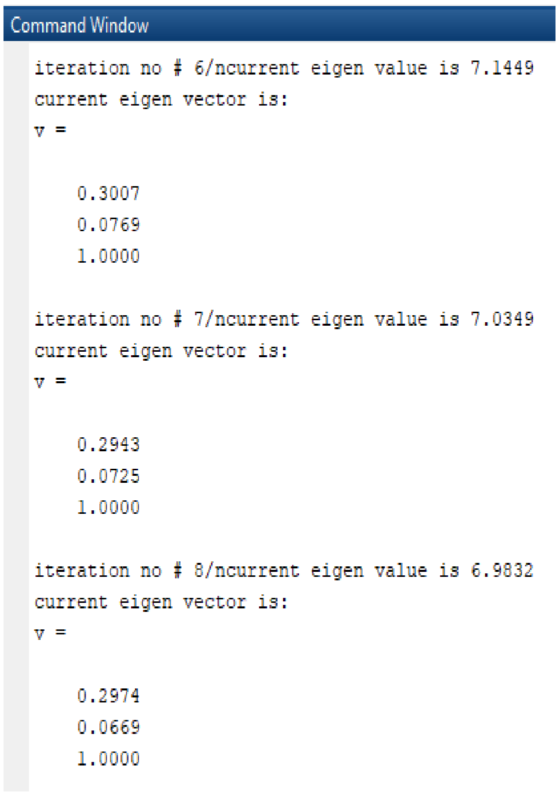
OUTPUT



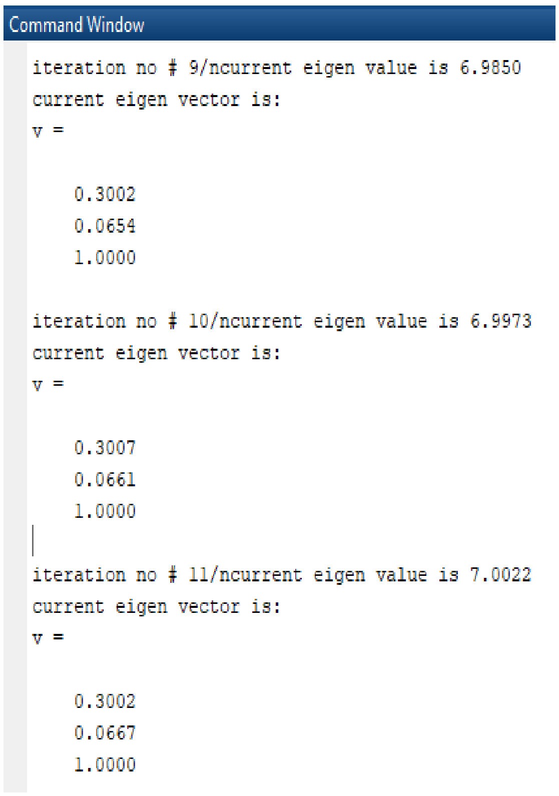
```
>> powermethod
enter your matrix A:[1,-3,2;4,4,-1;6,3,5]
enter normalized initial guess vector:[1;1;1]
enter number of iterartions:20
enter tolerance:0.0001
iteration no # 1/ncurrent eigen value is 14.0000
current eigen vector is:
v =


         0
    0.5000
    1.0000

iteration no # 2/ncurrent eigen value is 6.5000
current eigen vector is:
v =

    0.0769
    0.1538
    1.0000
```



```
iteration no # 3/ncurrent eigen value is 5.9231
current eigen vector is:
v =

    0.2727
   -0.0130
    1.0000

iteration no # 4/ncurrent eigen value is 6.5974
current eigen vector is:
v =

    0.3504
    0.0059
    1.0000

iteration no # 5/ncurrent eigen value is 7.1201
current eigen vector is:
v =

    0.3276
    0.0597
    1.0000
```



```
iteration no # 6/ncurrent eigen value is 7.1449
current eigen vector is:
v =

    0.3007
    0.0769
    1.0000

iteration no # 7/ncurrent eigen value is 7.0349
current eigen vector is:
v =

    0.2943
    0.0725
    1.0000

iteration no # 8/ncurrent eigen value is 6.9832
current eigen vector is:
v =

    0.2974
    0.0669
    1.0000
```



```
iteration no # 9/ncurrent eigen value is 6.9850
current eigen vector is:
v =

    0.3002
    0.0654
    1.0000

iteration no # 10/ncurrent eigen value is 6.9973
current eigen vector is:
v =

    0.3007
    0.0661
    1.0000

iteration no # 11/ncurrent eigen value is 7.0022
current eigen vector is:
v =

    0.3002
    0.0667
    1.0000
```

