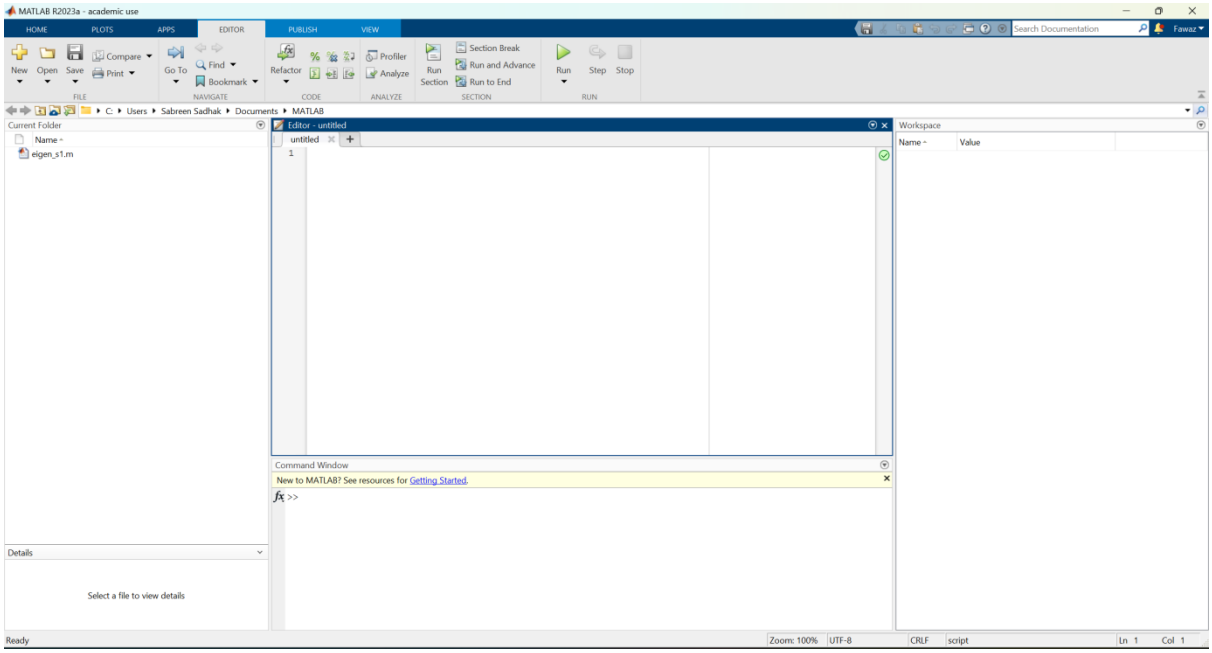
INSTITUTE OF REMOTE SENSING
ANNA UNIVERSITY, CHENNAI - 600 025.

LAB MANUAL
MATLAB LABORATORY

TABLE OF CONTENTS

Expt No. 1	GETTING STARTED WITH MATLAB	Date of Expt:

MATLAB (Matrix Laboratory) is a high-level programming language and interactive environment designed for numerical computation, data visualization, and algorithm development. When you open MATLAB, you'll see a user interface that's designed to make working with data, code, and results easy. Let's go through the main tabs and features of the MATLAB interface.



1. Home Tab

The Home tab is the default tab that you see when you launch MATLAB. It includes the following key features:

- **New Script**: Creates a new script (file with .m extension) where you can write MATLAB code.
- **Open**: Opens existing MATLAB scripts, functions, or variables.
- **Save/Save As**: Allows you to save your scripts and variables.
- **Import Data**: Imports data from external files like spreadsheets, text files, etc.
- **Preferences**: Accesses settings related to the MATLAB environment (like display, language, etc.).
- **Add-Ons**: Allows you to access and download additional toolboxes and apps.

2. Plot Tab

The Plot tab becomes active when you are working with data in MATLAB. It provides:

- **2D and 3D Plots**: Options for generating various types of plots, such as line plots, bar charts, histograms, surface plots, etc.
- **Visualization Customization**: Tools to add titles, labels, legends, grid lines, and format axes in plots.
- **Different Styles**: Quick buttons to change plot styles (e.g., different marker types, colors).

3. Editor Tab

When you open a new script or function, the Editor tab becomes visible. It's where you write and edit your code. Key features include:

- **Run/Debug**: Allows you to run the script or step through it line by line to identify errors.
- **Breakpoints**: Used to pause execution at specific lines for debugging purposes.
- **Comment/Uncomment**: Options to comment out parts of code for readability or testing.
- **Code Folding**: Collapses code sections to simplify navigation in large scripts.

4. Apps Tab

MATLAB includes many built-in apps for specific tasks. In this tab, you can:

- **Access Apps**: Use apps for tasks like machine learning, signal processing, control systems, etc. You can search for and launch apps for various domains.
- **Install New Apps**: Download apps from the MATLAB Add-On Explorer.

5. View Tab

The View tab allows you to customize the layout of the MATLAB interface. Features include:

- **Command Window**: Displays the main window where you can enter commands and see output.
- **Workspace**: Shows variables that are currently in memory, along with their size and class.
- **Current Folder**: Displays the files and directories available for use in the current session.
- **Figure Windows**: Shows open figures and visualizations.

- **Variable Editor**: Allows you to view and edit variables in spreadsheet form.

6. Editor and Live Editor

MATLAB has two modes for coding:

- **Editor**: This is the standard environment where you write your code and run scripts.
- **Live Editor**: Allows you to create interactive notebooks that combine code, output, and formatted text. You can include comments, equations, and plots directly in the notebook.

7. Command Window

The Command Window is where you interact with MATLAB by typing commands directly. It's ideal for quick calculations, testing code, and seeing immediate results.

8. Workspace

This panel lists all the variables created in your session. For each variable, you'll see its name, size, and data type. You can double-click any variable to open it in the Variable Editor.

9. Current Folder

The Current Folder panel shows the files in the working directory. You can navigate through your directories, run scripts, and manage files here.

10. Tool strip Customization

The tool strip is the bar that includes all these tabs and buttons. You can customize it by adding or removing buttons, rearranging tabs, or even creating your own custom toolbars.

Expt No. 2	ARRAYS AND MATRICES	Date of Expt:

Ex 2.1

Procedure:

Step 1: Array Initialization

- **A**: Creates a row vector A containing elements [1 4 5 4].
- **A1**: Creates another row vector A1 containing elements [2, 4, 6, 8].
- **B**: Creates a column vector B by transposing a row vector [10 25 74 35 42] using the ' operator.
- **C**: Adds 10 to each element of the vector [2 0 4 0], resulting in C = [12 10 14 10].

Step 2: Matrix Operations

- **D = C'A*: Transposes C to a column vector and performs matrix multiplication with A. Result is a scalar since it's a dot product between a column vector and a row vector.
- **E = A'C*: Transposes A to a column vector and performs matrix multiplication with C. This is also a dot product, resulting in a scalar.

Step 3: Defining More Arrays

- **F**: Creates a column vector F containing elements [3; 6; 9; 12].
- **G**: Creates a row vector G with elements [1 2 3].
- **H**: Creates a column vector H with elements [2; 4].
- **I = 1:8**: Creates a row vector I containing integers from 1 to 8.
- **J = 1:2:8**: Creates a row vector J starting at 1, incrementing by 2 up to 8, resulting in [1 3 5 7].

Step 4: Scalar Assignment

- **X = 0.5**: Assigns the value 0.5 to the variable X.

Step 5: Matrix Initialization and Concatenation

- **MAT1**: Defines a matrix MAT1 with the following values:

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

- **MAT2 = [C; MAT1]**: Concatenates the row vector C on top of the matrix MAT1, creating a new matrix MAT2.

Step 6: Matrix Operations

- **MAT3**: Defines a matrix MAT3 with the values:

2 1 3
2 5 1
6 3 4

- **MAT4 = G*MAT3**: Multiplies row vector G (1x3) with matrix MAT3 (3x3), resulting in a 1x3 row vector MAT4.

Step 7: Matrix Multiplication

- **MAT5**: Defines a matrix MAT5 with the following values:

1 2
3 4
5 6

- **MAT6 = [7 8; 9 1; 2 3]H*: Multiplies matrix [7 8; 9 1; 2 3] (3x2) with column vector H (2x1), resulting in a 3x1 column vector MAT6.

Step 8: Element-Wise Operations

- **MAT7 = [3 4 5 6; 7 8 9 1].^X**: Performs an element-wise power operation on the matrix [3 4 5 6; 7 8 9 1], raising each element to the power X (which is 0.5, i.e., square root).
- **MAT8 = MAT7(1:3)**: Extracts the first 3 elements of the first row from MAT7 and stores them in MAT8.
- **MAT9 = MAT7(1:2)**: Extracts the first 2 elements of the first row from MAT7 and stores them in MAT9.

Step 9: More Matrix Operations

- **MAT10**: Defines a matrix MAT10 with the following values:

1 2 3
3 2 1
5 4 3

- **MAT11 = MAT10.MAT10*: Performs element-wise multiplication of MAT10 with itself.
- **MAT12 = MAT10.MAT3*: Performs element-wise multiplication of MAT10 with MAT3.

Step 10: Linspace Function

- **MAT13 = linspace(1,2)**: Creates a row vector with 100 points linearly spaced between 1 and 2.
- **MAT14 = linspace(1,2,5)**: Creates a row vector with 5 points linearly spaced between 1 and 2.

Code:

```
A = [1 4 5 4]
A1 = [2,4,6,8]
B = [10 25 74 35 42]'
C = [2 0 4 0] + 10
D = C'*A
E = A'*C
F = [3;6;9;12]
G = [1 2 3]
H = [2;4]
I = 1:8
J = 1:2:8.
X = 0.5
MAT1 = [1 2 3 4;3 6 9 12;4 8 12 16;5 10 15 20]
MAT2 = [C;MAT1]          % Concatenates the row vector C with matrix MAT1
.
MAT3 = [2 1 3;2 5 1;6 3 4]
MAT4 = G*MAT3
MAT5 = [1 2;3 4;5 6]     .
MAT6 = [7 8;9 1;2 3]*H
MAT7 = [3 4 5 6; 7 8 9 1].^X % Element-wise power operation. Each element
of the matrix is raised to the power X (0.5 here).
MAT8 = MAT7(1:3)         % Extracts the first 3 elements from the first row of
matrix MAT7 into a row vector MAT8.
MAT9 = MAT7(1:2)
MAT10 = [1 2 3; 3 2 1; 5 4 3]
MAT11 = MAT10.*MAT10      % Element-wise multiplication of MAT10 with
itself.
MAT12 = MAT10.*MAT3       % Element-wise multiplication of MAT10 with
MAT3.
```

```
MAT13 = linspace(1,2)     % Row vector of 100 points from 1 to 2
MAT14 = linspace(1,2,5)   % 5 points from 1 to 2.
```

O/P (Command Window)

```
A =     1     4     5     4

A1 =     2     4     6     8

B =
   10
   25
   74
   35
   42

C =    12    10    14    10

D =
   12    48    60    48
   10    40    50    40
   14    56    70    56
   10    40    50    40

E =
   12    10    14    10
   48    40    56    40
   60    50    70    50
   48    40    56    40

F =
    3
    6
    9
   12

G =     1     2     3
```

H =

2

4

I = 1 2 3 4 5 6 7 8

J = 1 3 5 7

X = 0.5000

MAT1 =

1 2 3 4

3 6 9 12

4 8 12 16

5 10 15 20

MAT2 =

12 10 14 10

1 2 3 4

3 6 9 12

4 8 12 16

5 10 15 20

MAT3 =

2 1 3

2 5 1

6 3 4

MAT4 = 24 20 17

MAT5 =

1 2

3 4

5 6

MAT6 =

```
   46
   22
   16

MAT7 =
   1.7321    2.0000    2.2361    2.4495
   2.6458    2.8284    3.0000    1.0000

MAT8 =
   1.7321    2.6458    2.0000

MAT9 =
   1.7321    2.6458    2.0000    2.8284    2.2361

MAT10 =
    1     2     3
    3     2     1
    5     4     3

MAT11 =
    1     4     9
    9     4     1
   25    16     9


MAT12 =

    2     2     9
    6    10     1
   30    12    12

MAT13 =

  Columns 1 through 13

   1.0000    1.0101    1.0202    1.0303    1.0404    1.0505    1.0606    1.0707
1.0808    1.0909    1.1010    1.1111    1.1212
```

Columns 14 through 26

1.1313 1.1414 1.1515 1.1616 1.1717 1.1818 1.1919 1.2020 1.2121 1.2222 1.2323 1.2424 1.2525

Columns 27 through 39

1.2626 1.2727 1.2828 1.2929 1.3030 1.3131 1.3232 1.3333 1.3434 1.3535 1.3636 1.3737 1.3838

Columns 40 through 52

1.3939 1.4040 1.4141 1.4242 1.4343 1.4444 1.4545 1.4646 1.4747 1.4848 1.4949 1.5051 1.5152

Columns 53 through 65

1.5253 1.5354 1.5455 1.5556 1.5657 1.5758 1.5859 1.5960 1.6061 1.6162 1.6263 1.6364 1.6465

Columns 66 through 78

1.6566 1.6667 1.6768 1.6869 1.6970 1.7071 1.7172 1.7273 1.7374 1.7475 1.7576 1.7677 1.7778

Columns 79 through 91

1.7879 1.7980 1.8081 1.8182 1.8283 1.8384 1.8485 1.8586 1.8687 1.8788 1.8889 1.8990 1.9091

Columns 92 through 100

1.9192 1.9293 1.9394 1.9495 1.9596 1.9697 1.9798 1.9899 2.0000

MAT14 =

1.0000 1.2500 1.5000 1.7500 2.0000

Ex 2.2
Procedure:

Step 1: Vector Initialization

- **a**: Defines a sample vector a with elements [10, 5, 7, 2, 8, 3, 4, 9, 6, 1].

Step 2: Sum of Elements

- **sumOfA = sum(a)**: Calculates the sum of the elements in vector a.
- **fprintf('Sum of a: %d\n', sumOfA)**: Prints the sum of the vector a.

Step 3: Mean of Elements

- **meanOfA = mean(a)**: Computes the mean (average) value of the elements in vector a.
- **fprintf('Mean of a: %.2f\n', meanOfA)**: Prints the mean, formatted to two decimal places.

Step 4: Median of Elements

- **medianOfA = median(a)**: Finds the median value of the elements in vector a.
- **fprintf('Median of a: %.2f\n', medianOfA)**: Prints the median value, formatted to two decimal places.

Step 5: Standard Deviation

- **stdDevA = std(a)**: Computes the standard deviation of the elements in vector a.
- **fprintf('Standard Deviation of a: %.2f\n', stdDevA)**: Prints the standard deviation, formatted to two decimal places.

Step 6: Minimum Value and Index

- **[minVal, minIdx] = min(a)**: Finds the minimum value of the vector a and its index.
- **fprintf('Minimum value of a: %d (at index %d)\n', minVal, minIdx)**: Prints the minimum value and its corresponding index.

Step 7: Maximum Value and Index

- **[maxVal, maxIdx] = max(a)**: Finds the maximum value of the vector a and its index.
- **fprintf('Maximum value of a: %d (at index %d)\n', maxVal, maxIdx)**: Prints the maximum value and its corresponding index.

Step 8: Sorting in Ascending Order

- **sortedAscend = sort(a, 'ascend')**: Sorts the vector a in ascending order.
- **fprintf('Ascending order of a: ')**: Prints the sorted vector in ascending order.

Step 9: Sorting in Descending Order

- **sortedDescend = sort(a, 'descend')**: Sorts the vector a in descending order.
- **fprintf('Descending order of a: ')**: Prints the sorted vector in descending order.

Step 10: Variance of the Elements

- **varianceOfA = var(a)**: Calculates the variance of the elements in vector a.
- **fprintf('Variance of a: %.2f\n', varianceOfA)**: Prints the variance, formatted to two decimal places.

Step 11: Cumulative Sum of Elements

- **cumSumOfA = cumsum(a)**: Computes the cumulative sum of the elements in vector a.
- **fprintf('Cumulative sum of a: ')**: Prints the cumulative sum of vector a.

Step 12: Vector Operations - Dot Product

- **b = [2, 3, 4]**: Defines vector b.
- **c = [5, 6, 7]**: Defines vector c.
- **dotProduct = dot(b, c)**: Computes the dot product of vectors b and c.
- **fprintf('Dot product of b and c: %d\n', dotProduct)**: Prints the dot product of b and c.

Step 13: Vector Operations - Cross Product

- **crossProduct = cross(b, c)**: Computes the cross product of vectors b and c.
- **fprintf('Cross product of b and c: [%d %d %d]\n', crossProduct(1), crossProduct(2), crossProduct(3))**: Prints the cross product of b and c.

Step 14: Division Operations

- **d = [2; 3]**: Defines a column vector d.
- **e = [4; 6]**: Defines a column vector e.
- **f = [2 3]**: Defines a row vector f.
- **g = [4 6]**: Defines a row vector g.

Now, different division operations are performed:

  - **d/e**: Performs element-wise division between column vectors d and e.

- **d\e**: Solves d for e in a least-squares sense.
- **f/g**: Performs element-wise division between row vectors f and g.
- **f\g**: Solves f for g in a least-squares sense.

The results of these operations are printed using fprintf.

Step 15: Generating Random Numbers and Matrices

- **rand(1,5)**: Generates a 1x5 matrix of random numbers between 0 and 1.
- **randperm(5)**: Generates a random permutation of integers from 1 to 5.
- **ones(1,5)**: Creates a 1x5 matrix of ones.
- **zeros(1,5)**: Creates a 1x5 matrix of zeros.

Code:

```
a = [10, 5, 7, 2, 8, 3, 4, 9, 6, 1]; % Define a sample vector

sumOfA = sum(a); % Sum
fprintf('Sum of a: %d\n', sumOfA);

meanOfA = mean(a); % Mean
fprintf('Mean of a: %.2f\n', meanOfA);

medianOfA = median(a); % Median
fprintf('Median of a: %.2f\n', medianOfA);

stdDevA = std(a); % Standard deviation
fprintf('Standard Deviation of a: %.2f\n', stdDevA);

[minVal, minIdx] = min(a); % Min
fprintf('Minimum value of a: %d (at index %d)\n', minVal, minIdx);

[maxVal, maxIdx] = max(a); % Max
fprintf('Maximum value of a: %d (at index %d)\n', maxVal, maxIdx);

sortedAscend = sort(a, 'ascend'); % Ascending
fprintf('Ascending order of a: ');
fprintf('%d ', sortedAscend);
fprintf('\n');

sortedDescend = sort(a, 'descend'); % Descending
fprintf('Descending order of a: ');
fprintf('%d ', sortedDescend);
fprintf('\n');
```

```
varianceOfA = var(a); % Variance
fprintf('Variance of a: %.2f\n', varianceOfA);

cumSumOfA = cumsum(a); % Cumulative sum
fprintf('Cumulative sum of a: ');
fprintf('%d ', cumSumOfA);

b = [2, 3, 4];
c = [5, 6, 7];

dotProduct = dot(b, c);
fprintf('Dot product of b and c: %d\n', dotProduct);

crossProduct = cross(b, c);
fprintf('Cross product of b and c: [%d %d %d]\n', crossProduct(1),
crossProduct(2), crossProduct(3));

d = [2; 3];
e = [4; 6];
f = [2 3];
g = [4 6];

fprintf('Result of d/e: %f\n', d/e);
fprintf('Result of d\\e: %f\n', d\e);
fprintf('Result of f/g: %f\n', f/g);
fprintf('Result of f\\g: %f\n', f\g);

rand(1,5)   % 1x5 random numbers
randperm(5) % Random order of 1-5
ones(1,5)   % 1x5 matrix of ones
zeros(1,5)  % 1x5 matrix of zeros
```

O/P (Command Window)

```
Sum of a: 55
Mean of a: 5.50
Median of a: 5.50
Standard Deviation of a: 3.03
Minimum value of a: 1 (at index 10)
Maximum value of a: 10 (at index 1)
Ascending order of a: 1 2 3 4 5 6 7 8 9 10
Descending order of a: 10 9 8 7 6 5 4 3 2 1
Variance of a: 9.17
```

Cumulative sum of a: 10 15 22 24 32 35 39 48 54 55
Dot product of b and c: 56
Cross product of b and c: [-3 6 -3]
Result of d/e: 0.000000
Result of d/e: 0.000000
Result of d/e: 0.333333
Result of d/e: 0.500000
Result of d\e: 2.000000
Result of f/g: 0.500000
Result of f\g: 0.000000
Result of f\g: 1.333333
Result of f\g: 0.000000
Result of f\g: 2.000000

im =

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

ans = 0.9502 0.0344 0.4387 0.3816 0.7655
ans = 2 4 3 5 1
ans = 1 1 1 1 1
ans = 0 0 0 0 0

Ex 2.3
Procedure:

Step 1: Temperature Data Initialization

- **temperatures**: The vector temperatures holds daily temperature data for a month, where each element represents the temperature recorded for a specific day.

Step 2: Trend Analysis

- **avg_temp = mean(temperatures)**: Computes the average temperature for the entire month.
- **days_above_avg = sum(temperatures > avg_temp)**: Counts the number of days where the temperature was above the monthly average by comparing each day's temperature to the average and summing the boolean results.