19.EULER'S METHOD

Euler's Method is a simple numerical technique used to solve ordinary differential equations (ODEs) that cannot be solved analytically. It provides an approximate solution by iteratively stepping through the ODE using a small step size.

$$dy/dx = f(x,y)$$

with the initial condition:

$$y(x_0) = y_0$$

The formula for Euler's method is:

$$y_{n+1} = y_n + h * f(x_n, y_n)$$

Variables:

$-\, x_n\text{: } Current\ value\ of\ x$

$-\, y_n\text{: } Current\ value\ of\ y$

$-\, h\text{: } Step\ size$

$-\, f(x,y)\text{: } Function\ defining\ the\ derivative$

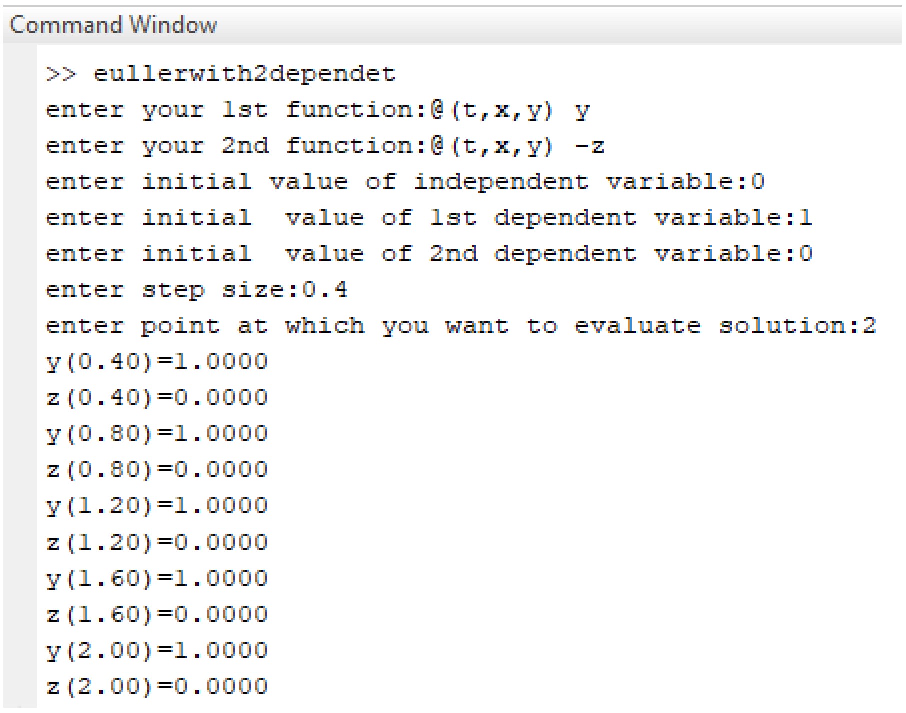
QUESTION:

19.1. Implementation of Euler's Method with 2 dependent variable.

CODE:

```
f=input('enter your 1st function:'); %right hand side of ODE
g=input('enter your 2nd function:'); %right hand side of ODE
t0=input('enter initial value of independent variable:');
y0=input('enter initial  value of 1st dependent variable:');
z0=input('enter initial  value of 2nd dependent variable:');
h=input('enter step size:');
tn=input('enter point at which you want to evaluate solution:');
n=(tn-t0)/h;
t(1)=t0; y(1)=y0; z(1)=z0;
for i=1:n
    y(i+1)=y(i) + h*f(t(i),y(i),z(i));
    z(i+1)=z(i) + h*g(t(i),y(i),z(i));
    t(i+1)=t0 + i*h;
    fprintf('y(%.2f)=%.4f\n',t(i+1),y(i+1))
    fprintf('z(%.2f)=%.4f\n',t(i+1),z(i+1))
end
```