Step 3: Extreme Days Identification

- **hottest_temp = max(temperatures)**: Identifies the highest temperature in the month (the hottest day).
- **hottest_day = find(temperatures == hottest_temp, 1)**: Finds the day corresponding to the hottest temperature. The find function returns the index of the first occurrence of the maximum temperature.
- **coldest_temp = min(temperatures)**: Identifies the lowest temperature in the month (the coldest day).
- **coldest_day = find(temperatures == coldest_temp, 1)**: Finds the day corresponding to the coldest temperature. The find function returns the index of the first occurrence of the minimum temperature.
- **temp_range = hottest_temp - coldest_temp**: Calculates the temperature range by subtracting the coldest temperature from the hottest temperature.

Step 4: Temperature Consistency

- **temp_std = std(temperatures)**: Computes the standard deviation of the temperatures, which provides a measure of the consistency of the data. A low standard deviation means the temperatures are close to the average, whereas a high standard deviation indicates more variation in the temperatures.

Step 5: Predictive Analysis (Handling Missing Data)

- **if isnan(temperatures(11))**: This checks if the temperature for the 11th day is missing (NaN). If so, linear interpolation is used to estimate the missing value.
- **temperatures(11) = mean([temperatures(10), temperatures(12)])**: If the data for the 11th day is missing, the code fills it with the average of the temperatures on the 10th and 12th days to estimate the missing value.

Step 6: Temperature Anomalies

- **anomaly_threshold = avg_temp + 2 * temp_std**: Defines an anomaly threshold, which is the average temperature plus two times the standard deviation. Any temperature significantly above or below this threshold is considered an anomaly.
- **anomaly_days = find(abs(temperatures - avg_temp) > anomaly_threshold)**: Identifies days where the temperature is greater than the anomaly threshold by comparing the absolute difference between the day's temperature and the average.

Step 7: Displaying the Results

- **fprintf('Average monthly temperature: %.2f°C\n', avg_temp)**: Displays the calculated average temperature for the month.
- **fprintf('Number of days above the monthly average: %d\n', days_above_avg)**: Displays the number of days where the temperature exceeded the average.

- **fprintf('Hottest day: Day %d with %.2f°C\n', hottest_day, hottest_temp)**: Displays the day number and temperature of the hottest day.
- **fprintf('Coldest day: Day %d with %.2f°C\n', coldest_day, coldest_temp)**: Displays the day number and temperature of the coldest day.
- **fprintf('Temperature range: %.2f°C\n', temp_range)**: Displays the range between the hottest and coldest temperatures.
- **fprintf('Standard deviation of temperatures: %.2f°C\n', temp_std)**: Displays the standard deviation of the temperature data.
- **fprintf('Days identified as temperature anomalies: %d ', anomaly_days)**: If anomalies are detected, the days with anomalous temperatures are displayed.
- **fprintf('No temperature anomalies identified.\n')**: If no anomalies are found, a message is displayed.

Code:

```
% Temperature data for a month
temperatures = [23, 25, 26, 24, 23, 28, 29, 30, 28, 29, 31, 30, 29, 28, 27, 26, 25, 25, 24, 24, 23, 22, 23, 23, 24, 24, 25, 25, 26, 26];

% 1. Trend Analysis:
avg_temp = mean(temperatures);
days_above_avg = sum(temperatures > avg_temp);

% 2. Extreme Days:
hottest_temp = max(temperatures);
hottest_day = find(temperatures == hottest_temp, 1); % returns the first occurrence
coldest_temp = min(temperatures);
coldest_day = find(temperatures == coldest_temp, 1);
temp_range = hottest_temp - coldest_temp;

% 3. Temperature Consistency:
temp_std = std(temperatures);

% 4. Predictive Analysis (using linear interpolation for the 11th day as an example):
if isnan(temperatures(11)) % Check if the data for the 11th day is missing
    temperatures(11) = mean([temperatures(10), temperatures(12)]);
end
```

```
% 5. Temperature Anomalies:
anomaly_threshold = avg_temp + 2 * temp_std;
anomaly_days = find(abs(temperatures - avg_temp) > anomaly_threshold);

% Displaying the results:
fprintf('Average monthly temperature: %.2f°C\n', avg_temp);
fprintf('Number of days above the monthly average: %d\n', days_above_avg);
fprintf('Hottest day: Day %d with %.2f°C\n', hottest_day, hottest_temp);
fprintf('Coldest day: Day %d with %.2f°C\n', coldest_day, coldest_temp);
fprintf('Temperature range: %.2f°C\n', temp_range);
fprintf('Standard deviation of temperatures: %.2f°C\n', temp_std);

if ~isempty(anomaly_days)
    fprintf('Days identified as temperature anomalies: ');
    fprintf('%d ', anomaly_days);
    fprintf('\n');
else
    fprintf('No temperature anomalies identified.\n');
end
```

O/P (Command Window)

```
Average monthly temperature: 25.83°C
Number of days above the monthly average: 14
Hottest day: Day 11 with 31.00°C
Coldest day: Day 22 with 22.00°C
Temperature range: 9.00°C
Standard deviation of temperatures: 2.51°C
No temperature anomalies identified.
```

Ex 2.4

Procedure:

Step 1: Temperature Matrix Initialization

- **temperature_matrix**: Initializes a 4x4 matrix representing temperatures across different locations or sensors:

25, 30, 35, 40

26, 29, 33, 39

24, 28, 34, 38

23, 27, 32, 37

Step 2: Setting Border Temperatures to 20°C

- **temperature_matrix(1, :) = 20**: Sets all the elements in the first row of the matrix to 20°C.
- **temperature_matrix(end, :) = 20**: Sets all the elements in the last row of the matrix to 20°C (i.e., the 4th row in this case).
- **temperature_matrix(:, 1) = 20**: Sets all the elements in the first column of the matrix to 20°C.
- **temperature_matrix(:, end) = 20**: Sets all the elements in the last column of the matrix to 20°C (i.e., the 4th column).

After this operation, the matrix will look like this:

20, 20, 20, 20

20, 29, 33, 20

20, 28, 34, 20

20, 20, 20, 20

Step 3: Normalizing the Matrix

- **min_temp = min(temperature_matrix(:))**: Finds the minimum value in the entire matrix. Here, the minimum value is 20°C.
- **max_temp = max(temperature_matrix(:))**: Finds the maximum value in the matrix, which is 34°C.
- **normalized_matrix = (temperature_matrix - min_temp) / (max_temp - min_temp)**: Normalizes the matrix by subtracting the minimum value from each element and then dividing by the range (maximum - minimum). This scales all the values in the matrix between 0 and 1.

The normalized matrix will look something like this:

0, 0, 0, 0

0, 0.45, 0.87, 0

0, 0.40, 1.00, 0

0, 0, 0, 0

Step 4: Identifying the Sensor with the Highest Temperature

- **[max_val, max_idx] = max(normalized_matrix(:))**: Finds the highest value (after normalization) in the matrix. The variable max_val stores the value, and max_idx stores its linear index.
- **[row_idx, col_idx] = ind2sub(size(normalized_matrix), max_idx)**: Converts the linear index (max_idx) to a row and column index (row_idx and col_idx). This identifies the position of the sensor with the highest temperature.

In this case, the sensor with the highest temperature is at row 3 and column 3 with a normalized value of 1.00.

Step 5: Rotating the Matrix 90 Degrees Clockwise

- **rotated_matrix = rot90(normalized_matrix, -1)**: Rotates the matrix 90 degrees clockwise. The function rot90 rotates the matrix counterclockwise by default, so rot90(..., -1) is used to rotate it clockwise.

Step 6: Displaying the Results

- **disp('Corrected Matrix:')**: Displays the normalized matrix.
- **disp(['Sensor with highest temperature is at row ', ...])**: Displays the location of the sensor with the highest temperature after normalization.
- **disp('Rotated Matrix:')**: Displays the rotated matrix.

Code:

```
% Temperature matrix

temperature_matrix = [
   25, 30, 35, 40;
   26, 29, 33, 39;
   24, 28, 34, 38;
   23, 27, 32, 37;
];

% 1. Set the border temperatures of the matrix to 20°C.
temperature_matrix(1, :) = 20;       % First row
temperature_matrix(end, :) = 20;     % Last row
temperature_matrix(:, 1) = 20;       % First column
temperature_matrix(:, end) = 20;     % Last column
```

```
% 2. Normalize the entire matrix.
min_temp = min(temperature_matrix(:));
max_temp = max(temperature_matrix(:));
normalized_matrix = (temperature_matrix - min_temp) / (max_temp -
min_temp);

% 3. Identify the sensor with the highest temperature (after normalization).
[max_val, max_idx] = max(normalized_matrix(:));
[row_idx, col_idx] = ind2sub(size(normalized_matrix), max_idx);

% 4. Rotate the matrix 90 degrees clockwise.
rotated_matrix = rot90(normalized_matrix, -1);

% Display results
disp('Corrected Matrix:');
disp(normalized_matrix);

disp(['Sensor with highest temperature is at row ', num2str(row_idx), ' and
column ', num2str(col_idx), ' with value: ', num2str(max_val)]);

disp('Rotated Matrix:');
disp(rotated_matrix);
```

O/P (Command Window)

```
ex2_mat
Corrected Matrix:
         0         0         0         0
         0    0.6429    0.9286         0
         0    0.5714    1.0000         0
         0         0         0         0

Sensor with highest temperature is at row 3 and column 3 with value: 1
Rotated Matrix:
         0         0         0         0
         0    0.5714    0.6429         0
         0    1.0000    0.9286         0
         0         0         0         0
```

Expt No. 3	LOOPS	Date of Expt:

Ex 3.1
Procedure:

1. FOR Loop
Example 1: Simple Iteration

1. **Initialize the loop**: Set up a for loop that iterates over a defined range of numbers (e.g., 1 through 10).
2. **Display each value**: During each iteration, display the current value of the loop variable.
3. **End the loop**: The loop finishes when the last value in the range has been processed.

Example 2: Dynamic Array Update

1. **Initialize an array**: Create an array with specified initial values (e.g., all elements set to 1).
2. **Start loop**: Begin the loop to iterate over a subset of the array's indices.
3. **Update array values**: For each iteration, update the current element based on a formula (e.g., set it to twice the previous element).
4. **End the loop**: The loop ends after updating all specified elements.

---

2. WHILE Loop
Example 1: Simple Counter

1. **Initialize a counter**: Set a variable to hold the initial count value.
2. **Check the loop condition**: Set a condition to control the loop (e.g., while the counter is less than or equal to 10).
3. **Display the count**: For each iteration, display the current value of the counter.
4. **Increment the counter**: Increase the counter by a specified amount after each iteration.
5. **End the loop**: The loop ends once the condition is no longer true.

Example 2: Factorial Calculation

1. **Initialize variables**: Set an initial value for the factorial and a variable to hold the number.
2. **Check the loop condition**: Continue the loop while the calculated factorial is less than a certain large number (e.g., 10^100).
3. **Update the factorial**: Multiply the current factorial by the next integer in each iteration.
4. **End the loop**: The loop terminates when the factorial exceeds the specified limit.

---

3. NESTED LOOP

Example 1: Displaying Pairs of Values

1. **Set up the outer loop**: Define the first loop that iterates over a set of values (e.g., 1 to 3).
2. **Set up the inner loop**: Within the outer loop, define another loop that iterates over its own set of values.
3. **Display paired values**: For each pair of values from the outer and inner loops, display them together.
4. **End the loops**: Both loops finish when all combinations of values have been processed.

Example 2: Populating a Matrix

1. **Initialize a matrix**: Create a matrix with the desired dimensions and initial values (e.g., all zeros).
2. **Set up the outer loop**: Iterate over the rows of the matrix.
3. **Set up the inner loop**: For each row, iterate over the columns of the matrix.
4. **Update matrix elements**: Apply a formula to calculate each element based on its row and column indices.
5. **End the loops**: The nested loops finish when the matrix is fully populated.

---

4. Controlling Loop Execution

Example 1: Using break

1. **Initialize the loop**: Set up a loop to iterate over a range of values.
2. **Check a condition**: Inside the loop, check if a certain condition is met (e.g., if a variable reaches a specific value).
3. **Exit the loop**: If the condition is met, use the break statement to exit the loop immediately.
4. **End the loop**: The loop finishes once it is exited or the range of values is exhausted.

Example 2: Using continue

1. **Initialize the loop**: Set up a loop to iterate over a range of values.
2. **Check a condition**: Inside the loop, check if a certain condition is met (e.g., if a variable reaches a specific value).
3. **Skip to the next iteration**: If the condition is met, use the continue statement to skip the current iteration and move to the next one.
4. **End the loop**: The loop finishes when all iterations have been completed, except for those that were skipped.

Code:

1. FOR LOOP

```
for index = 1:10 % Define a for loop that iterates over numbers 1 through 10
   disp(index) % Display the current value of 'index'
```

```
end
```

O/P:

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

```
x = ones(1,10);              % Initializes a 1x10 vector with all elements set to 1
for n = 2:6                  % Begins a loop for n ranging from 2 to 6
   x(n) = 2 * x(n - 1);      % Sets the nth element of x as double the (n-1)th
element
end                          % Ends the for loop
```

O/P:

```
x = 1  2  4  8  16  32  1  1  1  1
```

2. WHILE LOOP

```
count = 1; % Initialize the 'count' variable with a value of 1
while count <= 10  % While the 'count' is less than or equal to 10, execute the
loop
   disp(count) % Display the current value of 'count'
   count = count + 1; % Increment 'count' by 1
end
```

```
n = 1;                       % Sets n to 1
nFactorial = 1;              % Initializes factorial of n to 1
while nFactorial < 1e100     % Begins a loop while n's factorial is less than
10^100
   n = n + 1;                % Increments n by 1
   nFactorial = nFactorial * n;% Multiplies current nFactorial with n
end
```

O/P:
This loop is computing factorial values, and it stops once it hits a factorial larger than 1010010100. It doesn't produce a displayed output, but at the end of this loop, **n** will be the smallest integer such that **n!** is greater than 1010010100

3. NESTED LOOP

```
for i = 1:3 % Outer loop iterating from 1 to 3 for variable 'i'
    for j = 1:3  % Inner loop iterating from 1 to 3 for variable 'j'
        disp(['i = ', num2str(i), ', j = ', num2str(j)]) % Display the current values of
i, j
    end
end
```

O/P:

```
i = 1, j = 1
i = 1, j = 2
i = 1, j = 3
i = 2, j = 1
i = 2, j = 2
i = 2, j = 3
i = 3, j = 1
i = 3, j = 2
i = 3, j = 3
```

```
A = zeros(5,100);              % Initializes a 5x100 matrix with all zeros
for m = 1:5                    % Begins an outer loop for m ranging from 1 to 5
    for n = 1:100              % Begins an inner loop for n ranging from 1 to 100
        A(m, n) = 1/(m + n - 1); % Sets the (m,n) element of matrix A based on
given formula
    end                        % Ends the inner for loop
end                            % Ends the outer for loop
```

O/P:
This code initializes a 5x100 matrix **A** and populates it according to the given formula. It doesn't produce a displayed output, but the matrix **A** will be filled with the results of the formula.

5. CONTROLLING LOOP EXECUTION
- USING BREAK

```
% Define a for loop that iterates over numbers 1 through 10
for i = 1:10
    if i == 5 % If the value of 'i' is equal to 5
```

```
    break; % Exit the loop
  end
  disp(i) % Display the current value of 'i' (this won't be executed when i == 5)
end
```

O/P:

1
2
3
4

- USING CONTINUE

```
% Define a for loop that iterates over numbers 1 through 10
for i = 1:10
  if i == 5 % If the value of 'i' is equal to 5
    continue; % Skip to the next iteration of the loop
  end
  disp(i) % Display the current value of 'i' (this will skip displaying the number 5)
end
```

O/P:

1
2
3
4
6
7
8
9

Ex 3.2
Procedure:

Step 1: Load the Image

- **img = imread('sample.jpeg')**: This reads an image file named 'sample.jpeg' into the variable img. The image is stored as a 3D matrix where the dimensions are:
  - m = number of rows (height of the image)
  - n = number of columns (width of the image)

- c = number of color channels (typically 3 for RGB images)

Step 2: Initialize an Empty Matrix for the Grayscale Image

- **[m, n, c] = size(img)**: Retrieves the size of the image matrix img. This stores the number of rows (m), columns (n), and color channels (c) in the variables.
- **grayscale_img = zeros(m, n)**: Initializes a 2D matrix grayscale_img of size m by n, filled with zeros, which will store the grayscale values of the image. This matrix has no color channels as it will only store intensity values for grayscale.

Step 3: Convert the Image to Grayscale Using Loops

- A nested loop is used to process each pixel in the image:
  - **for i = 1**

: Loops over each row of the image.

  - **for j = 1**

: Loops over each column of the image.

  - Inside the loop, each pixel's Red (R), Green (G), and Blue (B) values are extracted from the RGB image, and the grayscale value is computed using the formula:
    - **grayscale = 0.299*R* + *0.587*G + 0.114*B**: This is the standard formula for converting an RGB image to grayscale, giving more weight to the Green channel, which the human eye perceives as brighter.
    - **grayscale_img(i, j) = 0.299 * img(i, j, 1) + 0.587 * img(i, j, 2) + 0.114 * img(i, j, 3)**: For each pixel at position (i, j), this computes the grayscale intensity and stores it in the corresponding location in the grayscale_img matrix.

Step 4: Convert Grayscale Image to uint8 Format

- **grayscale_img = uint8(grayscale_img)**: Converts the grayscale_img matrix, which is currently in double precision, into the uint8 format. Images in MATLAB are typically represented in uint8 format, where pixel values range from 0 to 255.

Step 5: Invert the Grayscale Image

- **inverted_img = 255 - grayscale_img**: Inverts the grayscale image by subtracting each pixel value from 255. Inversion means that dark pixels (near 0) become bright (near 255), and bright pixels (near 255) become dark (near 0).

Step 6: Display the Original, Grayscale, and Inverted Images

- **subplot(1, 3, 1)**: Creates a subplot layout where 3 images will be displayed in one row.
  - **imshow(img)**: Displays the original RGB image in the first subplot.
  - **title('Original Image')**: Adds the title "Original Image" to the first subplot.
- **subplot(1, 3, 2)**: Moves to the second subplot.
  - **imshow(grayscale_img)**: Displays the grayscale image.
  - **title('Grayscale Image')**: Adds the title "Grayscale Image" to the second subplot.
- **subplot(1, 3, 3)**: Moves to the third subplot.
  - **imshow(inverted_img)**: Displays the inverted grayscale image.
  - **title('Inverted Image')**: Adds the title "Inverted Image" to the third subplot.

Summary:

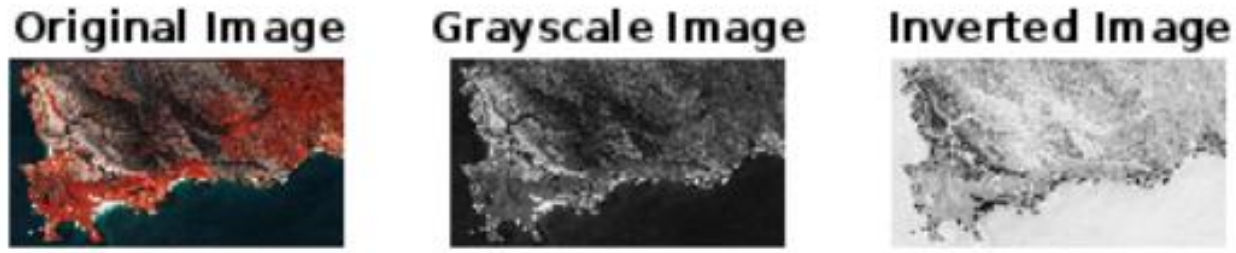
1. The image is loaded into memory and its dimensions are determined.
2. A grayscale image is created using loops to apply the grayscale conversion formula to each pixel.
3. The grayscale image is converted to the appropriate format (uint8).
4. The grayscale image is inverted by subtracting pixel values from 255.
5. The original, grayscale, and inverted images are displayed side by side for comparison.

Code:

```
% Load an image
img = imread('sample.jpeg');

% Initialize an empty matrix for the grayscale image
[m, n, c] = size(img);
grayscale_img = zeros(m, n);

% Convert the image to grayscale using loops
for i = 1:m
   for j = 1:n
      % Using the standard formula for grayscale conversion:
      % grayscale = 0.299*R + 0.587*G + 0.114*B
      grayscale_img(i, j) = 0.299 * img(i, j, 1) + 0.587 * img(i, j, 2) + 0.114 *
img(i, j, 3);
```

```
    end
end

% Convert the grayscale image to uint8 format
grayscale_img = uint8(grayscale_img);

% Invert the grayscale image
inverted_img = 255 - grayscale_img;

% Display the original, grayscale, and inverted images
subplot(1, 3, 1);
imshow(img);
title('Original Image');

subplot(1, 3, 2);
imshow(grayscale_img);
title('Grayscale Image');

subplot(1, 3, 3);
imshow(inverted_img);
title('Inverted Image');
```

O/P



Expt No. 4	ERRORS	Date of Expt:

Ex 4.1
Procedure:

Step 1: Define the number of terms:

- Set n = 4, which determines how many terms will be used in the series expansion to approximate the exponential function.

Step 2: Set the value of x:

- Define x = 0.3, the value for which the exponential function e^x will be calculated.

Step 3: Initialize the computed exponential value:

- Set expval = 1.0, which is the starting value for the series expansion of e^x. This corresponds to the first term of the series (which is 1 for any exponential).