OUTPUT



```
>> eullerwith2dependet
enter your 1st function:@(t,x,y) y
enter your 2nd function:@(t,x,y) -z
enter initial value of independent variable:0
enter initial  value of 1st dependent variable:1
enter initial  value of 2nd dependent variable:0
enter step size:0.4
enter point at which you want to evaluate solution:2
y(0.40)=1.0000
z(0.40)=0.0000
y(0.80)=1.0000
z(0.80)=0.0000
y(1.20)=1.0000
z(1.20)=0.0000
y(1.60)=1.0000
z(1.60)=0.0000
y(2.00)=1.0000
z(2.00)=0.0000
```

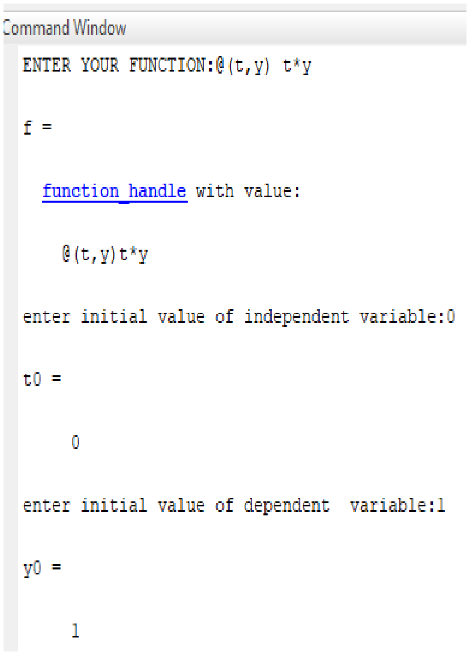
QUESTION:

19.2. Implementation of Euler's Method with 1 dependent variable.

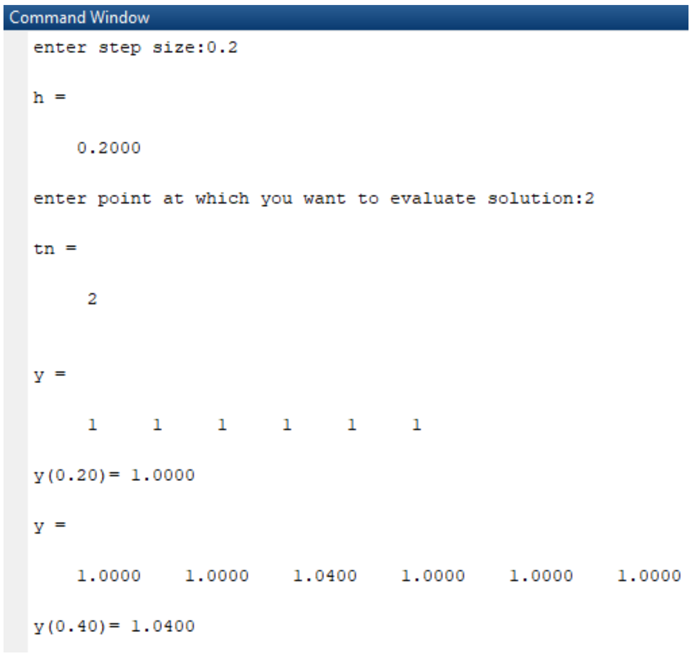
CODE:

```
f = input('ENTER YOUR FUNCTION:')
t0 = input('enter initial value of independent variable:')
y0 = input('enter initial value of dependent  variable:')
h = input('enter step size:')
tn = input('enter point at which you want to evaluate solution:')
n = (tn-t0)/h;
t(1) = t0; y(1) = y0;
for i=1:n
    y(i+1)=y(i) + h*f(t(i),y(i))
    t(i+1) = t0 + i*h;
    fprintf('y(%.2f)= %.4f\n',t(i+1),y(i+1))
end
```