Step 4: Initialize the first term of the series:

- Set currentterm = 1.0, which represents the first term in the series expansion for e^x.

Step 5: Start a loop to compute the series terms:

- A for loop is used to iterate n times (from 1 to n):
    1. **Update the current term**: For each iteration, the next term in the series is calculated by multiplying the previous term (currentterm) by x and dividing by the current iteration index i.
    2. **Update the exponential approximation**: Add the newly calculated term to expval, which accumulates the total value of the exponential approximation.

Step 6: Compute the true value of e^x:

- After the loop, use MATLAB's built-in function exp to compute the exact value of $e^{0.3}$ and store it in trueval.

Step 7: Calculate the error:

- Compute the absolute error between the true value (trueval) and the approximated value (expval) using the formula:

$$error = |trueval - expval|$$

Step 8: Display the results:

- Use the fprintf function to print the following:
    - The true value of $e^{0.3}$ (trueval)

- The computed approximated value (expval)
- The absolute error between the two values (error)

Code:

```
n=4;                    % Number of terms to use in the series expansion
x=0.3;                  % The value for which we want to compute the exponential
expval=1.0;             % Initialize the computed exponential value starting with the first term of the series
currentterm=1.0;        % Initialize the first term in the exponential series

for i = 1:n             % Loop n times to compute the first n terms of the series
   currentterm = currentterm * x/i;  % Calculate the next term in the series
   expval=expval+currentterm;        % Update the running total of the exponential approximation
end

trueval=exp(0.3);       % Compute the true value of e^0.3 using MATLAB's built-in function
error = abs(trueval - expval);      % Calculate the absolute error between the true value and our approximation

fprintf('True Value: %f\n', trueval);
fprintf('Exponential Value: %f\n', expval);
fprintf('Error: %f\n', error);
```

O/P:

```
True Value: 1.349859
Exponential Value: 1.349838
Error: 0.000021
```

Ex 4.2
Procedure:

Taylor Series Expansion (First Example):

1. **Initialize the Number of Terms**:
   - Set n = 4, which determines the number of terms in the Taylor series expansion for exe^xex.
2. **Set the Value of xxx**:
   - Define x = 0.3, the value for which the exponential function e^x will be evaluated.

3. **Initialize the Exponential Value**:
    - Set expval = 1.0 to begin the approximation, which corresponds to the 0th term in the Taylor series (since e^x starts with 1).
4. **Initialize the Current Term**:
    - Set currentterm = 1.0, which will be used to calculate each subsequent term in the series.
5. **Loop to Compute Taylor Series Terms**:
    - Start a loop to compute n terms of the Taylor series expansion:
        1. **Update the Current Term**: For each iteration, calculate the next term in the series using the previous term, multiplying by x/i, where i is the current index.
        2. **Update the Exponential Value**: After calculating each term, add it to the current sum stored in expval(i+1).
6. **Calculate the True Value**:
    - Use MATLAB's built-in exp function to compute the actual value of e^x at x=0.3x = 0.3x=0.3.
7. **Compute the Error**:
    - Calculate the absolute difference between the true value (trueval) and the approximated value (expval) to obtain the error for each iteration.
8. **Display Results**:
    - Print the true value, the approximated value at each iteration, and the associated error after each term is added.

Taylor Series Expansion with Multiple Values of x (Second Example):

1. **Initialize the Number of Terms**:
    - Set n = 4, which determines the number of terms in the Taylor series expansion.
2. **Define Multiple Values of xxx**:
    - Create a vector xall = [0.1, 0.5, 0.01, 0.02], which contains the different values of xxx for which the exponential function exe^xex will be computed.
3. **Initialize a Vector of Indices**:
    - Define vec = [1:n] to represent the powers of xxx in the Taylor series.
4. **Initialize the Error Vector**:
    - Set Error = [] to store the computed error for each value of xxx.

5. **Loop Over All Values of xxx**:
    - Start a loop that iterates over all elements of xall:
        1. **Select the Current xxx**: Pick the current xxx value from xall.
        2. **Compute Terms of the Series**: Use the powers of x and the cumulative product to compute each term of the Taylor series.
        3. **Cumulative Sum of the Series**: Use cumsum to compute the running total for the Taylor series.
        4. **Compute the True Value**: Use MATLAB's exp function to compute the actual value of e^x for the current xxx.
        5. **Calculate the Error**: Compute the absolute error between the true value and the Taylor series approximation (using the last value from the cumulative sum).
        6. **Store the Error**: Append the error to the Error array.
6. **Plot the Error**:
    - Plot the error as a function of the values in xall to visually compare the approximation errors for different xxx values.
7. **Display Results**:
    - Print the true value, the approximated value for each value of xxx, and the error for each approximation.

Code:

```
n=4;                % Number of terms in the Taylor series expansion
x=0.3;              % Value at which we want to evaluate the exponential function
expval=1.0;         % Initialize the Taylor series expansion result to 1.0
currentterm=1.0;    % Initialize the current term in the Taylor series to 1.0 (0th term)

% Loop to compute the Taylor series approximation for exp(0.3)
for i = 1:n
   % Calculate the next term in the Taylor series using the previous term
   currentterm = currentterm * x/i;
   % Update the result array by adding the new term to the previous sum
   expval(i+1) = expval(i) + currentterm;
end

trueval = exp(0.3);
error = abs(trueval - expval);
```

```
fprintf('True Value: %f\n', trueval);
fprintf('Exponential Value: %f\n', expval);
fprintf('Error: %f\n', error);
```

O/P:

```
True Value: 1.349859
Exponential Value: 1.000000
Exponential Value: 1.300000
Exponential Value: 1.345000
Exponential Value: 1.349500
Exponential Value: 1.349838
Error: 0.349859
Error: 0.049859
Error: 0.004859
Error: 0.000359
Error: 0.000021
```

Ex 4.3
Procedure:

Maclaurin Series Vector Approximation

1. **Initialize the Number of Terms**:
   - Set n = 4, which represents the number of terms in the Maclaurin series expansion for approximating e^x.
2. **Define the Vector of xxx Values**:
   - Create a vector xall = [0.1, 0.5, 0.01, 0.02], which contains the different values of xxx for which the exponential function e^x will be computed using the series expansion.
3. **Initialize the Vector of Indices for the Series**:
   - Set vec = [1:n], which creates a vector from 1 to n. This represents the powers of xxx and the factorial denominators in the Maclaurin series expansion.
4. **Initialize an Empty Error Array**:
   - Set Error = [], which will store the approximation error for each x value in xall.
5. **Loop Through Each Value of xxx**:
   - Start a loop that iterates over all elements in the xall vector:

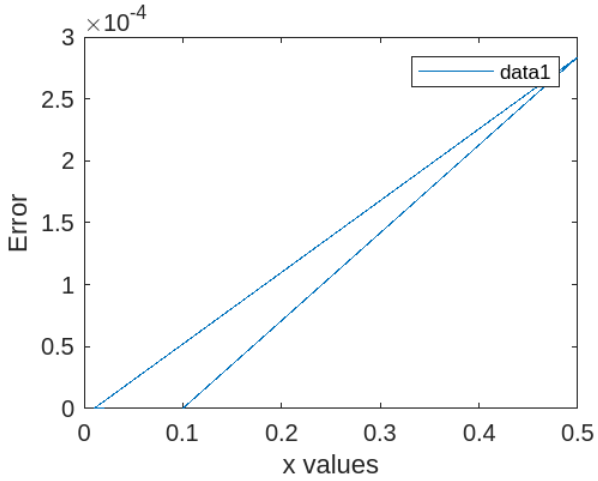
1. **Select the Current xxx**: In each iteration, select the current value of x from xall.
2. **Calculate the Terms of the Series**: Compute the terms of the Maclaurin series using the formula:
    - $\frac{x^k}{k!}$, where k is the index from 1 to n, and calculate this for each value of x.
3. **Cumulative Sum for the Series**: Use cumsum to compute the cumulative sum of the terms, which approximates e^x. The approximation is stored in expval.
4. **Compute the True Value**: Use MATLAB's built-in exp(x) function to calculate the actual value of e^x for the current xxx.
5. **Compute the Error**: Calculate the absolute error between the true value (trueval) and the final term in the cumulative sum (expval(end)).
6. **Store the Error**: Append the computed error to the Error array for the current value of x.

6. **Plot the Error**:
    - After looping through all xxx values, plot the Error array against the values in xall to visualize how the approximation error changes with different values of xxx.
    - Set labels for the x-axis (x values) and the y-axis (Error) on the plot.
7. **Display the Results**:
    - After processing all xxx values, print the true value of exe^xex, the final value of the Maclaurin series approximation, and the error for the last iteration.

Code:

```
%maclarin_vector
n=4;
xall=[0.1, 0.5, 0.01, 0.02];
vec = [1:n];
Error=[];

for i = 1:length(xall)
   x = xall(i);  % Picking the i-th element of xall
```

```
    terms = x.^vec ./ cumprod(vec);  % Calculating each term of the Taylor series
    expval = 1 + cumsum(terms);  % Cumulative sum for the exponential
approximation
    trueval = exp(x);  % Actual exponential value
    error = abs(trueval - expval(end));  % Error
    Error = [Error; error];  % Storing the error for each x value
end

plot(xall, Error);  % Plotting the errors
xlabel('x values');  % Fixing the xlabel and ylabel
ylabel('Error');

fprintf('True Value: %f\n', trueval);
fprintf('Exponential Value: %f\n', expval);
fprintf('Error: %f\n', error);
```



O/P:

True Value: 1.020201340027
Exponential Value: 1.020000000000
Exponential Value: 1.020200000000
Exponential Value: 1.020201333333
Exponential Value: 1.020201340000
Error: 0.000000000027

Ex 4.3
Procedure:

Estimating the Square Root of 2 Using Heron's Method (Iterative Approach)

1. **Initialize the Estimate**:
    - Set the initial guess for the square root of 2:
        - $x = 0.5$, which is an arbitrary starting point for the iterative process.
2. **Start the Iterative Loop**:
    - Use a loop to refine the estimate of the square root of 2. In this example, the loop runs for 7 iterations.
3. **Update the Estimate Using Heron's Method**:

- In each iteration, calculate the next approximation (xnew) for the square root of 2 using Heron's method:

$$x_{new} = \frac{1}{2}\left(x + \frac{1}{2}\right)$$

- This formula combines the current estimate with the result of dividing 2 by the current estimate to get closer to the true value of $\sqrt{2}$

4. **Calculate the Error**:
   - After computing the new estimate, calculate the absolute difference between the current estimate (x) and the new estimate (xnew):

$$err = |x - x_{new}|$$

   - This measures how much the estimate is changing with each iteration, providing an indication of convergence.
5. **Update the Estimate**:
   - Set the current estimate xxx to the newly computed value x_{new} so that the next iteration can further refine the estimate.
6. **Repeat the Process**:
   - The loop continues for 7 iterations, progressively refining the estimate of $\sqrt{2}$.
7. **Display Results**:
   - After the loop, you can use the fprintf function to print the results, such as the final estimate of the square root of 2, the true value, and the error. However, the current code has placeholders for printing exponential values instead of the square root estimate.

Notes:

- The loop will progressively refine the estimate for $\sqrt{2}$ with each iteration. Typically, the more iterations you perform, the closer the estimate will get to the true value of $\sqrt{2}$, which is approximately 1.414213562.
- The current code uses 7 iterations, which is generally sufficient for a good approximation of the square root of 2.

Code:

```
% Initialize x with an initial guess for the square root of 2
x = 0.5;

% Start a loop that will iterate 7 times to refine the estimate
for i = 1:7
   % Compute the next estimate for the square root of 2 using Heron's method
```

```
    xnew = 1/2 * (x + 2/x);

    % Calculate the absolute difference between the current estimate and the new
estimate
    % This gives an indication of how much the estimate is changing with each
iteration
    err = abs(x - xnew);

    % Update the estimate x for the next iteration
    x = xnew;
end

fprintf('True Value: %.12f\n', trueval);
fprintf('Exponential Value: %.12f\n', expval);
fprintf('Error: %.12f\n', error);
```

O/P:

```
True Value: 1.020201340027
Exponential Value: 1.020000000000
Exponential Value: 1.020200000000
Exponential Value: 1.020201333333
Exponential Value: 1.020201340000
Error: 0.000000000027
```

Ex 4.4
Procedure:

Approximating the Square Root of 2 Using Heron's Method (While Loop Approach)

1. **Initialize the Estimate**:
    - Set the initial guess for the square root of 2:
        - $x = 0.5$, which serves as the starting point for the iterative process.
2. **Set the Absolute Tolerance**:
    - Define the absolute tolerance atol = 1.0e-4, which determines how close the approximation needs to be before the loop terminates.
    - This tolerance controls when the algorithm stops refining the estimate. Once the error is smaller than this tolerance, the approximation is considered sufficiently accurate.
3. **Initialize the Error**:

- Set the error err = 1 to ensure the loop starts. Initially, the error is set to a value larger than the tolerance.

4. **Start the While Loop**:
   - The loop will continue running as long as the error (err) is greater than the absolute tolerance (atol).
5. **Update the Estimate Using Heron's Method**:
   - Inside the loop, calculate the next approximation (xnew) for the square root of 2 using Heron's method:

$$x_{new} = \frac{1}{2}\left(x + \frac{1}{2}\right)$$

   - This formula combines the current estimate with the result of dividing 2 by the current estimate to get closer to the true value of $\sqrt{2}$
6. **Calculate the Error**:
   - After computing the new estimate, calculate the absolute difference between the current estimate (x) and the new estimate (xnew):

$$err = |x - x_{new}|$$

   - This measures how much the estimate is changing with each iteration, providing an indication of convergence.
7. **Update the Estimate**:
   - Set the current estimate xxx to the newly computed value x_{new} so that the next iteration can further refine the estimate.
8. **Repeat the Process**:
   - The loop continues for 7 iterations, progressively refining the estimate of $\sqrt{2}$.
9. **Display Results**:
   - After the loop, you can use the fprintf function to print the results, such as the final estimate of the square root of 2, the true value, and the error. However, the current code has placeholders for printing exponential values instead of the square root estimate.

Code:

```
%% Heron's algorithm using a while loop to approximate the square root of 2

x = 0.5; % Initialize x with an initial guess for the square root of 2
atol = 1.0e-4; % Set the absolute tolerance for convergence
err = 1; % Initialize the error to a non-zero value to ensure the loop starts
```

```
% Continue refining the estimate as long as the error is greater than the
tolerance
while (err > atol)
    % Compute the next estimate for the square root of 2 using Heron's method
    xnew = 1/2 * (x + 2/x);

    % Calculate the absolute difference between the current estimate and the new
estimate
    err = abs(x - xnew);

    % Update the estimate x for the next iteration
    x = xnew;
end

% Display the final estimate for the square root of 2 and the error
fprintf('Estimate for the square root of 2: %.12f\n', x);
fprintf('Final error: %.12f\n', err);
```

O/P:

Estimate for the square root of 2: 1.414213562525
Final error: 0.000020723415

Ex 4.5
Procedure:

Approximating $e^{0.1}$ Using the Compound Interest Formula

1. **Set Constants**:
    - Define the target exponent:
        - $a = 0.1$, which represents the exponent for which you are calculating $\boldsymbol{e^{0.1}}$.
    - Define the step size for the approximation:
        - $h = 0.01$, which is the incremental step size used in the iterative approximation process.
    - Calculate the number of iterations required:
        - $n = a/h$, which calculates how many iterations are needed based on the step size h.
2. **Calculate the True Value**:
    - Use MATLAB's built-in function exp(a) to calculate the true value of $e^{0.1}$, which is stored in the variable truval.

- This true value will be used later to compute the error between the actual value and the approximation.

3. **Initialize the Approximation**:
   - Set expval = 1, which is the starting value for the iterative approximation. This corresponds to the initial value of the exponential function when using the compound interest method.
4. **Iterate to Approximate $e^{0.1}$:**
   - Start a loop that runs n times (where n = a/h).
   - In each iteration, multiply the current approximation expval by $(1 + h)$, simulating the compound interest formula:

$$expval = expval \times (1 + h)$$

   - This process iteratively builds the approximation of $e^{0.1}$.
5. **Calculate the Error**:
   - After completing the iterations, calculate the absolute error between the true value (truval) and the approximated value (expval):

$$err = | truval - expval |$$

   - The error represents how far the approximation deviates from the true value.
6. **Display the Results**:
   - Use disp and num2str to display the results:
     - The true value of $e^{0.1}$ (calculated with MATLAB's built-in function).
     - The approximated value obtained using the iterative method.
     - The absolute error between the true value and the approximation.

Code:

```
% Set constants
a=0.1;          % Target exponent for e
h=0.01;         % Step size for approximation
n=a/h;          % Number of iterations required

% Calculate the true value of e^0.1 for error comparison
truval=exp(a);

% Initialize the approximation value
expval=1;       % Start value for iterative approximation
```

```
% Iterate to approximate e^0.1 using the compound interest formula
for i =1:n
    expval = expval*(1+h);
end

% Calculate the absolute error between true and approximated value
err = abs(truval - expval);

% Display the variables at the end
disp('Results:');
disp(['True Value (truval) = ', num2str(truval)]);
disp(['Approximated Value (expval) = ', num2str(expval)]);
disp(['Absolute Error (err) = ', num2str(err)]);
```

O/P:

```
Results:
True Value (truval) = 1.1052
Approximated Value (expval) = 1.1046
Absolute Error (err) = 0.00054879
```

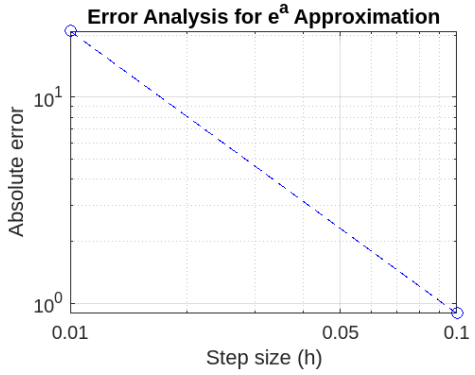
Ex 4.6
Procedure:
Code:

```
a = 0.1; % Define the value of 'a'
trueval = exp(a); % Calculate the true value of e^a

expval = 1; % Initialize the expval to 1 (assuming we're trying to calculate
exp(a) using a simple compound interest approximation)


% Preallocate arrays for efficiency
hall = zeros(1,2);
errall = zeros(1,2);

% Start a loop for two iterations
for i = 1:2
    % Compute the step size 'h'
    h = 10^(-i);

    % Update the expval using the compound interest approximation
```

```
    expval = expval * (1 + a/h);

    % Compute the absolute error between true value and approximate value
    err = abs(trueval - expval);

    % Store the values of 'h' and 'err' in arrays
    hall(i) = h;
    errall(i) = err;
end

% Display the arrays at the end
disp(['hall = ', num2str(hall)]);
disp(['errall = ', num2str(errall)]);

% Plot error against h on a log-log scale
loglog(hall, errall, '--bo');

% Modify x-axis ticks to include more gridlines
set(gca, 'XTick', [10^-2, 5*10^-2, 10^-1, 5*10^-1, 1]);

% Enable the grid
grid on;

% Add labels for clarity
xlabel('Step size (h)');
ylabel('Absolute error');
title('Error Analysis for e^a Approximation');

% Display the absolute errors for each step size
for i = 1:length(hall)
    disp(['Absolute error for h = ', num2str(hall(i)), ': ', num2str(errall(i))]);
end
```

O/P:

```
hall = 0.1        0.01
errall = 0.894829      20.8948
Absolute error for h = 0.1: 0.89483
Absolute error for h = 0.01: 20.8948
```



Ex 4.7
Procedure:

Approximating Sin, Cos, and Tan Using Taylor Series

1. **Set the Target Value**:
   - Define $a = 0.1$, which is the value for which the sine, cosine, and tangent functions will be approximated.
2. **Calculate the True Values**:
   - Use MATLAB's built-in trigonometric functions to calculate the exact values for $sin(a), cos(a), and\ tan(a)$, storing them in truval_sin, truval_cos, and truval_tan, respectively.
3. **Approximate $\boldsymbol{sin(a)}$ Using Taylor Series**:
   - Use the first three terms of the Taylor series expansion for
   $$sin(a) \approx a - \frac{a^3}{3!} + \frac{a^5}{5!}$$
   - This provides a polynomial approximation of $sin(a)$, stored in approxVal_sin.
4. **Approximate $\boldsymbol{cos}$ $\boldsymbol{(a)}$ Using Taylor Series**:
   - Use the first three terms of the Taylor series expansion for
   $$\cos(a) \approx 1 - \frac{a^2}{2!} + \frac{a^4}{4!}$$
   - This provides a polynomial approximation of $cos(a)$, stored in approxVal_cos.
5. **Approximate $\boldsymbol{tan}$ $\boldsymbol{(a)}$**
   - Use the previously calculated approximations of $sin(a), cos(a), and\ tan(a)$
   $$\tan(a) \approx \frac{\sin(a)}{\cos(a)}$$
   - Store this value in approxVal_tan.
6. **Calculate the Absolute Errors**:
   - Compute the absolute error between the true values and the approximated values for $sin(a), cos(a), and\ tan(a)$:
   $$error = |\ True\ Value - Approximated\ Value\ |$$
   - These errors are stored in err_sin, err_cos, and err_tan.
7. **Display the Results**:
   - Use the disp function to print:
     - The true values $sin(a), cos(a), and\ tan(a)$,
     - The approximated values using the Taylor series expansions,

- The absolute errors for each function.