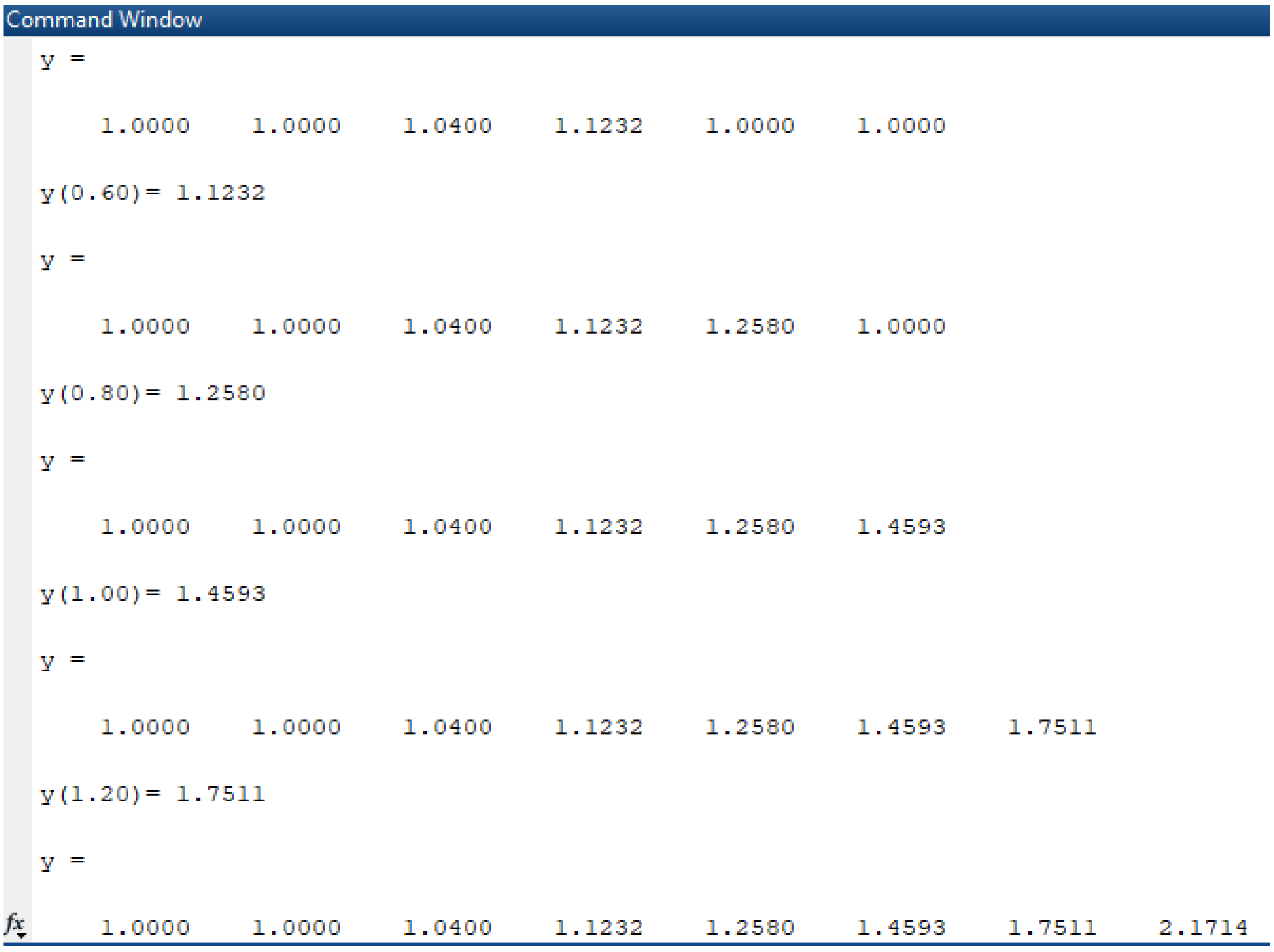
OUTPUT



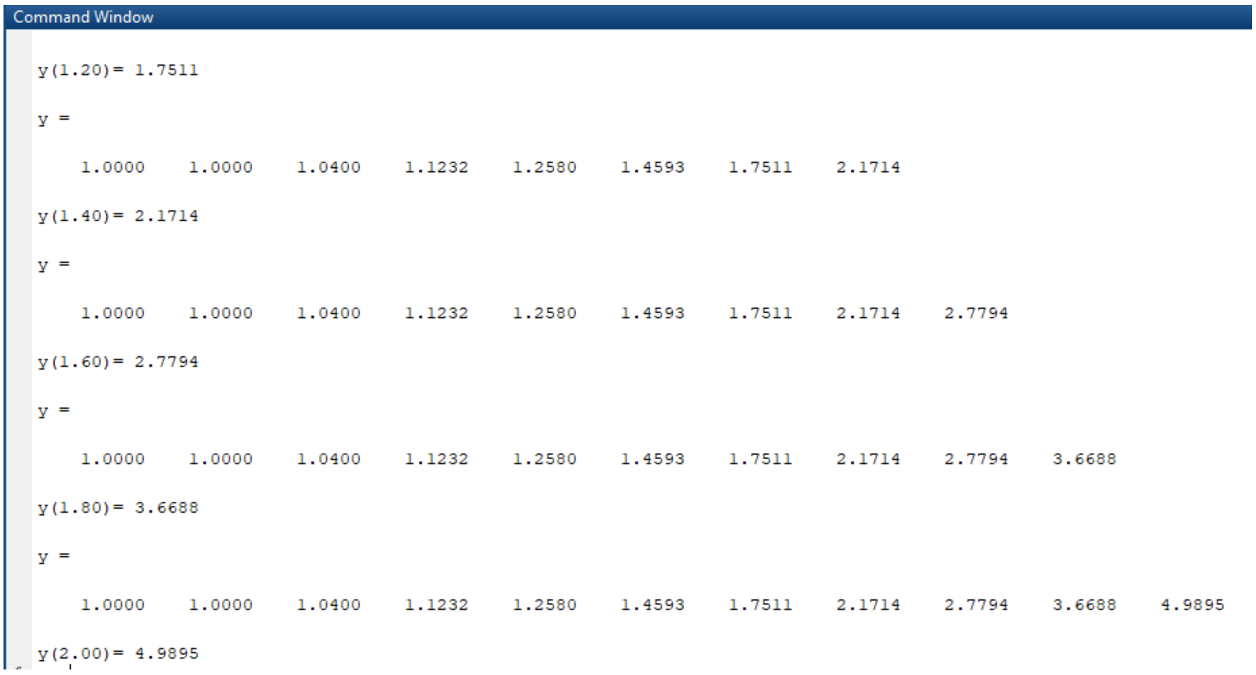
```
ENTER YOUR FUNCTION:@(t,y) t*y

f =

  function_handle with value:

    @(t,y)t*y

enter initial value of independent variable:0

t0 =

     0

enter initial value of dependent  variable:1

y0 =

     1
```



```
enter step size:0.2

h =

    0.2000

enter point at which you want to evaluate solution:2

tn =

     2


y =

     1     1     1     1     1     1

y(0.20)= 1.0000

y =

    1.0000    1.0000    1.0400    1.0000    1.0000    1.0000

y(0.40)= 1.0400
```



```
y =

    1.0000    1.0000    1.0400    1.1232    1.0000    1.0000

y(0.60)= 1.1232

y =

    1.0000    1.0000    1.0400    1.1232    1.2580    1.0000

y(0.80)= 1.2580

y =

    1.0000    1.0000    1.0400    1.1232    1.2580    1.4593

y(1.00)= 1.4593

y =

    1.0000    1.0000    1.0400    1.1232    1.2580    1.4593    1.7511

y(1.20)= 1.7511

y =

    1.0000    1.0000    1.0400    1.1232    1.2580    1.4593    1.7511    2.1714
```



```
y(1.20)= 1.7511

y =

    1.0000    1.0000    1.0400    1.1232    1.2580    1.4593    1.7511    2.1714

y(1.40)= 2.1714

y =

    1.0000    1.0000    1.0400    1.1232    1.2580    1.4593    1.7511    2.1714    2.7794

y(1.60)= 2.7794

y =

    1.0000    1.0000    1.0400    1.1232    1.2580    1.4593    1.7511    2.1714    2.7794    3.6688

y(1.80)= 3.6688

y =

    1.0000    1.0000    1.0400    1.1232    1.2580    1.4593    1.7511    2.1714    2.7794    3.6688    4.9895

y(2.00)= 4.9895
```

20.RANGA KUTTA METHOD(RK-METHOD)

The Runge-Kutta Method (RK Method) is a family of iterative techniques used to solve ordinary differential equations (ODEs) numerically. It provides an efficient and accurate method for solving ODEs by estimating the value of the dependent variable at successive points. The method derives its name from Carl Runge and Wilhelm Kutta, who independently developed this approach in the early 20th century.

$$dy/dx = f(x, y)$$

The method consists of four stages, followed by a final update to obtain the next value of y.