Summary:
This procedure uses the first three terms of the Taylor series to approximate $sin(a), cos(a), and\ tan(a)$, for a given $a = 0.1$. The true values are calculated using MATLAB's built-in trigonometric functions, and the absolute errors between the true and approximated values are computed and displayed.

Code:

```
% Set constant
a = 0.1;  % Target value

% Calculate the true values for sin, cos, and tan of a
truval_sin = sin(a);
truval_cos = cos(a);
truval_tan = tan(a);

% Approximate sin(a) using the first 3 terms of its Taylor series
approxVal_sin = a - (a^3)/factorial(3) + (a^5)/factorial(5);

% Approximate cos(a) using the first 3 terms of its Taylor series
approxVal_cos = 1 - (a^2)/factorial(2) + (a^4)/factorial(4);

% Approximate tan(a) using the ratio of approximated sin(a) and cos(a)
approxVal_tan = approxVal_sin / approxVal_cos;

% Calculate the absolute errors
err_sin = abs(truval_sin - approxVal_sin);
err_cos = abs(truval_cos - approxVal_cos);
err_tan = abs(truval_tan - approxVal_tan);

% Display the results
disp('Results:');
disp(['a = ', num2str(a)]);

disp('SIN:');
disp(['True Value of sin(a) = ', num2str(truval_sin)]);
disp(['Approximated Value of sin(a) = ', num2str(approxVal_sin)]);
disp(['Absolute Error for sin = ', num2str(err_sin)]);
disp('---');

disp('COS:');
```

```
disp(['True Value of cos(a) = ', num2str(truval_cos)]);
disp(['Approximated Value of cos(a) = ', num2str(approxVal_cos)]);
disp(['Absolute Error for cos = ', num2str(err_cos)]);
disp('---');

disp('TAN:');
disp(['True Value of tan(a) = ', num2str(truval_tan)]);
disp(['Approximated Value of tan(a) = ', num2str(approxVal_tan)]);
disp(['Absolute Error for tan = ', num2str(err_tan)]);
```

O/P:

```
Results:
a = 0.1
SIN:
True Value of sin(a) = 0.099833
Approximated Value of sin(a) = 0.099833
Absolute Error for sin = 1.9839e-11
---
COS:
True Value of cos(a) = 0.995
Approximated Value of cos(a) = 0.995
Absolute Error for cos = 1.3886e-09
---
TAN:
True Value of tan(a) = 0.10033
Approximated Value of tan(a) = 0.10033
Absolute Error for tan = 1.2009e-10
```

Expt No. 5	NUMERICAL DIFERENTIATION	Date of Expt:

Ex 5.1
Procedure:

Numerical Derivative Using Forward Difference

1. **Define the Value of a**:
   - Set $a = 1$, which is the point at which the numerical derivative of $f(x) = \tan^{-1}(x)$ will be calculated.
2. **Calculate the True Derivative**:
   - The derivative of $f(x) = \tan^{-1}(x)$ is $f'(x) = \frac{1}{1+x^2}$
   - Compute the true derivative at $x = 1$, using the formula: $trueval = \frac{1}{1+a^2}$
   - Store the result in trueval for comparison with the numerical approximation.
3. **Set the Step Size**:
   - Define $h = 1.0 \times 10^{-4}$, which is the step size used in the forward difference approximation.
4. **Apply the Forward Difference Formula**:
   - Use the forward difference formula to approximate the derivative of $f(x) = \tan^{-1}(x)$ at $x = 1$,

$$f'(x) \approx \frac{f(x+h) - f(x)}{h}$$

   - In this case, $f(x) = \tan^{-1}(x)$ at $x = 1$, so the formula becomes:

$$fwddiff \approx \frac{\tan^{-1}(a+h) - \tan^{-1}(a)}{h}$$

   - Store the result in fwddiff.
5. **Calculate the Error**:
   - Compute the absolute error between the true derivative (trueval) and the forward difference approximation (fwddiff):

$$errfwd = |trueval - fwddiff|$$

6. **Display the Results**:
   - Use the fprintf function to print the following:

- The true value of the derivative at $x = 1$,
- The forward difference approximation of the derivative,
- The absolute error between the true and approximated derivative.

Code:

```
% Calculate the numerical derivative of f(x) = atan(x) at x=1 using forward difference.

a=1;              % Set x=1
trueval=1/(1+a^2);  % True derivative at x=1

h=1.0e-4;          % Step size for approximation

% Forward difference formula for derivative
fwddiff = (atan(a+h)-atan(a))/h;

% Absolute error between true and approximated derivative
errfwd = abs(trueval-fwddiff);

% Display results
fprintf('True value of the derivative at x=1: %.12f\n', trueval);
fprintf('Forward difference approximation: %.12f\n', fwddiff);
fprintf('Absolute error: %.12f\n',errfwd );
```

O/P:

```
True value of the derivative at x=1: 0.500000000000
Forward difference approximation: 0.499975000834
Absolute error: 0.000024999166
```

Ex 5.2

Procedure:

Numerical Derivative Using Different Difference Methods

1. **Set the Value of a**:
   - Define $a = 1$, which is the point at which the numerical derivative of $f(x) = \tan^{-1}(x)$ will be calculated.
2. **Calculate the True Derivative**:
   - The derivative of $f(x) = \tan^{-1}(x)$ is $f'^{(x)} = \frac{1}{1+x^2}$
   - Compute the true derivative at $x = 1$ using the formula:

$$trueval = \frac{1}{1+a^2}$$

- Store the result in trueval for comparison with the numerical approximations.

3. **Set the Step Size**:
   - Define $h = 1.0 \times 10^{-4}$, which is the small increment used in the difference methods to approximate the derivative.
4. **Apply the Forward Difference Method**:
   - Use the forward difference formula to approximate the derivative:

$$f'(x) \approx \frac{f(x+h) - f(x)}{h}$$

   - In this case, $f(x) = \tan^{-1}(x)$, so the forward difference becomes:

$$fwddiff \approx \frac{\tan^{-1}(a+h) - \tan^{-1}(a)}{h}$$

   - Store the result in fwddiff, and compute the error:

$$errfwd = |trueval - fwddiff|$$

5. **Apply the Central Difference Method**:
   - Use the Central difference formula to approximate the derivative:

$$f'(x) \approx \frac{f(x+h) - f(x-h)}{2h}$$

   - In this case, $f(x) = \tan^{-1}(x)$, so the central difference becomes:

$$centdiff \approx \frac{\tan^{-1}(a+h) - \tan^{-1}(a-h)}{2h}$$

   - Store the result in centdiff, and compute the error:

$$errcent = |trueval - centdiff|$$

6. **Apply the Backward Difference Method**:
   - Use the backward difference formula to approximate the derivative:

$$f'(x) \approx \frac{f(x) - f(x-h)}{h}$$

   - In this case, $f(x) = \tan^{-1}(x)$, so the backward difference becomes:

$$backdiff \approx \frac{\tan^{-1}(a) - \tan^{-1}(a-h)}{h}$$

- Store the result in backdiff, and compute the error:

$$errfwd = |trueval - backdiff|$$

7. **Display the Results**:
   - Use disp to print:
     - The true value of the derivative at $x = 1$,
     - The forward difference approximation and its error,
     - The central difference approximation and its error,
     - The backward difference approximation and its error.

Code:

```
% Calculate the numerical derivative of f(x) = atan(x) at x=1 using different difference methods.
a = 1;              % Set x=1
trueval = 1/(1+a^2); % True derivative at x=1
h = 1.0e-4;         % Step size for approximation

% Forward difference formula for derivative
fwddiff = (atan(a+h) - atan(a))/h;
errfwd = abs(trueval - fwddiff);  % Error for forward difference

% Central difference formula for derivative
centdiff = (atan(a+h) - atan(a-h))/(2*h);
errcent = abs(trueval - centdiff);  % Error for central difference

% Backward difference formula for derivative
backdiff = (atan(a) - atan(a-h))/h;
errback = abs(trueval - backdiff);  % Error for backward difference

% Display results
disp(['True value of the derivative at x=1: ', num2str(trueval)]);
disp('---');
disp(['Forward difference approximation: ', num2str(fwddiff)]);
disp(['Error (Forward): ', num2str(errfwd)]);
disp('---');
disp(['Central difference approximation: ', num2str(centdiff)]);
disp(['Error (Central): ', num2str(errcent)]);
disp('---');
disp(['Backward difference approximation: ', num2str(backdiff)]);
```

```
disp(['Error (Backward): ', num2str(errback)]);
```

O/P:

```
True value of the derivative at x=1: 0.5
---
Forward difference approximation: 0.49998
Error (Forward): 2.4999e-05
---
Central difference approximation: 0.5
Error (Central): 8.3317e-10
---
Backward difference approximation: 0.50003
Error (Backward): 2.5001e-05
```

Ex 5.3

Procedure:

Numerical Differentiation with Various Methods

1. **Define Step Sizes**:
   - Create a vector h that holds step sizes from $10^{-1}\ to\ 10^{-8}$, using powers of 10:

     $$h\ =\ 10.\hat{}[-1:-2:-8];$$
   - These step sizes will be used for the numerical differentiation methods.
2. **Define the Point for Derivative Calculation**:
   - Set $a = 2$, which is the point where the derivative of the function $f(x)\ =\ x^x$ will be calculated.
3. **Symbolic Differentiation**:
   - Use symbolic differentiation to compute the true derivative of the function $f(x)\ =\ x^x$:
     1. Declare x as a symbolic variable using syms x.
     2. Differentiate AssignmentDiffFun1(x) symbolically with respect to x.
     3. Substitute the value $a = 2$ into the symbolic derivative and convert the result to a double precision number for comparison:

        $$trueval1\_at\_a\ =\ double(subs(trueval1, x, a))$$

4. **Manual Calculation of the Derivative**:
    - Manually calculate the true derivative of the function $f(x) = x^x \; at \; x = 2$ using the formula:

$$f'(x) = x^x \times (1 + log(x))$$

    - Store the manually calculated result in trueval2.
5. **Forward Difference Approximation**:
    - Use the forward difference method to approximate the derivative:
    - Compute the forward difference for each step size in h and calculate the error
6. **Central Difference Approximation**:
    - Use the central difference method to approximate the derivative:
    - Compute the central difference for each step size in h and calculate the error
7. **Backward Difference Approximation**:
    - Use the backward difference method to approximate the derivative:
    - Compute the backward difference for each step size in h and calculate the error
8. **Display the Results**:
    - Print the true derivative values:
        - The symbolic derivative calculated using MATLAB's built-in functions (trueval1_at_a).
        - The manually calculated derivative (trueval2).
    - Print the derivatives obtained from the forward, central, and backward difference methods.
9. **Plot the Errors**:
    - Create a log-log plot to visualize the errors for each method:
        - Plot the forward difference errors in red.
        - Plot the central difference errors in green.
        - Plot the backward difference errors in blue.
    - Add labels, title, and a legend to the plot for clarity:
        - **x-axis**: Step Size (h),
        - **y-axis**: Error,
        - **Title**: Error in Differentiation Methods.
    - Enable grid for better visualization and turn off hold to release the plot.

Function Definition:

- **AssignmentDiffFun1**:
  - This function defines the function $f(x) = x^x$, which is being differentiated:

```
function fval = AssignmentDiffFun1(x)
    fval = (x.^x);
end
```

Code:

```
% Define a vector of step sizes (h) using powers of 10 from -1 to -8
h = 10.^[-1:-2:-8];
% Define the x value at which the derivative is to be estimated
a = 2;

syms x;  % Declare x as a symbolic variable

trueval1 = diff(AssignmentDiffFun1(x), x);  % Differentiate symbolically
trueval1_at_a = double(subs(trueval1, x, a))  % Substitute 'a' for 'x' and convert to double
trueval2 = (a^a)*(1+log(a))

% Calculate forward difference approximation of derivative and associated error
fwddiff = (AssignmentDiffFun1(a+h) - AssignmentDiffFun1(a))./h;
errfwd = abs(trueval2 - fwddiff);

% Calculate central difference approximation of derivative and associated error
centdiff = (AssignmentDiffFun1(a+h) - AssignmentDiffFun1(a-h))./(2*h);
errcent = abs(trueval2 - centdiff);

% Calculate backward difference approximation of derivative and associated error
backdiff = (AssignmentDiffFun1(a) - AssignmentDiffFun1(a-h))./h;
errback = abs(trueval2 - backdiff);

% Display actual derivative values calculated through symbolic differentiation and manual calculation
fprintf('Actual Integral (By In-built Integral Formula in MATLAB: %.12f\n', trueval1_at_a);
fprintf('Actual Derivative (By manually solving): %.12f\n', trueval2);
fprintf('\n');
fprintf('Derivative obtained by Forward Difference Method: %.12f\n:', fwddiff);
fprintf('\n');
```

```
fprintf('Derivative obtained by Central Difference Method: %.12f\n:', centdiff);
fprintf('\n');
fprintf('Derivative obtained by Backward Difference Method: %.12f\n:',
backdiff);

% Plot errors for each differentiation method on a log-log scale
loglog(h,errfwd,'r'); % Forward difference errors in red
hold on; % Retain current plot when adding more lines
loglog(h,errcent,'g'); % Central difference errors in green
loglog(h,errback,'b'); % Backward difference errors in blue

% Add labels, title, and legend to the plot
xlabel('Step Size (h)');
ylabel('Error');
title('Error in Differentiation Methods');
legend('Forward Difference', 'Central Difference', 'Backward Difference');

% Enable grid on plot for better visibility of data points
grid on;
% Release the current plot
hold off;
```

Function File:

```
function fval = AssignmentDiffFun1(x)
   fval = (x.^x);
end
```

O/P:

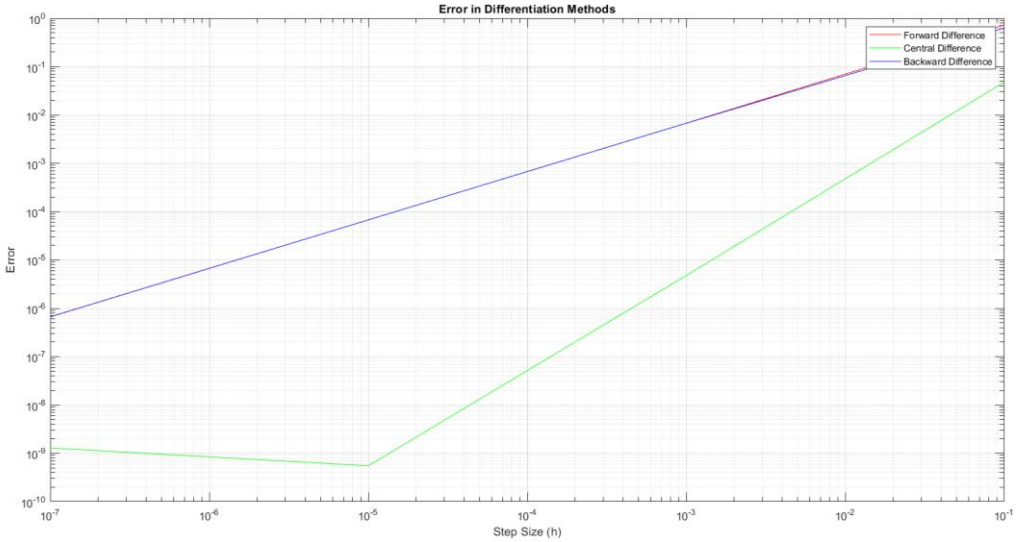
Actual Derivative (By In-built Differentiation Formula in MATLAB:
6.772588722240
Actual Derivative (By manually solving): 6.772588722240

:Derivative obtained by Forward Difference Method: 7.496380917422
:Derivative obtained by Forward Difference Method: 6.779326982042
:Derivative obtained by Forward Difference Method: 6.772656057752
:Derivative obtained by Forward Difference Method: 6.772589387083

:Derivative obtained by Central Difference Method: 6.820338739119
:Derivative obtained by Central Difference Method: 6.772593484604
:Derivative obtained by Central Difference Method: 6.772588722792
:Derivative obtained by Central Difference Method: 6.772588720949

:Derivative obtained by Backward Difference Method: 6.144296560815

:Derivative obtained by Backward Difference Method: 6.765859987167
:Derivative obtained by Backward Difference Method: 6.772521387832
:Derivative obtained by Backward Difference Method: 6.772588054815



Ex 5.4
Code:

```
% Define a vector of step sizes (h) using powers of 10 from -1 to -8
h = 10.^[-6:-1:-10];
% Define the x value at which the derivative is to be estimated
a = 2;

syms x;  % Declare x as a symbolic variable

trueval1 = diff(AssignmentDiffFun2(x), x);  % Differentiate symbolically
trueval1_at_a = double(subs(trueval1, x, a));  % Substitute 'a' for 'x' and convert to double
trueval2 = (a^sin(a))*((sin(a)/a) + log(a)*cos(a))  + ((sin(a))^a)*(a*cot(a) + log(sin(a)));

% Calculate forward difference approximation of derivative and associated error
fwddiff = (AssignmentDiffFun2(a+h) - AssignmentDiffFun2(a))./h;
errfwd = abs(trueval2 - fwddiff);

% Calculate central difference approximation of derivative and associated error
centdiff = (AssignmentDiffFun2(a+h) - AssignmentDiffFun2(a-h))./(2*h);
errcent = abs(trueval2 - centdiff);
```

```
% Calculate backward difference approximation of derivative and associated
error
backdiff = (AssignmentDiffFun2(a) - AssignmentDiffFun2(a-h))./h;
errback = abs(trueval2 - backdiff);

% Display actual derivative values calculated through symbolic differentiation
and manual calculation
fprintf('Actual Derivative (By In-built Integral Formula in MATLAB: %.12f\n',
trueval1_at_a);
fprintf('Actual Derivative (By manually solving): %.12f\n', trueval2);
fprintf('\n');
fprintf('Derivative obtained by Forward Difference Method: %.12f\n:', fwddiff);
fprintf('\n');
fprintf('Derivative obtained by Central Difference Method: %.12f\n:', centdiff);
fprintf('\n');
fprintf('Derivative obtained by Backward Difference Method: %.12f\n:',
backdiff);

% Plot errors for each differentiation method on a log-log scale
loglog(h,errfwd,'r'); % Forward difference errors in red
hold on; % Retain current plot when adding more lines
loglog(h,errcent,'g'); % Central difference errors in green
loglog(h,errback,'b'); % Backward difference errors in blue

% Add labels, title, and legend to the plot
xlabel('Step Size (h)');
ylabel('Error');
title('Error in Differentiation Methods');
legend('Forward Difference', 'Central Difference', 'Backward Difference');

% Enable grid on plot for better visibility of data points
grid on;
% Release the current plot
hold off;
```

Function File:

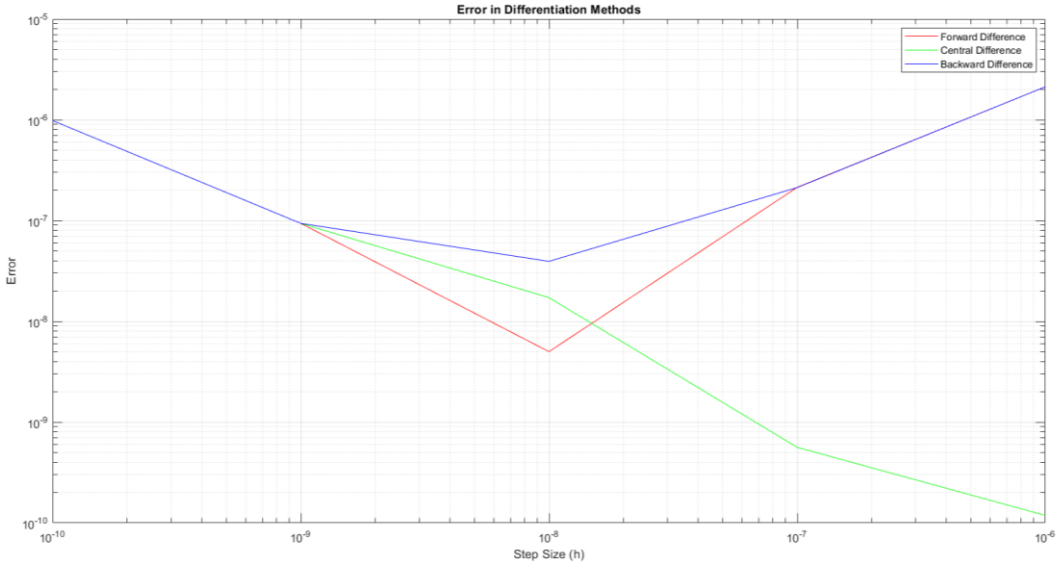
```
function fval = AssignmentDiffFun2(x)
    fval = (x.^sin(x)) + ((sin(x)).^x);
end
```

O/P:

Actual Derivative (By In-built Differentiation Formula in MATLAB: -0.523278215751

Actual Derivative (By manually solving): -0.523278215751

:Derivative obtained by Forward Difference Method: -0.523280342613
:Derivative obtained by Forward Difference Method: -0.523278429476
:Derivative obtained by Forward Difference Method: -0.523278220754
:Derivative obtained by Forward Difference Method: -0.523278309572
:Derivative obtained by Forward Difference Method: -0.523279197751
:
:Derivative obtained by Central Difference Method: -0.523278215869
:Derivative obtained by Central Difference Method: -0.523278216313
:Derivative obtained by Central Difference Method: -0.523278198550
:Derivative obtained by Central Difference Method: -0.523278309572
:Derivative obtained by Central Difference Method: -0.523279197751
:
:Derivative obtained by Backward Difference Method: -0.523276089126
:Derivative obtained by Backward Difference Method: -0.523278003151
:Derivative obtained by Backward Difference Method: -0.523278176345
:Derivative obtained by Backward Difference Method: -0.523278309572
:Derivative obtained by Backward Difference Method: -0.523279197751



Ex 5.5
Code:

```
% Define a vector of step sizes (h) using powers of 10 from -1 to -8
h = 10.^[-4:-1:-9];
% Define the x value at which the derivative is to be estimated
```

```
a = 2;

syms x;  % Declare x as a symbolic variable

trueval1 = diff(AssignmentDiffFun3(x), x);  % Differentiate symbolically
trueval1_at_a = double(subs(trueval1, x, a));  % Substitute 'a' for 'x' and convert
to double
trueval2 = (a+1/a)^a*(((a^2 - 1)/(a^2 + 1)) + log(a+1/a)) +
a^(1+1/a)*(((1/a)+(1/(a^2)) - (log(a)/(a^2))));

% Calculate forward difference approximation of derivative and associated error
fwddiff = (AssignmentDiffFun3(a+h) - AssignmentDiffFun3(a))./h;
errfwd = abs(trueval2 - fwddiff);

% Calculate central difference approximation of derivative and associated error
centdiff = (AssignmentDiffFun3(a+h) - AssignmentDiffFun3(a-h))./(2*h);
errcent = abs(trueval2 - centdiff);

% Calculate backward difference approximation of derivative and associated
error
backdiff = (AssignmentDiffFun3(a) - AssignmentDiffFun3(a-h))./h;
errback = abs(trueval2 - backdiff);

% Display actual derivative values calculated through symbolic differentiation
and manual calculation
fprintf('Actual Derivative (By In-built Differentiation Formula in MATLAB:
%.12f\n', trueval1_at_a);
fprintf('Actual Derivative (By manually solving): %.12f\n', trueval2);
fprintf('\n');
fprintf('Derivative obtained by Forward Difference Method: %.12f\n:', fwddiff);
fprintf('\n');
fprintf('Derivative obtained by Central Difference Method: %.12f\n:', centdiff);
fprintf('\n');
fprintf('Derivative obtained by Backward Difference Method: %.12f\n:',
backdiff);

% Plot errors for each differentiation method on a log-log scale
loglog(h,errfwd,'r'); % Forward difference errors in red
hold on; % Retain current plot when adding more lines
loglog(h,errcent,'g'); % Central difference errors in green
loglog(h,errback,'b'); % Backward difference errors in blue

% Add labels, title, and legend to the plot
```

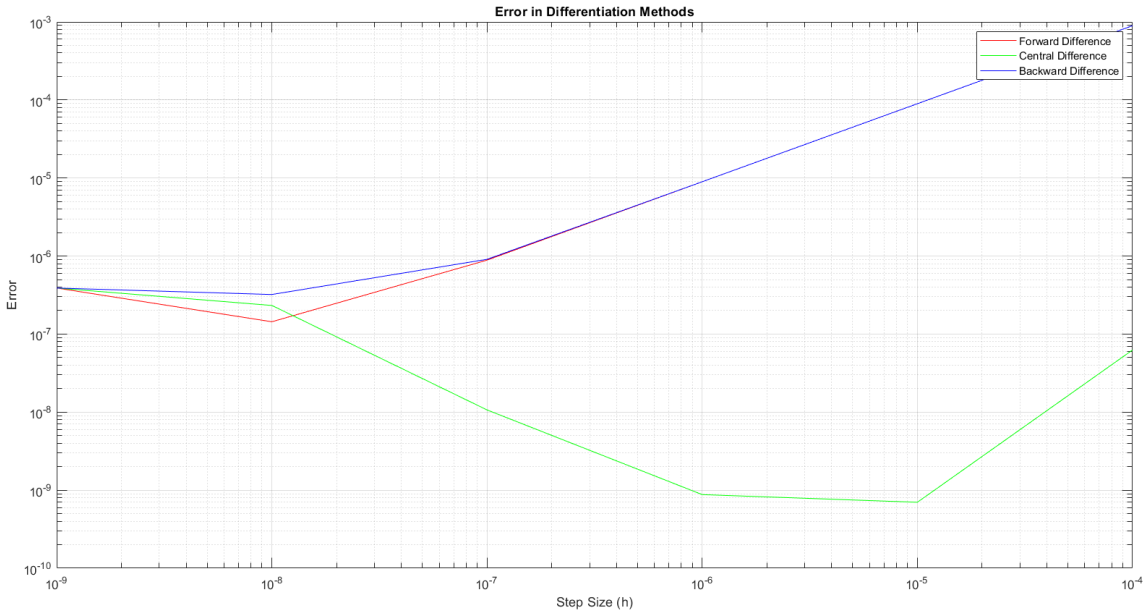
```
xlabel('Step Size (h)');
ylabel('Error');
title('Error in Differentiation Methods');
legend('Forward Difference', 'Central Difference', 'Backward Difference');

% Enable grid on plot for better visibility of data points
grid on;
% Release the current plot
hold off;
```

Function File:

```
function fval = AssignmentDiffFun3(x)
    fval = ((x + 1./x).^x) + x.^(1+1./x);
end
```

O/P:

```
Actual Derivative (By In-built Differentiation Formula in MATLAB:
11.108008346039
Actual Derivative (By manually solving): 11.108008346039

:Derivative obtained by Forward Difference Method: 11.108903793708
:Derivative obtained by Forward Difference Method: 11.108097885248
:Derivative obtained by Forward Difference Method: 11.108017302419
:Derivative obtained by Forward Difference Method: 11.108009232430
:Derivative obtained by Forward Difference Method: 11.108008202143
:Derivative obtained by Forward Difference Method: 11.108008735050
:
:Derivative obtained by Central Difference Method: 11.108008408600
:Derivative obtained by Central Difference Method: 11.108008346739
:Derivative obtained by Central Difference Method: 11.108008346916
:Derivative obtained by Central Difference Method: 11.108008335370
:Derivative obtained by Central Difference Method: 11.108008113325
:Derivative obtained by Central Difference Method: 11.108008735050
:
:Derivative obtained by Backward Difference Method: 11.107113023492
:Derivative obtained by Backward Difference Method: 11.107918808229
:Derivative obtained by Backward Difference Method: 11.107999391413
:Derivative obtained by Backward Difference Method: 11.108007438310
:Derivative obtained by Backward Difference Method: 11.108008024507
:Derivative obtained by Backward Difference Method: 11.108008735050
```


Error in Differentiation Methods
Forward Difference
Central Difference
Backward Difference
Error
Step Size (h)
10^-3
10^-4
10^-5
10^-6
10^-7
10^-8
10^-9
10^-10
10^-9
10^-8
10^-7
10^-6
10^-5
10^-4

Expt No. 6	NUMERICAL INTEGRATION	Date of Expt:

Ex 6.1
Procedure:

Numerical Integration Using the Trapezoidal Rule

1. **Define the Limits of Integration**:
   - Set a = π/4 and b = π/2, which represent the lower and upper limits for the integral.
2. **Calculate the True Value of the Integral (Method 1)**:
   - Use MATLAB's built-in integral function to calculate the true value of the integral numerically:
   - trueval1 = integral(@AssignmentIntFun1, a, b)
3. **Calculate the True Value of the Integral (Method 2)**:
   - Manually solve the integral using the given formula:

   trueval2 = (2 * log|sin²(b) - 4sin(b) + 5| + 7atan(sin(b) - 2)) - (2 * log|sin²(a) - 4sin(a) + 5| + 7atan(sin(a) - 2))
4. **Define an Array of n Values**:
   - Set up an array of values for n (the number of sub intervals for the trapezoidal rule): n_values = [2, 5, 20, 40, 80, 160]
   - This array will be used to compute the integral with different resolutions (i.e., with increasing numbers of sub intervals).
5. **Initialize an Array to Store Errors**:
   - Create an array errors initialized to zero, which will store the error values for each n:

     errors = zeros(size(n_values))
6. **Loop Over Different nnn Values**:
   - Start a loop that iterates over each value in n_values:
     1. **Set n**: For each iteration, set n to the current number of sub intervals from n_values.
     2. **Calculate Step Size**: Compute the step size h for the trapezoidal rule: h = (b - a) / n
     3. **Create xxx Vector**: Create a vector xvec of n+1n+1n+1 points from a to b spaced by h.

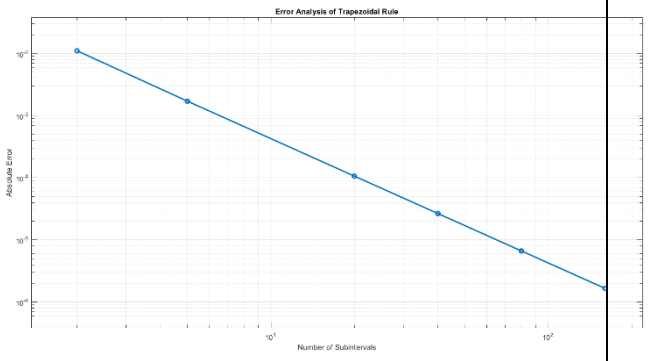
4. **Evaluate Function**: Compute the function values at each point in xvec using AssignmentIntFun1(xvec) and store them in fvec.
5. **Apply the Trapezoidal Rule**:
    - Initialize an array interval to store the contribution of each interval.
    - Use the trapezoidal rule formula to compute the integral for each sub interval:
      linterval(i) = (h/2) * (fvec(i) + fvec(i+1))
    - Sum up the contributions of all intervals to get the total integral approximation ItrapI_{\text{trap}}Itrap.
6. **Compute the Error**:
    - Calculate the absolute error for the current n by comparing the numerical integral
      errors(idx) = |trueval1 - I_trap|

7. **Plot the Error vs. n in a Log-Log Plot**:
    - Create a log-log plot to visualize how the error decreases as n increases:
        - **x-axis**: Number of sub-intervals n,
        - **y-axis**: Absolute error.
    - Plot the error for each value of n and add labels, title, and grid to the plot for clarity.
8. **Display Results**:
    - Use the fprintf function to display:
        - The true value of the integral calculated using MATLAB's built-in integral function (trueval1).
        - The manually calculated integral value (trueval2).
        - The computed integral using the trapezoidal rule (I_trap).
        - The array of errors for each n.

Code:

```
a = pi/4;
b = pi/2;

trueval1 = integral(@AssignmentIntFun1, a, b);

trueval2 = (2*log(abs(sin(b).^2 - 4*sin(b) + 5)) + 7*atan(sin(b) - 2)) -
(2*log(abs(sin(a).^2 - 4*sin(a) + 5)) + 7*atan(sin(a) - 2));

n_values = [2, 5, 20, 40, 80, 160]; % Array of n values
```

```
errors = zeros(size(n_values)); % Array to store error for each n value

% Loop over different n values
for idx = 1:length(n_values)
    n = n_values(idx);

    % Trapezoidal rule calculation
    h = (b-a)/n;
    xvec = a:h:b;
    fvec = AssignmentIntFun1(xvec);
    linterval = zeros(n,1);

    for i = 1:n
        linterval(i) = h/2 * (fvec(i) + fvec(i+1));
        disp(linterval(i))
    end

    I_trap = sum(linterval);

    % Compute and store error for current n
value
    errors(idx) = abs(trueval1 - I_trap);
end

% Plot error versus n in log-log plot
loglog(n_values, errors, 'o-', 'LineWidth', 2)
xlabel('Number of Sub intervals')
ylabel('Absolute Error')
title('Error Analysis of Trapezoidal Rule')
grid on

fprintf('Actual Integral (By In-built Integral Formula in MATLAB): %.12f\n',
trueval1)
fprintf('Actual Integral (By manually solving): %.12f\n', trueval2)
fprintf('Computed Integral: %.12f\n', I_trap)
disp('Absolute Error:')
disp(errors)
```

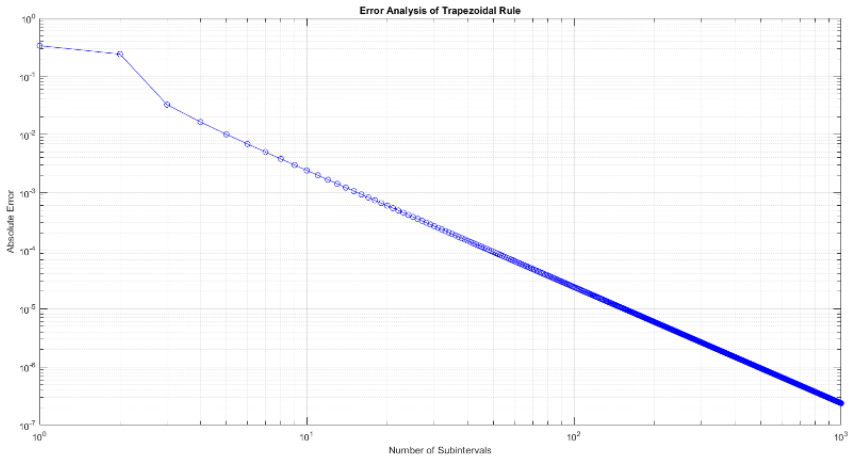


Function File:

```
function fval = AssignmentIntFun1(x)
    fval = (2.*sin(2*x) - cos(x))./(6 - cos(x).^2 - 4.*sin(x));
end
```

O/P:

Actual Integral (By In-built Integral Formula in MATLAB): 0.310320165564
Actual Integral (By manually solving): 0.310320165564
Computed Integral: 0.310315957052
Absolute Error:
0.0276 0.0043 0.0003 0.0001 0.0000 0.0000



Ex 6.2
Procedure:

Numerical Integration Using Simpson's 1/3 Rule

1. **Define the Limits of Integration**:
   - Set the limits of integration a=15a = 15a=15 and b=28b = 28b=28.
2. **Calculate the True Value of the Integral (Method 1)**:
   - Use MATLAB's built-in integral function to compute the exact value of the integral:
     trueval1 = integral(@AssignmentIntFun3, a, b);
3. **Calculate the True Value of the Integral (Method 2)**:
   - Manually solve the integral using the provided formula:
     trueval2 = 1/4*(sin(12*b)/12 + sin(8*b)/8 + b + sin(4*b)/4) - 1/4*(sin(12*a)/12 + sin(8*a)/8 + a + sin(4*a)/4)
4. **Set the Number of Sub intervals n**:
   - Choose n=50n = 50n=50 for the Simpson's 1/3 rule.
   - Calculate the step size h:
     h = (b-a)/n;
5. **Create x Vector and Evaluate Function**:

- Create a vector xvec that holds the points from a to b spaced by h, and evaluate the function AssignmentIntFun3(x) at those points to get fvec:
  xvec = a:h:b;
  fvec = AssignmentIntFun3(xvec);

6. **Simpson's 1/3 Rule Calculation**:
   - Use a for-loop to apply Simpson's 1/3 rule over every pair of intervals:
     - For Simpson's rule, the formula for the composite Simpson's rule is:

$$I_{simp} = \frac{h}{3}(f(x_0) + 4f(x_1) + 2f(x_2) + 4f(x_3) + \cdots . + f(b))$$

     - This is implemented by iterating over the odd-indexed points and applying Simpson's 1/3 rule:

```
linterval = zeros(n,1);
for i = 1:2:n-1
   linterval(i) = h/3 * (fvec(i) + 4*fvec(i+1) + fvec(i+2));
end
I_simp3 = sum(linterval);
```

7. **Calculate the Error**:
   - Compute the absolute error between the true value (trueval1) and the computed integral (I_simp3):
     err1 = abs(trueval1 - I_simp3);
8. **Display Results**:
   - Use fprintf to print the results:
     - The true value of the integral (computed using MATLAB's integral function).
     - The manually calculated value using the given formula.
     - The computed integral using Simpson's 1/3 rule.
     - The absolute error between the true and computed values.

Code:

```
a = 15;
b = 28;
trueval1 = integral(@AssignmentIntFun3, a, b);
```

```
trueval2 = 1/4*(sin(12*b)/12 + sin(8*b)/8 + b + sin(4*b)/4) - 1/4*(sin(12*a)/12
+ sin(8*a)/8 + a + sin(4*a)/4);
n = 50;

%% Simpson's 1/3rd rule
h = (b-a)/n;
xvec = a:h:b;
fvec = AssignmentIntFun3(xvec);
linterval = zeros(n,1);

%% Simpson's 1/3rd rule
for i = 1:2:n-1
    linterval(i) = h/3 * (fvec(i) + 4*fvec(i+1) + fvec(i+2));
end

I_simp3 = sum(linterval);
err1 = abs(trueval1 - I_simp3); % Error between true value and computed
integral

% Displaying results
fprintf('Actual Integral (By In-built Integral Formula in MATLAB): %.12f\n',
trueval1)
fprintf('Actual Integral (By manually solving): %.12f\n', trueval2)
fprintf('Computed Integral: %.12f\n', I_simp3)
disp('Absolute Error:')
disp(err1)
```

Function File:

```
function fval = AssignmentIntFun3(x)
   fval = cos(2.*x).*cos(4.*x).*cos(6.*x);
end
```

O/P:

```
Actual Integral (By In-built Integral Formula in MATLAB): 3.189731183736
Actual Integral (By manually solving): 3.189731183736
Computed Integral: 4.115455602037
Absolute Error: 0.9257
```

Ex 6.3:

Code:

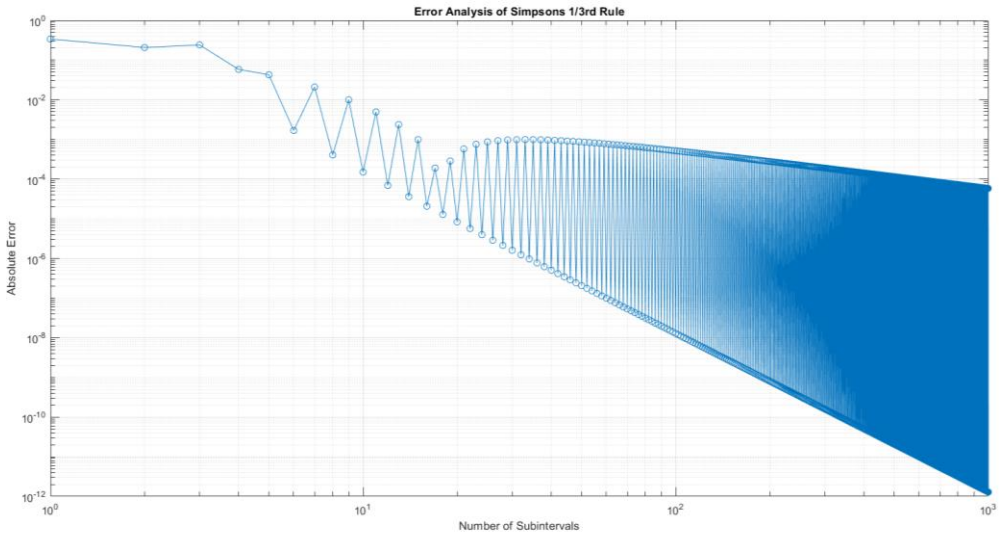
```
% Error Analyis of Integration by Simpsons 1/3rd Rule
a = 15;
b = 16;
truval1 = integral(@AssignmentIntFun3, a, b);
truval2 = 1/4*(sin(12*b)/12 + sin(8*b)/8 + b + sin(4*b)/4) - 1/4*(sin(12*a)/12 +
sin(8*a)/8 + a + sin(4*a)/4);
n_values = [1:1000];
errors = zeros(size(n_values));

for idx = 1:length(n_values)
    n = n_values(idx);
    h = (b-a)/n;
    xvec = a:h:b;
    fvec = AssignmentIntFun3(xvec);
    linterval = zeros(n,1);

% Simpsons 1/3rd Rule
% Thats why in for loop we divide by 2 steps at each iterations
    for i = 1:2:n-1
      linterval(i) = h/3*(fvec(i) + 4*fvec(i+1) + fvec(i+2));
    end

I_simp3 = sum(linterval);
err1(idx) = abs(truval1 - I_simp3);
end

% Plot error versus n in lo-log plot
loglog(n_values, err1, 'o-')
xlabel('Number of Sub intervals')
ylabel('Absolute Error')
title('Error Analysis of Simpsons 1/3rd Rule')
grid on

fprintf('Actual Integral (By In-built Integral Formula in MATLAB): %.12f\n',
truval1);
fprintf('Actual Integral (By Manual solving): %.12f\n', truval2);
fprintf('Computed Integral: %.12f\n', I_simp3)
disp('Absolute Error:')
disp(err1)
```

Function File:

```
function fval = AssignmentIntFun3(x)
   fval = cos(2.*x).*cos(4.*x).*cos(6.*x);
end
```

O/P:

Actual Integral (By In-built Integral Formula in MATLAB): 0.340236750946
Actual Integral (By Manual solving): 0.340236750946
Computed Integral: 0.340236750947
Absolute Error - Maximum: 0.3402



Ex 6.4:

Procedure:

Here we have used Simpson's 3/8th Rule:

$$I_{3/8} = \frac{3h}{8}(f(a) + 3f(x_1) + 3f(x_2). + f(b))$$

Code:

```
a = 15;
b = 20;
trueval1 = integral(@AssignmentIntFun5, a, b);
trueval2 = (b^2/2*asin(b) + 1/2*(b/2*sqrt(1-b^2) + 1/2*asin(b) - asin(b))) -
(a^2/2*asin(a) + 1/2*(b/2*sqrt(1-a^2) + 1/2*asin(a) - asin(a)));
n = 150;

%% Simpson's 3/8th rule
h = (b-a)/n;
xvec = a:h:b;
fvec = AssignmentIntFun5(xvec);
```

```
linterval = zeros(n,1);

%% Simpson's 3/8th rule
for i = 1:3:n-2
   if i+3 <= length(fvec)
      linterval(i) = 3*h/8 * (fvec(i) + 3*fvec(i+1) + 3*fvec(i+2) + fvec(i+3));
   end
end


I_simp8 = sum(linterval);
err1 = abs(trueval1 - I_simp8); % Error between true value and computed
integral

% Displaying results
fprintf('Actual Integral (By In-built Integral Formula in MATLAB): %.12f\n',
trueval1)
fprintf('Actual Integral (By manually solving): %.12f\n', trueval2)
fprintf('Computed Integral: %.12f\n', I_simp8)
disp('Absolute Error:')
disp(err1)
```

Function File:

```
function fval = AssignmentIntFun5(x)
  fval = x.*asin(x);
end
```

O/P:

```
Actual Integral (By In-built Integral Formula in MATLAB): 137.444678594553
Actual Integral (By manually solving): 137.444678594553
Computed Integral: 137.444678594553
Absolute Error:
  2.9729e-11
```

Ex 6.5:
Procedure:

```
% Error Analysis of Integration by Simpsons 3/8th Rule
a = 15;
b = 16;
truval1 = integral(@AssignmentIntFun3, a, b);
```

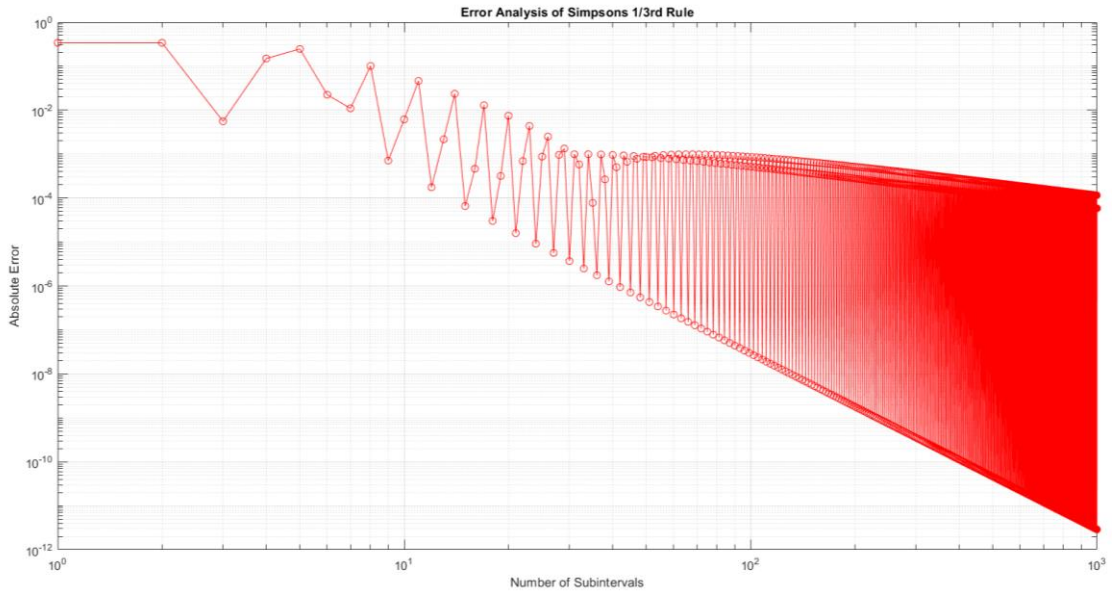
```
truval2 = 1/4*(sin(12*b)/12 + sin(8*b)/8 + b + sin(4*b)/4) - 1/4*(sin(12*a)/12 +
sin(8*a)/8 + a + sin(4*a)/4);
n_values = [1:1000];
errors = zeros(size(n_values));

for idx = 1:length(n_values)
    n = n_values(idx);
    h = (b-a)/n;
    xvec = a:h:b;
    fvec = AssignmentIntFun3(xvec);
    linterval = zeros(n,1);

% Simpsons 3/8th Rule
% That's why, in 'for' loop we divide by 3 steps at each iterations
for i = 1:3:n-2
    if i+3 <= length(fvec)
        linterval(i) = 3*h/8 * (fvec(i) + 3*fvec(i+1) + 3*fvec(i+2) + fvec(i+3));
    end
end

I_simp3 = sum(linterval);
err1(idx) = abs(truval1 - I_simp3);
end

% Plot error versus n in lo-log plot
loglog(n_values, err1, 'o-r')
xlabel('Number of Sub intervals')
ylabel('Absolute Error')
title('Error Analysis of Simpsons 1/3rd Rule')
grid on

fprintf('Actual Integral (By In-built Integral Formula in MATLAB): %.12f\n',
truval1);
fprintf('Actual Integral (By Manual solving): %.12f\n', truval2);
fprintf('Computed Integral: %.12f\n', I_simp3)
disp('Absolute Error - Maximum:')
disp(max(err1))
```

Function File:

```
function fval = AssignmentIntFun3(x)
   fval = cos(2.*x).*cos(4.*x).*cos(6.*x);
end
```

O/P:

Actual Integral (By In-built Integral Formula in MATLAB): 0.340236750946
Actual Integral (By Manual solving): 0.340236750946
Computed Integral: 0.340242642660
Absolute Error - Maximum: 0.3402



Expt No. 7	ORDINARY DIFFERENTIAL EQUATIONS	Date of Expt:

Ex 7.1:

Procedure:

Using Euler's Explicit Method

We are solving the following ODE using **Euler's Explicit Method**:

$$\frac{dy}{dx} = -5t^2y^3$$

with the initial condition $y(0) = 1$, over the interval [0,10] with step size h=0.1

1. Define the Problem

- Initial condition: $y(0) = 1$
- Time interval: t∈[0,10]
- Differential equation: $y' - 5t^2y^3$

2. Set Parameters

- $t_0 = 0$: initial time
- $y_o = 1$: initial value of y
- $t_{end} = 10$: end time
- Step size h=0.1
- Number of time steps $N = \left(\frac{t_{end} - t_0}{h}\right)$

3. Initialize Variables

- Define the time vector T from t_0 to t_{end} with step size h.
- Create an array Y to store the solution of $y(t)$. Initialize $Y(0) = y_0$

4. Euler's Explicit Method Formula

For each time step i, compute the next value of $Y(i+1)$ using the formula:

$$Y(i+1) = Y(i) + h.f(T(i), Y(i))$$

where $f(t, y) = -5t^2y^3$ is the right-hand side of the ODE.

5. Implementing the Method

Loop through all time steps from i= 1 to N, calculating $Y(i+1)$ using the explicit Euler formula.

6. Plot the Results

Once the solution is obtained, plot the values of $y(t)$ over the interval [0,10].

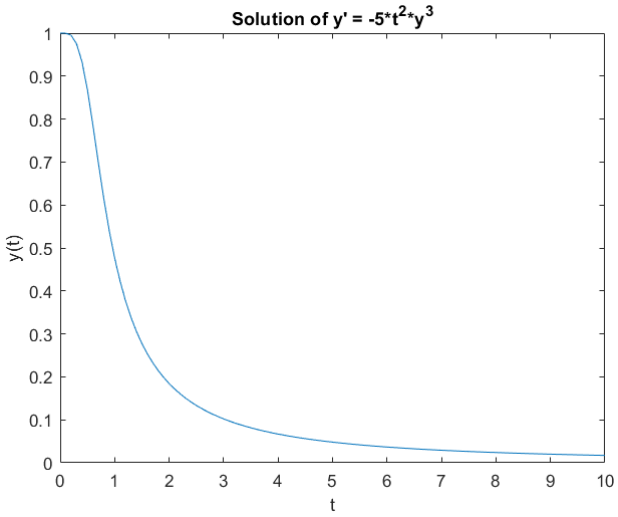
7. Calculate the True Solution and Error

Use ode45, MATLAB's built-in ODE solver, to compute the true solution for comparison. Calculate the error as the absolute difference between the Euler solution and the true solution.

Code:

EULER'S EXPLICIT METHOD

```
% Solve ODE - IVP using Euler's Explicit method
% y' = -5*t^2*y^3
% y(0) = 1

t0 = 0;
y0 = 1;
tEnd = 10;
h = 0.1 ;
N = (tEnd - t0)/h;

%% Initializing Solutions
T = [t0:h:tEnd]';
Y = zeros(N+1, 1);
Y(1) = y0;

%% Solving using Euler's Explicit Method
for i = 1:N
    fi = myFunEx1(T(i),Y(i));
    Y(i+1) = Y(i) + h*fi;
end

%% Plot Results
plot(T, Y);
title('Solution of y'' = -5*t^2*y^3');
xlabel('t');
ylabel('y(t)');

%% Obtain errors
[t, Ytrue] = ode45(@myFunEx1, T, y0); % element by element squaring
ERR = abs(Ytrue - Y);
maxError = max(ERR)
```



Function File:

```
function dy = myFunEx1(x,y)
  dy = -5*x^2*y^3;
```

```
end
```

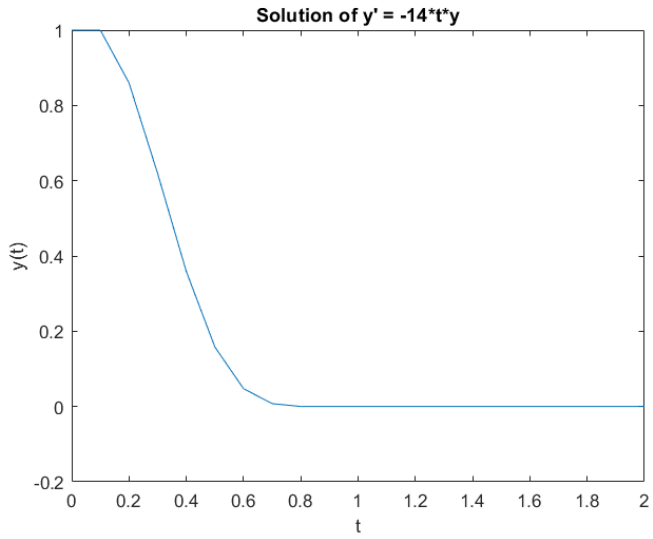
O/P:

```
maxError1 =    0.0282
```

Ex 7.2:

Code:

```
% Solve ODE - IVP using Euler's Explicit method
% y' = -14ty
% y(0) = 1

t0 = 0;
y0 = 1;
tEnd = 2;
h = 0.1 ;
N = (tEnd - t0)/h;

%% Initializing Solutions
T = [t0:h:tEnd]';
Y = zeros(N+1, 1);
Y(1) = y0;

%% Solving using Euler's Explicit
Method
for i = 1:N
    fi = -14*T(i)*Y(i);
    Y(i+1) = Y(i) + h*fi;
end

%% Plot Results and obtain errors
plot(T,Y);
title('Solution of y" = -14*t*y');
xlabel('t');
ylabel('y(t)');

Ytrue = exp(-7.*(T.^2));
ERR = abs(Ytrue - Y)
maxError = max(ERR)
```

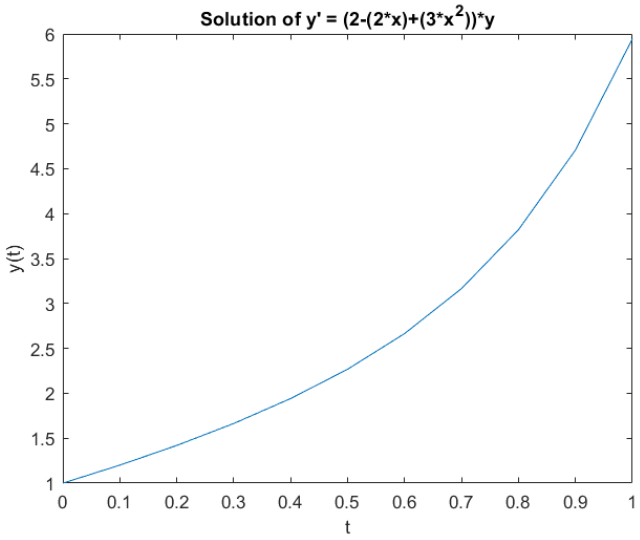


O/P:

```
maxError = 0.1042
```

Ex 7.3:
Code:

```
% Solve ODE - IVP using Euler's Explicit method
% y' = (2-(2*x)+(3*x^2))*y
% y(0) = 1

t0 = 0;
y0 = 1;
tEnd = 1;
h = 0.1 ;
N = (tEnd - t0)/h;

%% Initializing Solutions
T = [t0:h:tEnd]';
Y = zeros(N+1, 1);
Y(1) = y0;

%% Solving using Euler's Explicit Method
for i = 1:N
    fi = myFunEx2(T(i),Y(i));
    Y(i+1) = Y(i) + h*fi;
end

%% Plot Results and obtain errors
plot(T,Y);
title('Solution of y'' = (2-(2*x)+(3*x^2))*y');
xlabel('t');
ylabel('y(t)');

Ytrue = exp(T-(T.^2)+(T.^3));
ERR = abs(Ytrue - Y)
maxError = max(ERR)
```



O/P:

```
maxError = 3.2271
```

Ex 7.4:
Procedure:

Using Euler's Implicit Method

We are solving the following ODE using **Euler's Implicit Method**:

$$\frac{dy}{dx} = -14ty$$

with the initial condition $y(0) = 1$, over the interval [0,2] with step size h=0.1

1. Define the Problem

- Initial condition: $y(0) = 1$
- Time interval: t∈[0,2]
- Differential equation: $y' - 14ty$

2. Set Parameters

- $t_0 = 0$: initial time
- $y_o = 1$: initial value of y
- $t_{end} = 2$: end time
- Step size h=0.1
- Number of time steps $N = \left(\frac{t_{end} - t_0}{h}\right)$

3. Initialize Variables

- Define the time vector T from t_0 to t_{end} with step size h.
- Create an array Y to store the solution of $y(t)$. Initialize $Y(0) = y_0$

4. Euler's Implicit Method Formula

Implicit Euler's method involves solving for Y(i+1)Y(i+1)Y(i+1) from the following equation:

$$Y(i+1) = Y(i) + h.f(T(i+1), Y(i+1))$$

where $f(t, y) = -14ty$ is the right-hand side of the ODE.

Since $Y(i+1)$ appears on both sides, we need to solve this non-linear equation for $Y(i+1)$. This is typically done using a numerical solver like fsolve in MATLAB.

5. Implementing the Method

- For each time step, use fsolve to solve for $Y(i+1)$ implicitly.
- Update the time and solution vectors after each iteration.

6. Plot the Results

Once the solution is obtained, plot the values of $y(t)$ over the interval [0,2].

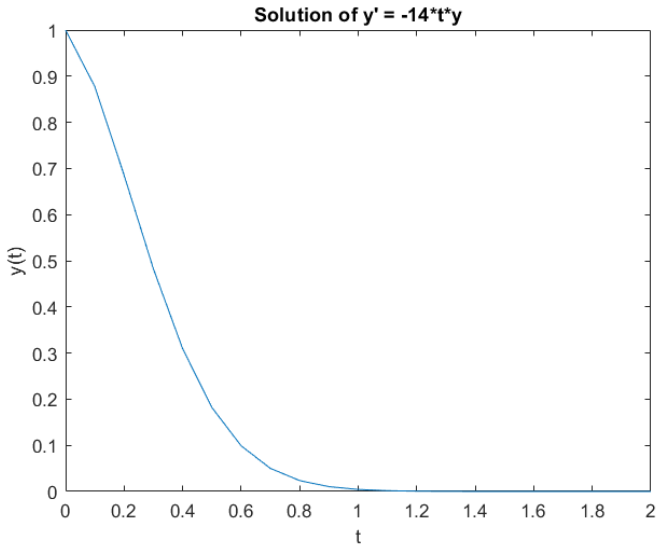
7. Calculate the True Solution and Error

The true solution is $Y(t) = e^{-7t^2}$. Compare the implicit Euler solution to the true solution and calculate the error.

Code:

2. EULER'S IMPLICIT METHOD

```
% ODE-IVP using Euler's Implicit
method
% y' = -14ty
% y(0) = 1

t0 = 0;
y0 = 1;
tEnd = 2;
h = 0.1 ;
N = (tEnd - t0)/h;

%% Initializing Solutions
T = [t0:h:tEnd]';
Y = zeros(N+1, 1);
Y(1) = y0;

%% Solving using Euler's Implicit Method
for i = 1:N
    t = T(i) + h;
    y = fsolve(@(y) y - Y(i) + h*(14*t*y), Y(i));
    T(i+1) = t;
    Y(i+1) = y;
end

%% Plot Results and obtain errors
plot(T,Y);
title('Solution of y" = -14*t*y');
xlabel('t');
ylabel('y(t)');

Ytrue = exp(-7.*(T.^2));
ERR = abs(Ytrue - Y)
maxError = max(ERR)
```



O/P:

maxError = 0.0705

Ex 7.5:

Code:

```
% Solve ODE - IVP using Euler's Implicit method
```

```
% y' = (2-(2*t)+(3*t^2))*y
% y(0) = 1

t0 = 0;
y0 = 1;
tEnd = 1;
h = 0.1 ;
N = (tEnd - t0)/h;

%% Initializing Solutions
T = [t0:h:tEnd]';
Y = zeros(N+1, 1);
Y(1) = y0;

%% Solving using Euler's Implicit Method
for i = 1:N
    t = T(i) + h;
    y = fsolve(@(y) y - Y(i) + h*((2-(2*t)+(3*t^2))*y), Y(i));
    T(i+1) = t;
    Y(i+1) = y;
end

%% Plot Results and obtain errors
plot(T,Y);
title('Solution of y" = (2-(2*t)+(3*t^2))*y');
xlabel('t');
ylabel('y(t)');

Ytrue = exp(T-(T.^2)+(T.^3));
ERR = abs(Ytrue - Y)
maxError = max(ERR)
```

O/P:

maxError = 7.4154

Ex 7.6:

Procedure:

Using RK-2 -Huen's Method

We are solving the following ODE using **RK-2 -Huen's Method**:

$$\frac{dy}{dx} = -14ty$$

with the initial condition $y(0) = 1$, over the interval [0,5] with step size h=0.1

1. Define the Problem

- Initial condition: $y(0) = 1$
- Time interval: t∈[0,5]
- Differential equation: $y' - 14ty$

2. Set Parameters

- $t_0 = 0$: initial time
- $y_o = 1$: initial value of y
- $t_{end} = 5$: end time
- Step size h=0.1
- Number of time steps $N = \left(\frac{t_{end} - t_0}{h}\right)$

3. Initialize Variables

- Define the time vector T from t_0 to t_{end} with step size h.
- Create an array Y to store the solution of $y(t)$. Initialize $Y(0) = y_0$

4. Huen's Method Formula

Huen's method is a two-stage Runge-Kutta method:

1. **Predictor** step (Euler method):

$$k_1 = f(t_i, y_i) = -14t_i y_i$$
$$y_{predict} = y_i + h \cdot k_1$$
$$t_{new} = t_i + h$$

2. **Corrector** step:

$$k_2 = f(t_{new}, y_{predict}) = -14t_{new} y_{predict}$$
$$y_{i+1} = y_i + \frac{h}{2}(k_1 + k_2)$$

5. Implementing the Method

- For each time step, compute the values of k_1 and k_2.

Update the solution using the corrector formula

$$y_{i+1} = y_i + \frac{h}{2}(k_1 + k_2)$$

- **6. Plot the Results**

Once the solution is obtained, plot the values of $y(t)$ over the interval [0,5].

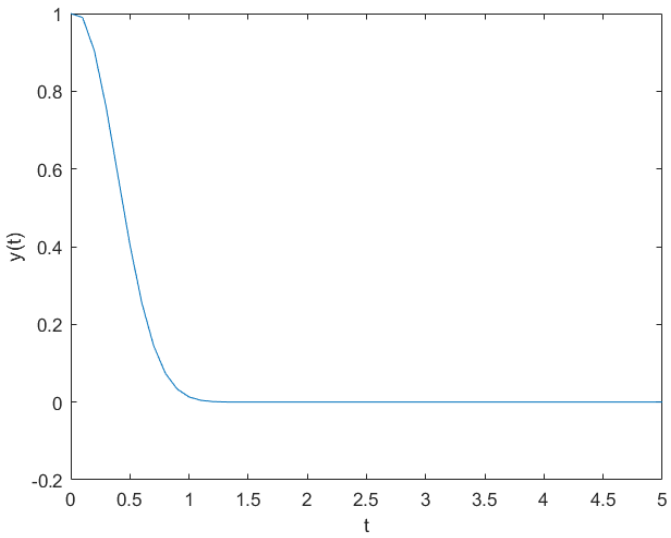
7. Calculate the True Solution and Error

The true solution is $Y(t) = e^{-7t^2}$. Compare the implicit Euler solution to the true solution and calculate the error.