Stage 1 (k1):

$$k_1 = h * f(x_n, y_n)$$

Stage 2 (k2):

$$k_2 = h * f(x_n + h/2, y_n + k_1/2)$$

Stage 3 (k3):

$$k_3 = h * f(x_n + h/2, y_n + k_2/2)$$

Stage 4 (k4):

$$k_4 = h * f(x_n + h, y_n + k_3)$$

Final Update (y_{n+1}):

$$y_{n+1} = y_n + (1/6) * (k_1 + 2k_2 + 2k_3 + k_4)$$

Variables:

$- y_n$: *Current value of the dependent variable*

$- f(x_n, y_n)$: *ODE to be solved*

$- h$: *Step size*

$- k_1, k_2, k_3, k_4$: *Intermediate values calculated at each stage*

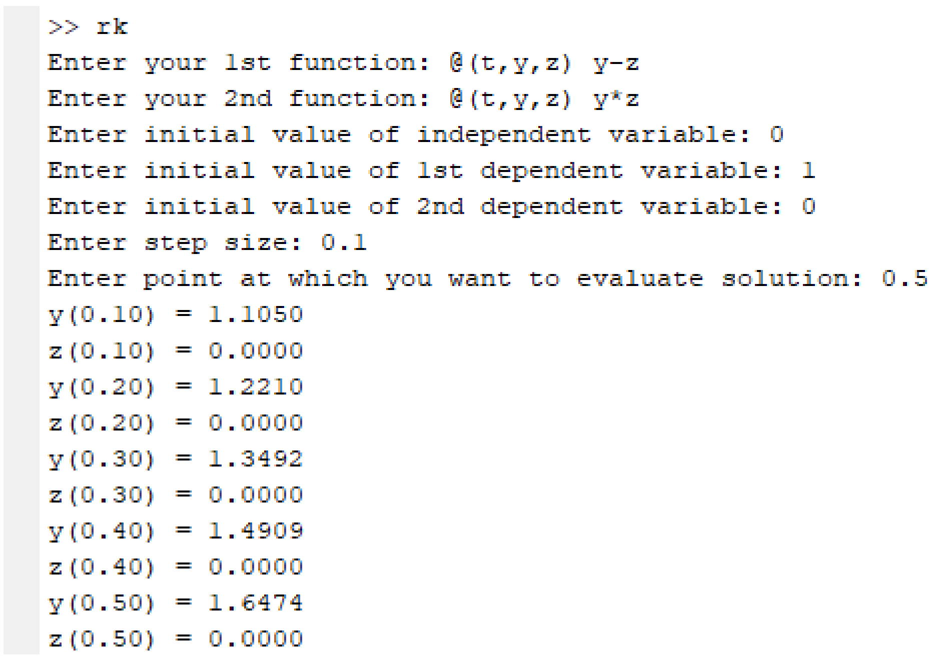
QUESTION:

20.1. Implementation of RK-Method.

CODE:

```
f = input('Enter your 1st function: '); % right hand side of ODE
g = input('Enter your 2nd function: '); % right hand side of ODE
t0 = input('Enter initial value of independent variable: ');
y0 = input('Enter initial value of 1st dependent variable: ');
z0 = input('Enter initial value of 2nd dependent variable: ');
h = input('Enter step size: ');
tn = input('Enter point at which you want to evaluate solution: ');
n = (tn - t0) / h;
t(1) = t0; y(1) = y0; z(1) = z0;

for i = 1:n
    t(i + 1) = t0 + i * h;
    k1 = h * f(t(i), y(i), z(i));
    m1 = h * g(t(i), y(i), z(i));
    k2 = h * f(t(i + 1), y(i) + k1, z(i) + m1); % Corrected k2 calculation
    m2 = h * g(t(i + 1), y(i) + k1, z(i) + m1); % Corrected m2 calculation
    y(i + 1) = y(i) + (1 / 2) * (k1 + k2);
    z(i + 1) = z(i) + (1 / 2) * (m1 + m2);
    fprintf('y(%.2f) = %.4f\n', t(i + 1), y(i + 1))
    fprintf('z(%.2f) = %.4f\n', t(i + 1), z(i + 1))
end
```

OUTPUT



```
>> rk
Enter your 1st function: @(t,y,z) y-z
Enter your 2nd function: @(t,y,z) y*z
Enter initial value of independent variable: 0
Enter initial value of 1st dependent variable: 1
Enter initial value of 2nd dependent variable: 0
Enter step size: 0.1
Enter point at which you want to evaluate solution: 0.5
y(0.10) = 1.1050
z(0.10) = 0.0000
y(0.20) = 1.2210
z(0.20) = 0.0000
y(0.30) = 1.3492
z(0.30) = 0.0000
y(0.40) = 1.4909
z(0.40) = 0.0000
y(0.50) = 1.6474
z(0.50) = 0.0000
```