Code:

RUNGE KUTTA (RK-2) - HUEN'S METHOD

```
% Solve ODE - IVP using RK2 - Huen's Method
% y' = -14ty
% y(0) = 1

t0 = 0;
y0 = 1;
tEnd = 5;
h = 0.1 ;
N = (tEnd - t0)/h;

%% Initializing Solutions
T = [t0:h:tEnd]';
Y = zeros(N+1, 1);
Y(1) = y0;

%% Solving using Huen's Method
for i = 1:N
    k1 = myFunEx3(T(i),Y(i));
    tNew = T(i) + h;
    yNew = Y(i) + h*k1;
    k2 = -2*tNew*yNew;
    Y(i+1) = Y(i) + h/2*(k1+k2);
end

%% Plot Results and obtain errors
plot(T,Y);
xlabel('t');
ylabel('y(t)');
Ytrue = exp(-7*(T.^2)) % element by element squaring
ERR = abs(Ytrue - Y)
maxError = max(ERR)
```



Function File:

```
function dy = myFunEx3(x,y)
  dy = -14*x*y;
end
```

O/P:
maxError = 0.2547

Ex 7.7:
Procedure:
RK-2 Mid-point Method
The second-order Runge-Kutta method (midpoint method) can be described as:

Predictor (k1) step:

$$k_1 = f(t_i, y_i) = -14 t_i y_i$$

$$y_{mid} = y_i + \frac{h}{2} k_1$$

$$t_{mid} = t_i + \frac{h}{2}$$

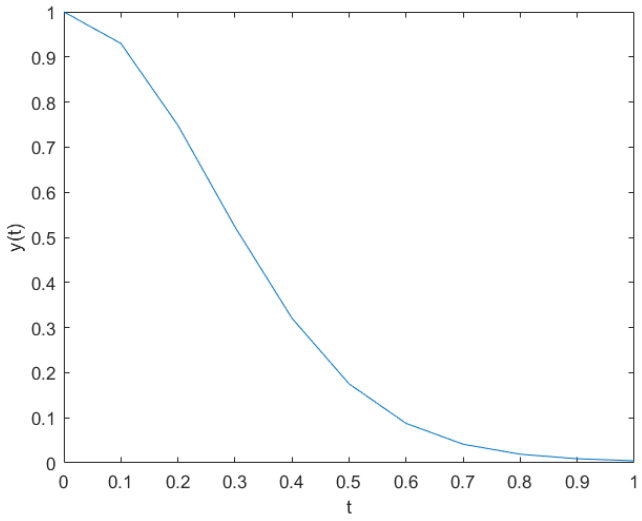
3. **Corrector** step:

$$k_2 = f(t_{mid}, y_{mid}) = -14 t_{mid} y_{mid}$$

$$y_{i+1} = y_i + h.k_2$$

Code:
RUNGE KUTTA (RK-2) – MID POINT METHOD

```
% Solve ODE - IVP using RK2 - Huen's Method
% y' = -14ty
% y(0) = 1

t0 = 0;
y0 = 1;
tEnd = 1;
h = 0.1 ;
N = (tEnd - t0)/h;

%% Initializing Solutions
T = [t0:h:tEnd]';
Y = zeros(N+1, 1);
Y(1) = y0;

%% Solving using Huen's Method
for i = 1:N
   k1 = myFunEx3(T(i),Y(i));
   tNew = T(i) + h/2;
   yNew = Y(i) + h*k1/2;
   k2 = myFunEx3(tNew,yNew);
   Y(i+1) = Y(i) + h*(k2);
end
```



```
%% Plot Results and obtain errors
plot(T,Y);
xlabel('t');
ylabel('y(t)');
Ytrue = exp(-7*(T.^2)) % element by element squaring
ERR = abs(Ytrue - Y)
maxError = max(ERR)
```

Function File:

```
function dy = myFunEx3(x,y)
  dy = -14*x*y;
end
```

O/P:

```
maxError =    0.0095
```

Ex 7.8:

Procedure:

RK-4 Method Formula

The fourth-order Runge-Kutta method can be described by the following steps:

1. **k1 (at t_i)**:
$$k_1 = f(t_i, y_i) = -14 t_i y_i$$
2. **k2 (at $t_i + \frac{h}{2}$)**:
$$k_2 = f\left(t_i + \frac{h}{2}, y_i + \frac{h}{2} k_1\right)$$
3. **k3 (at $t_i + \frac{h}{2}$)**:
$$k_3 = f\left(t_i + \frac{h}{2}, y_i + \frac{h}{2} k_1\right)$$
4. **k4 ($t_i + h$)**:
$$k_1 = f(t_i + h, y_i + h k_3)$$

Update the solution using the formula:

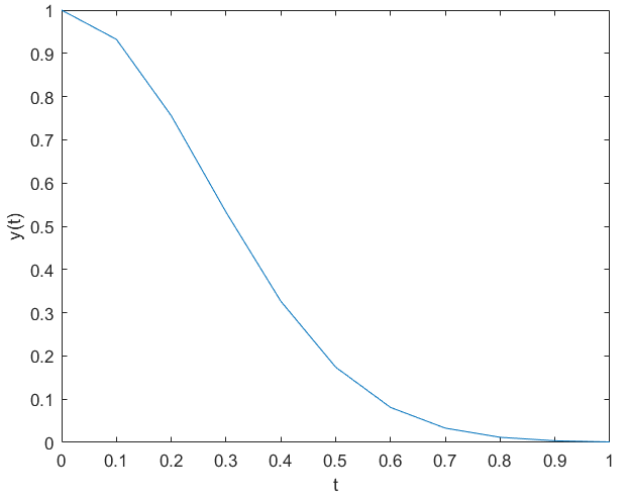
$$y_{i+1} = y_i + \frac{h}{6}(k_1 + 2k_2 + 2k_3 + k_4)$$

Code:

STANDARD RUNGE KUTTA (RK-4) METHOD

```
% Solve ODE - IVP using RK2 - Huen's Method
% y' = -14ty
% y(0) = 1
```

```
t0 = 0;
y0 = 1;
tEnd = 1;
h = 0.1 ;
N = (tEnd - t0)/h;

%% Initializing Solutions
T = [t0:h:tEnd]';
Y = zeros(N+1, 1);
Y(1) = y0;

%% Solving using Standard RK-4 Method
for i = 1:N
    k1 = myFunEx3(T(i),Y(i));
    k2 = myFunEx3(T(i)+h/2,Y(i)+h*k1/2);
    k3 = myFunEx3(T(i)+h/2,Y(i)+h*k2/2);
    k4 = myFunEx3(T(i)+h,Y(i)+h*k3);
    Y(i+1) = Y(i) +
h/6*(k1+2*k2+2*k3+k4);
end
```



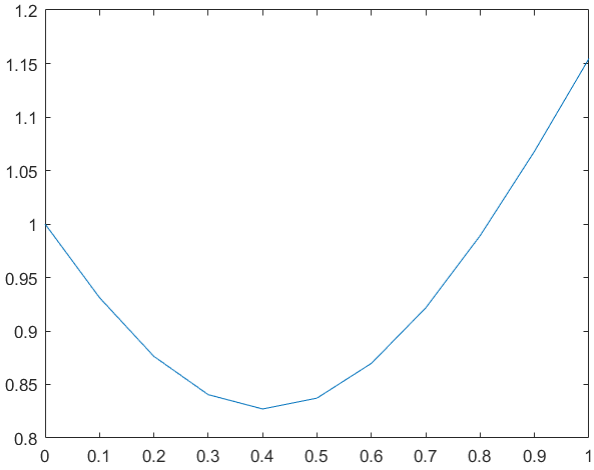

```
%% Plot Results and obtain errors
plot(T,Y);
xlabel('t');
ylabel('y(t)');
Ytrue = exp(-7*(T.^2)) % element by element squaring
ERR = abs(Ytrue - Y)
maxError = max(ERR)
```

Function File:

```
function dy = myFunEx3(x,y)
  dy = -14*x*y;
end
```

O/P:

```
maxError =    5.3192e-04
```



```
% Solve ODE - IVP using Standard RK-4
Method
% y' = -(10*x^2 - 2*y)/(exp(x+y))
% y(0) = 1
```

```
x0 = 0;
y0 = 1;
%xspan = [x0, y0];
xEnd = 1;
h = 0.1 ;
N = (xEnd - x0)/h;

%% Initializing Solutions
X = [x0:h:xEnd]';
Y = zeros(N+1, 1);
Y(1) = y0;

%% Solving using Standard RK-4 Method
for i = 1:N
    k1 = myFunEx4(X(i),Y(i));
    k2 = myFunEx4(X(i)+h/2,Y(i)+h*k1/2);
    k3 = myFunEx4(X(i)+h/2,Y(i)+h*k2/2);
    k4 = myFunEx4(X(i)+h,Y(i)+h*k3);
    Y(i+1) = Y(i) + h/6*(k1+2*k2+2*k3+k4);
end

%% Plot Results and obtain errors
plot(X,Y);
[x, Ytrue] = ode45(@myFunEx4, X, y0) % element by element squaring
ERR = abs(Ytrue - Y)
maxError = max(ERR)
```

Function File:

```
function dy = myFunEx4(x,y)
  dy = (10.*x^2 - 2*y)./(exp(x+y));
end
```

O/P:

```
maxError =    2.3717e-06
```

Ex 7.9:

Procedure:

RK-5 Method Formula

The fifth-order Runge-Kutta (RK-5) method uses six intermediate slopes $k_1, k_2, k_3, k_4, k_5, k_6$ to update the solution.

1. **k1**:

$$k_1 = h \cdot f(t_i, y_i)$$

2. **k2**:

$$k_2 = h \cdot f\left(t_i + \frac{h}{2}, y_i + \frac{k_1}{2}\right)$$

3. **k3**:

$$k_3 = h \cdot f\left(t_i + \frac{h}{4}, y_i + \frac{3}{16}k_1 + \frac{1}{16}k_2\right)$$

4. **k4**:

$$k_4 = h \cdot f\left(t_i + \frac{h}{2}, y_i + \frac{k_3}{2}\right)$$

5. **k5**:

$$k_5 = h \cdot f\left(t_i + \frac{3h}{4}, y_i - \frac{3}{16}k_2 + \frac{6}{16}k_3 + \frac{9}{16}k_4\right)$$

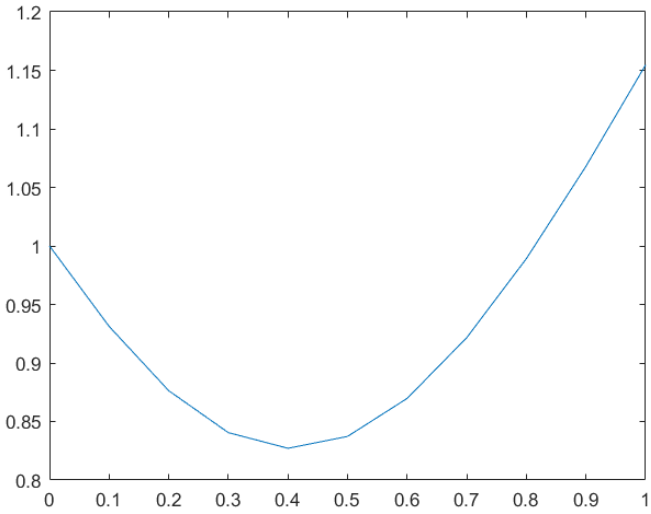
6. **k6**:

$$k_6 = h \cdot f\left(t_i + h, y_i + \frac{1}{7}k_1 + \frac{4}{7}k_2 + \frac{6}{7}k_3 - \frac{12}{7}k_4 + \frac{8}{7}k_5\right)$$

Update the solution:

$$y_{i+1} = y_i + \frac{1}{90}(7k_1 + 32k_3 + 12k_4 + 32k_5 + 7k_6)$$

RUNGE KUTTA (RK-5) METHOD




```
% Solve ODE - IVP using Standard RK-4 Method
% y' = -(10*x^2 - 2*y)/(exp(x+y))
% y(0) = 1

t0 = 0;
y0 = 1;
tEnd = 1;
h = 0.1 ;
N = (tEnd - t0)/h;

%% Initializing Solutions
T = [t0:h:tEnd]';
Y = zeros(N+1, 1);
Y(1) = y0;

%% Solving using Standard RK-5 Method
for i = 1:N
    k1 = h*myFunEx4(T(i),Y(i));
    k2 = h*myFunEx4(T(i)+h/2,Y(i)+k1/2);
    k3 = h*myFunEx4(T(i)+h/4,Y(i)+3*k1/16+k2/16);
    k4 = h*myFunEx4(T(i)+h/2,Y(i)+k3/2);
```

```
    k5 = h*myFunEx4(T(i)+3*h/4,Y(i)-3*k2/16+6*k3/16+9*k4/16);
    k6 = h*myFunEx4(T(i)+h,Y(i)+k1/7+4*k2/7+6*k3/7-12*k4/7+8*k5/7);
    Y(i+1) = Y(i) + 1/90*(7*k1+32*k3+12*k4+32*k5+7*k6);
end

%% Plot Results and obtain errors
plot(X,Y);
[x, Ytrue] = ode45(@myFunEx4, X, y0) % element by element squaring
ERR = abs(Ytrue - Y)
maxError = max(ERR)
```

Function File:

```
function dy = myFunEx4(x,y)
  dy = (10.*x^2 - 2*y)./(exp(x+y));
end
```

O/P:

```
maxError =    1.4692e-08
```

Expt No. 8	LINEAR EQUATIONS	Date of Expt:

Ex 8.1:
Procedure:
Naive Gauss Elimination

1. **Form the Augmented Matrix**:
   Combine the coefficient matrix A and the right-hand side vector b into an augmented matrix Ab.
2. **Forward Elimination**:
   Perform the following steps to eliminate the elements below the diagonal, converting the system into an upper triangular form:
   - For each row iii from 2 to n, where n is the number of rows in the matrix:
     - Divide the first element of the *i*-th row by the pivot element (the leading element of the current row).
     - Subtract the scaled row from the current row to eliminate the *i*-th element in the column.
   - Continue this process for each column, choosing the diagonal element of the current row as the pivot.
3. **Back Substitution**:
   After forward elimination, you will have an upper triangular matrix.
   Solve for the unknowns starting from the last row and working upward:
   - Start by solving for x_n in the last equation.
   - Substitute x_n into the second-last equation to solve for x_{n-1}.
   - Continue substituting and solving for each unknown, moving upwards through the system.
4. **Solution**:
   Once back substitution is complete, you will have the values of all the unknowns $x_1, x_2, x_3 \ \ x_n$.

Code:
GAUSS ELIMINATION METHOD

```
% Solve Ax = b using Naive Gauss Elimination
A = [2 8 10; 12 17 23; 34 47 -28];
b = [41; 7; 9];
```

```
%% Gauss Elimination
% Get Augmented Matrix
Ab = [A, b];
n = length(A);

% With A(1,1) as Pivot Element
for i = 2:3
    alpha = A(i,1)/A(1,1);
    Ab(i,:) = Ab(i,:) - alpha*Ab(1,:);
end

% With A(2,2) as Pivot Element
i = 3;
alpha = Ab(i,2)/Ab(2,2);
Ab(i,:) = Ab(i,:) - alpha*Ab(2,:);

%% Back Substitution
x = zeros(n,1);
x(3) = Ab(3,end)/Ab(3,3);
for i = n-1:-1:1
    x(i) = (Ab(i,end) - Ab(i,i+1:n)*x(i+1:n))/Ab(i,i);
end
```

O/P:

```
x =
  -10.3432
    7.6858
    0.0200
```

```
A = [1 1 1 1; 2 1 3 6; 3 4 -2 9; 1 5 7 9];
b = [4; 7; 9; 5];
x = naiveGaussElimination(A, b)
```

Function File:

```
function x = naiveGaussElimination(A, b)
    n = size(A, 1); % Determine the order of the matrix
    Ab = [A, b]; % Create the augmented matrix

    % Forward elimination
    for i = 1:n-1
```

```
        for j = i+1:n
            alpha = Ab(j,i) / Ab(i,i);
            Ab(j,:) = Ab(j,:) - alpha * Ab(i,:);
        end
    end

    % Back substitution
    x = zeros(n, 1); % Preallocate the solution vector x
    x(n) = Ab(n,end) / Ab(n,n); % Compute the last variable
    for i = n-1:-1:1 % Loop from the second-to-last row up to the first row
        x(i) = (Ab(i,end) - Ab(i,i+1:n) * x(i+1:n)) / Ab(i,i);
    end
end
```

O/P:

```
x =

    3.6531
    0.2755
    0.3367
   -0.2653
```

Ex 8.2:
Procedure:
LU Decomposition

1. **Initialize Matrices:**
    - Define the matrix A and the vector b.
    - Create the augmented matrix Ab by appending b to A.
    - Initialize the identity matrix L of the same size as A.
2. **Gauss Elimination:**
    - **For each column j:**
        - **For each row i below the diagonal:**
            - Compute the multiplier α using the element in the pivot column.
            - Update the L matrix with α.
            - Eliminate the elements below the pivot by subtracting α times the pivot row from the current row.

3. **Extract Matrices:**
    - Extract the upper triangular matrix U from the augmented matrix Ab.

Summary of Steps

1. Define A and b.
2. Create $Ab = [A, b]$ and initialize L.
3. Perform Gauss elimination to make A upper triangular:
    - Eliminate below each pivot element.
4. Extract U from Ab.

This procedure will decompose matrix A into lower triangular matrix L and upper triangular matrix U, such that $A = LU$.

Code:

LU DECOMPOSITION

```
% LU Decomposition using Naive Gauss Elimination Method
A = [11 12 13;27 14 3; 13 14 -24];
b= [14;17;9];

%% Gauss elimination
% creating augmented matrix
Ab = [A,b];
n = length(A);
L = eye(n);

%% with A(1,1) as pivot element
for i = 2:3
alpha = Ab(i,1)/Ab(1,1);
L(i,1) = alpha;
Ab(i,:) = Ab(i,:)-alpha*Ab(1,:);
end

%% with A(2,2) as pivot element
for i=3;
alpha = Ab(i,2)/Ab(2,2);
L(i,2) = alpha;
Ab(i,:) = Ab(i,:)-alpha*Ab(2,:);
End

U = Ab(1:n,1:n);
```

O/P:

L =

```
   1.0000         0         0
   2.4545    1.0000         0
   1.1818    0.0118    1.0000
```

U =

```
  11.0000   12.0000   13.0000
        0  -15.4545  -28.9091
        0         0  -39.0235
```

Ex 8.3:

Procedure:

LU Decomposition with Partial Pivoting

1. **Initialize Matrices:**
    - Define matrix A and vector b.
    - Create augmented matrix Ab by appending b to A.
2. **Gauss Elimination with Partial Pivoting:**

Step 1: Pivoting for the First Column

- Find the largest element in the first column of Ab.
- Swap the current row with the row containing the largest element.
- Perform elimination to zero out elements below the pivot in the first column.

Step 2: Pivoting for the Second Column

- Find the largest element in the remaining rows of the second column.
- Swap the current row with the row containing the largest element (adjust row index if needed).
- Perform elimination to zero out elements below the pivot in the second column.

3. **Back Substitution:**
    - Initialize solution vector x with zeros.
    - Compute values for x starting from the last equation and working backward.

This procedure decomposes matrix A into a lower triangular matrix L and an upper triangular matrix U, while solving $AX = b$ using partial pivoting to enhance numerical stability.

Code:

LU DECOMPOSITION AND PARTIAL PIVOTING

```
% solving AX = b using Gauss Elimination with Partial Pivoting
A = [11 12 13;27 14 3; 13 14 -24];
b= [14;17;9];

%% Gauss elimination
% creating augmented matrix
Ab = [A, b];
n= length(A);

%% with A(1,1) as pivot element
% Row exchange to ensure A(1,1) is the largest in column 1
col1=Ab(:,1);
[dummy,idx] = max(col1);
dummy = Ab(1,:);
Ab(1,:) = Ab(idx,:);
Ab(idx,:) = dummy;
for i = 2:3
alpha = Ab(i,1)/Ab(1,1);
Ab(i,:) = Ab(i,:)-alpha*Ab(1,:);
end

%% with A(2,2) as pivot element
% Row exchange to ensure A(2,2) is the largest in column 2
col2=Ab(2:end,2);
[dummy,idx] = max(col2);
dummy = Ab(2,:);
Ab(2,:) = Ab(idx,:);
Ab(idx,:) = dummy;
for i=3;
alpha = Ab(i,2)/Ab(2,2);
Ab(i,:) = Ab(i,:)-alpha*Ab(2,:);
end

%% Back substitution
x = zeros(3,1);
x(3) = Ab(3,end)/Ab(3,3);
x(2) = (Ab(2,end) - Ab(2,3)*x(3))/Ab(2,2);
x(1) = (Ab(1,end) - (Ab(1,3)*x(3)+Ab(1,2)*x(2)))/Ab(1,1);
```

O/P:

```
Ab =

   27.0000   14.0000    3.0000   17.0000
         0    6.2963   11.7778    7.0741
         0         0  -39.0235   -7.3412
x =

    0.2086
    0.7716
    0.1881
```

Ex 8.4:
Procedure:
Gauss-Seidel Method (Iterative Method)

1. **Initialize:**
   - Define matrix A and vector b.
   - Create the initial guess for vector x (e.g., zeros).
   - Set max_iter for maximum iterations and tolerance for convergence criteria.
2. **Gauss-Seidel Iterations:**
   - For each iteration up to max_iter:
     - Store the current x values as x_old.
     - For each variable k:

$$Sum = \sum_{j \neq k} A(k,j) \cdot x(j)$$

       - Compute the sum of the known variables:
       - Update $x(k)$:

$$x(k) = \frac{b(k) - sum}{A(k,k)}$$

     - Compute the error:

$$error = ||x - x_{old}||$$

     - Check if the error is less than tolerance for convergence:
       - If converged, exit the loop.
3. **Display Solution:**
   - Output the final values of x.

Code:

GAUSS SEIDAL METHOD (ITERATIVE METHOD)

```
% Solving AX = b using Gauss-Seidel iteration method
A = [4 -1 0; -1 4 -1; 0 -1 3];
b = [8; 7; 3];
Ab = [A, b];

% Initialising
n = 3;
x = zeros(n, 1);
max_iter = 25;
tolerance = 1e-5;

%% Gauss-Seidel iterations
for iter = 1:max_iter
    x_old = x;
    for k = 1:n
        sum = 0;
        for j = 1:n
            if j ~= k
                sum = sum + A(k, j) * x(j);
            end
        end
        x(k) = (b(k) - sum) / A(k, k);
    end

    % Error calculation
    err = norm(x - x_old);
    disp(['Iteration ', num2str(iter), ': Error ', num2str(err)])

    % Convergence check
    if err < tolerance
        disp('Convergence achieved.');
        break;
    end
end

% Display the solution
disp('Solution X:');
disp(x);
```

O/P:

```
Iteration 1: Error 3.4821
```

Iteration 2: Error 0.82932
Iteration 3: Error 0.16967
Iteration 4: Error 0.024743
Iteration 5: Error 0.0036084
Iteration 6: Error 0.00052622
Iteration 7: Error 7.6741e-05
Iteration 8: Error 1.1191e-05
Iteration 9: Error 1.6321e-06
Convergence achieved.
Solution X:
2.7317
2.9268
1.9756

Ex 8.5:
Procedure:
Jacobi Method (Iterative Method)

1. **Initialize:**
   - Define matrix A and vector b.
   - Create the initial guess for vector x (e.g., zeros).
   - Set max_iter for maximum iterations and tolerance for convergence criteria.
2. **Jacobi Iterations:**
   - For each iteration up to max_iter:
     - Initialize a new vector x_{new} as a copy of x.
     - For each variable k:
       - Compute the sum of the off-diagonal elements:

$$Sum = \sum_{j \neq k} A(k,j) \cdot x(j)$$

       - Update $x_{new}(k)$:

$$x_{new}(k) = \frac{b(k) - sum}{A(k.k)}$$

       Compute the error: $error = \| xnew - x \|$
     - Update x to x_{new} .
     - Display the current iteration number and error.
     - Check if the error is less than tolerance for convergence:
       - If converged, exit the loop.
3. **Display Solution:**
   - Output the final values of x.

- Optionally, compute and display the residual to check if the solution satisfies the original equations:

$$residual = \| A \cdot x - b \|$$

Code:

JACOBI METHOD (ITERATIVE METHOD)

```
% Solving AX = b using Jacobi iteration method
A = [10 -1 2;
    -1 10 -2;
    1  1  5];
b  = [7; 6; 5];
Ab = [A, b];

% Initialising
n = size(A,1);
x = zeros(n, 1);
x_new = x;
max_iter = 100; % Increase the number of iterations if needed
tolerance = 1e-5;

% Jacobi iterations
for iter = 1:max_iter
   for k = 1:n
      % Calculate the sum for off-diagonal elements
      sum = 0;
      for j = 1:n
         if j ~= k
            sum = sum + A(k,j) * x(j);
         end
      end

      % Update x_new
      x_new(k) = (b(k) - sum) / A(k, k);
   end

   % Calculate error and update x
   err = norm(x_new - x);
   x = x_new;

   % Display current iteration and error
   disp(['Iteration ', num2str(iter), ': Error ', num2str(err)])
```

```
    % Convergence check
    if err < tolerance
        disp('Convergence achieved.');
        break;
    end
end

% Display the solution
disp('Solution X:');
disp(x);

% Optional: Check if solution satisfies original equations
residual = norm(A*x - b);
disp('Residual (A*x - b):');
disp(residual);
```

O/P:

```
Iteration 1: Error 1.3601
Iteration 2: Error 0.40012
Iteration 3: Error 0.10617
Iteration 4: Error 0.0040012
Iteration 5: Error 0.0010617
Iteration 6: Error 4.0012e-05
Iteration 7: Error 1.0617e-05
Iteration 8: Error 4.0012e-07
Convergence achieved.
Solution X:
   0.6384
   0.8061
   0.7111

Residual (A*x - b):
  1.0376e-06
```

Ex 8.6:

Procedure:

Tri-Diagonal Matrix Algorithm (TDMA)

1. **Initialize:**
    - Define the subdiagonal vector e, diagonal vector f, superdiagonal vector g, and the right-hand side vector r.
    - Ensure the vectors are correctly sized for the problem (i.e., e and g should be of size $n-1$, and f and r should be of size n).

2. **Forward Elimination:**
   - For each k from 2 to n:
     - Compute the factor to eliminate the subdiagonal element:
$$factor = \frac{e(k)}{f(k-1)}$$
     - Update the diagonal element:
$$f(k) = f(k) - factor \cdot g(k-1)$$
     - Update the right-hand side element:
$$r(k) = r(k) - factor \cdot r(k-1)$$
3. **Back Substitution:**
   - Solve for the last unknown:
$$x(n) = \frac{r(n)}{f(n)}$$
   - For each k from $n-1$:
     - Compute the unknown x(k)x(k)x(k) using:
$$x(k) = \frac{r(k) - g(k) \cdot x(k+1)}{f(k)}$$
4. **Output Solution:**
   - Display the computed values of x.

Code:

TRI-DIAGONAL MATRIX ALGORITHM

```
% Solving linear equation using TDMA algorithm
% Input
% e = Subdiagonal vector
% f = Diagonal vector
% g = Superdiagonal vector
% r = Right hand side vector

%% Problem Setting
n = 4;
e = [2;-1;10;0];
f = [2;4;6;5];
g = [0;1;5;3];
r = [1;2;3;5];

%% Forward Elimination
for k = 2:n
factor = e(k)/f(k-1);
f(k) = f(k) - factor *g(k-1);
r(k) = r(k) - factor* r(k-1);
```

```
end

%% Back Substitution
x(n) = r(n)/f(n);
for k = n-1:-1:1
x(k) = (r(k) - g(k)*x(k+1))/f(k);
end
```

O/P:

```
x =

    0.5000
    0.7391
   -2.3571
    1.0000

x =

    0.5000
    1.2143
   -2.3571
    1.0000

x =

    0.5000
    1.2143
   -2.3571
    1.0000
```

Expt No. 9	NON-LINEAR EQUATIONS	Date of Expt:

Ex 9.1:
Procedure:
Bisection Method (Bracketing Method)

1. **Define the Function:**
   - Specify $f(x)$.
2. **Set Initial Guesses:**
   - Choose xl and xu such that $f(xl)$ and $f(xu)$ have opposite signs.
3. **Evaluate Function at Initial Guesses:**
   - Compute $fl = f(xl)$ and $fu = f(xu)$.
4. **Check Validity:**
   - Ensure $fl \times fu < 0$ (i.e., the function values at the guesses have opposite signs).
5. **Set Tolerance and Initialize Error:**
   - Define tolerance tol and initialize err.
6. **Iterate:**
   - While err>tol:
     - Compute the midpoint

$$xnew = \frac{xl + xu}{2}$$

     - Evaluate $f(xnew)$.
     - Update xl or xu based on the sign of $f(x_{new})$.
     - Update err as $err = | xl - xu|$
7. **Display Results:**
   - Print the final values of xl, xu, and the approximate root x_{new}.

Code:
BISECTION METHOD (BRACKETING METHOD)

```
% Define the function f(x)
f = @(x) x^2 - 4*x + 3;

% Initial guesses
xl = 0;  % Lower bound
xu = 2;  % Upper bound (chosen so that the root at x = 1 lies between xl and xu)

% Evaluate f at the initial guesses
```

```
fl = f(xl);
fu = f(xu);

% Check if the initial guesses are valid
if (fl * fu > 0)
    error('Initial guess should have different signs')
end

% Define the error tolerance
tol = 1e-6;  % Tolerance for the stopping criterion
err = abs(xl - xu);

% Bisection method loop
while err > tol
    xnew = (xl + xu) / 2;  % Midpoint
    fnew = f(xnew);        % Evaluate f at the midpoint

    if (fl * fnew > 0)
        xl = xnew;  % Update lower bound
        fl = fnew;  % Update function value at lower bound
    else
        xu = xnew;  % Update upper bound
    end

    err = abs(xl - xu);  % Update error
end

% Print current values
fprintf('xl = %.4f\n', xl);
fprintf('fl = %.4f\n', fl);
fprintf('fu = %.4f\n', fu);
fprintf('xu = %.4f\n\n', xu);

% Display the final approximation
fprintf('The root is approximately at x = %f\n', xnew);
```

O/P:

```
xl = 1.0000
fl = 0.0000
fu = -1.0000
xu = 1.0000

The root is approximately at x = 0.999999
```

Ex 9.2:
Procedure:

Regula Falsi Method (Bracketing Method)

1. **Define the Function:**
   - Specify $f(x)$.
2. **Set Initial Guesses:**
   - Choose xl and xu such that $f(xl)$ and $f(xu)$ have opposite signs.
3. **Evaluate Function at Initial Guesses:**
   - Compute $fl = f(xl)$ and $fu = f(xu)$.
4. **Check Validity:**
   - Ensure $fl \times fu < 0$ (i.e., the function values at the guesses have opposite signs).
5. **Set Tolerance and Maximum Iterations:**
   - Define the *tolerance* and maximum number of iterations *maxIterations*.
6. **Iterate:**
   - For *i=1* to *maxIterations*:
     - Compute the new estimate $x_{new} = \frac{xl - fl \times (xu - xl)}{fu - fl}$
     - Evaluate $f(xnew)$
     - Update the bounds based on the sign of $f(x_{new})$:
       - If $fl \times f(xnew) > 0$, set $xl = x_{new}$ and $fl = f(x_{new})$
       - Otherwise, set $xu = x_{new}$ and $fu = f(x_{new})$.
     - Check for convergence:
       - If $| f(x_{new}) | < tolerance$, stop iterating
7. **Display Results:**
   - Print the approximate root and the number of iterations.

Code:

REGULA FALSI METHOD (BRACKETING METHOD)

```
% Define the function f(x)
f = @(x) exp(x) - 3*x;

% Adjusted initial guesses
xl = 0; % Lower bound
xu = 1; % Upper bound
```

```
% Evaluate f at the initial guesses
fl = f(xl);
fu = f(xu);

% Check if the initial guesses bracket a root
if (fl * fu > 0)
    error('Initial guess should have different signs');
end

% Define the tolerance and maximum number of iterations
tolerance = 1e-6;
maxIterations = 100;

% Regula Falsi (False Position) method
for i = 1:maxIterations
    xnew = xl - fl * (xu - xl) / (fu - fl); % Compute the root estimate
    fnew = f(xnew); % Evaluate the function at the new estimate

    % Update the bounds
    if (fl * fnew > 0)
        xl = xnew;
        fl = fnew;
    else
        xu = xnew;
        fu = fnew;
    end

    % Check for convergence
    if abs(fnew) < tolerance
        break;
    end
end

% Display the results
fprintf('Approximate root: %f\n', xnew);
fprintf('Number of iterations: %d\n', i);
```

O/P:

Approximate root: 0.619062
Number of iterations: 12

Ex 9.3:
Procedure:
Secant Method (Open Method)

1. **Define the Function:**
   - Specify $f(x)$.
2. **Set Initial Guesses:**
   - Choose initial guesses x_0 and x_1.
3. **Set Parameters:**
   - Define the maximum number of iterations maxiter\text{maxiter}maxiter and the tolerance *tolx*.
4. **Secant Method Loop:**
   - For $i = 1$ to maxiter:
     - Compute $f(x_0)$ and $f(x_1)$.
     - Update x using the Secant formula:

$$x = x_1 - \frac{f(x_1) \cdot (x_1 - x_0)}{f(x_1) - f(x_0)}$$

     - Calculate the error $err = | x - x_1 |$
     - If *err* < *tolx*, stop iterating.
     - Update guesses:
       - Set $x_0 = x_1$
       - Set $x_1 = x$
5. **Display the Result:**
   - Print the approximate root x.

Code:
SECANT METHOD (OPEN METHOD)

```
% Define the function f(x)
f = @(x) cos(x) - x;

% Initial guesses
x0 = 0;
x1 = 1;
maxiter = 50;
tolx = 1e-4;

% Secant method loop
for i = 1:maxiter
   fx0 = f(x0);
```

```
    fx1 = f(x1);

    % Update x using the Secant formula
    x = x1 - (fx1 * (x1 - x0)) / (fx1 - fx0);

    % Calculate the error
    err = abs(x - x1);
    if err < tolx
        break;
    end

    % Update the guesses
    x0 = x1;
    x1 = x;
end

% Display the result
fprintf('The root is approximately at x = %f\n', x);
```

O/P:
The root is approximately at x = 0.739085

Ex 9.4:
Procedure:

Fixed Point Iterations (Open Method)

1. **Define Initial Parameters:**
    - Set initial guess x_0.
    - Define the maximum number of iterations *maxiter* and the tolerance *tolx*.
2. **First Rearrangement: $x = \sqrt{sin(x)}$**
    - Initialize x and x_{old} with x_0.
    - **Fixed Point Iteration Loop:**
        - For $i = 1$ to *maxiter*:
            - Update x using the formula $x = \sqrt{\sin(x)}$
            - Compute the error $err = |x - x_{old}|$
            - Update x_{old} with the current x.
            - If *err* < *tolx*, stop iterating.
    - Store the result in x_1.
3. **Second Rearrangement: $x = \sin^{-1}(x^2)$**

- Initialize x and x_{old} with x_0.
- **Fixed Point Iteration Loop:**
  - For $i = 1$ to *maxiter*:
    - Update x using the formula $x = \sin^{-1}(x^2)$
    - Compute the error $err = |x - x_{old}|$
    - Update x_{old} with the current x.
    - If *err* < *tolx*, stop iterating.
- Store the result in x_2.

4. **Display the Results:**
   - Print x_1 for the first rearrangement.
   - Print x_2 for the second rearrangement.

Code:

FIXED POINT ITERATIONS (OPEN METHOD)

```
% Define the initial guess and parameters
x0 = 0.5; % Initial guess (adjust as needed)
maxiter = 50;
tolx = 1e-4;

% First rearrangement: x = sqrt(sin(x))
x = x0;
xold = x0;
for i = 1:maxiter
    x = sqrt(sin(x)); % Update using the first rearrangement
    err = abs(x - xold);
    xold = x;
    if (err < tolx)
        break;
    end
end
x1 = x; % Store the result from the first rearrangement

% Second rearrangement: x = asin(x^2)
x = x0;
xold = x0;
for i = 1:maxiter
    x = asin(x^2); % Update using the second rearrangement
    err = abs(x - xold);
    xold = x;
    if (err < tolx)
```

```
        break;
    end
end
x2 = x; % Store the result from the second rearrangement

% Display the outputs
fprintf('Output using first rearrangement (x = sqrt(sin(x))): x = %f\n', x1);
fprintf('Output using second rearrangement (x = asin(x^2)): x = %f\n', x2);
```

O/P:

Output using first rearrangement (x = sqrt(sin(x))): x = 0.876699
Output using second rearrangement (x = asin(x^2)): x = 0.000000

Ex 9.5:

Procedure:

Newton-Raphson Method (Open Method)

1. **Define Initial Parameters:**
   - Set the initial guess x_0.
   - Define the maximum number of iterations *maxiter* and the tolerance *tolx*.
2. **Newton-Raphson Iteration Loop:**
   - Initialize x and x_{old} with x_0.
   - **For** $i = 1$ to maxiter:
     - Define the function $f(x)$ and its derivative $f'(x)$:
       - $f(x) = sin(x) - \frac{x}{2}$
       - $f'(x) = cos(x) - \frac{1}{2}$
     - Update x using the Newton-Raphson formula:

$$x = x - \frac{f(x)}{f'(x)}$$

     Compute the error $err = | x - x_{old} |$
     - Update x_{old} with the current x.
     - If *err* < *tolx*, stop iterating.
3. **Display the Result:**
   - Print the final value of x as the approximate root.

Code:

NEWTON RAPHSON METHOD (OPEN METHOD)

```
% Initial guess
x0 = 2; % Adjust if necessary
maxiter = 50;
tolx = 1e-4;

% Newton-Raphson loop
x = x0;
xold = x0;
for i = 1:maxiter
    % Define the function f(x) and its derivative df(x)
    f = sin(x) - x/2;
    df = cos(x) - 1/2;

    % Update x using the Newton-Raphson formula
    x = x - f/df;

    % Calculate the error
    err = abs(x - xold);
    xold = x;

    % Check if the error is within the tolerance
    if (err < tolx)
        break;
    end
end

% Display the result
fprintf('The root is approximately at x = %f\n', x);
```

O/P:

```
The root is approximately at x = 1.895494
```

Expt No. 10	ALGEBRA AND TRANSFORMS	Date of Expt:

Ex 10.1:
Procedure:
Solving Quadratic Equations Using solve Function

1. **Declare the Symbolic Variable:**
   - Use syms x to define x as a symbolic variable.
2. **Formulate Each Quadratic Equation:**
   - Define each quadratic equation in the form $ax^2 + bx + c = 0$ using symbolic expressions.
3. **Solve the Equations:**
   - Apply the *solve* function to each equation to find the roots.
4. **Output the Solutions:**
   - Display the solutions for each quadratic equation.

Code:

SOLVING QUADRACTIC EQUATION USING SOLVE FUNCTION - REAL AND IMAGINARY ROOTS

```
syms x

% Equation 1: 2x^2 - 4x + 2 = 0
eqn1 = 2*x^2 - 4*x + 2 == 0;
sol1 = solve(eqn1, x);

% Equation 2: 3x^2 + 6x + 3 = 0
eqn2 = 3*x^2 + 6*x + 3 == 0;
sol2 = solve(eqn2, x);

% Equation 3: x^2 + 2x + 6 = 0
eqn3 = x^2 + 2*x + 6 == 0;
sol3 = solve(eqn3, x);

% Equation 4: 3x^2 + 3x + 7 = 0
eqn4 = 3*x^2 + 3*x + 7 == 0;
sol4 = solve(eqn4, x);

% Display solutions
sol1, sol2, sol3, sol4
```

```
O/P:
sol1 =

1
1

sol2 =

-1
-1

sol3 =

- 1 - 5^(1/2)*1i
- 1 + 5^(1/2)*1i

sol4 =

- (3^(1/2)*5i)/6 - 1/2
  (3^(1/2)*5i)/6 - ½
```

Ex 10.2:

Procedure:

Solving Quadratic Equations Using the Quadratic Formula

1. **Identify Coefficients:**
   - For each quadratic equation of the form $ax^2 + bx + c = 0$, determine the coefficients $a, b, and\ c$.
2. **Compute the Discriminant:**
   - Calculate the discriminant $\Delta = b^2 - 4ac$.
3. **Calculate the Roots:**
   - Use the quadratic formula to find the roots:

$$x_{1,2} = \frac{-b \pm \sqrt{\Delta}}{2a}$$

   - Compute the roots x_1 and x_2 for each equation, considering both the positive and negative square root of the discriminant.
4. **Handle Complex Roots:**
   - If the discriminant is negative, the square root will be imaginary. Ensure to compute the roots as complex numbers.
5. **Display the Solutions:**
   - Present the solutions for each quadratic equation, including both real and imaginary parts if applicable.

Code:

SOLVING QUADRACTIC EQUATION USING QUADRACTIC FORMULA- REAL AND IMAGINARY ROOTS

```
% Equation 1: 2x^2 - 4x + 2 = 0
a1 = 2; b1 = -4; c1 = 2;
x1_1 = ((-b1) + sqrt(b1^2 - 4*a1*c1)) / (2*a1);
x1_2 = ((-b1) - sqrt(b1^2 - 4*a1*c1)) / (2*a1);

% Equation 2: 3x^2 + 6x + 3 = 0
a2 = 3; b2 = 6; c2 = 3;
x2_1 = ((-b2) + sqrt(b2^2 - 4*a2*c2)) / (2*a2);
x2_2 = ((-b2) - sqrt(b2^2 - 4*a2*c2)) / (2*a2);

% Equation 3: x^2 + 2x + 6 = 0
a3 = 1; b3 = 2; c3 = 6;
x3_1 = ((-b3) + sqrt(b3^2 - 4*a3*c3)) / (2*a3);
x3_2 = ((-b3) - sqrt(b3^2 - 4*a3*c3)) / (2*a3);

% Equation 4: 3x^2 + 3x + 7 = 0
a4 = 3; b4 = 3; c4 = 7;
x4_1 = ((-b4) + sqrt(b4^2 - 4*a4*c4)) / (2*a4);
x4_2 = ((-b4) - sqrt(b4^2 - 4*a4*c4)) / (2*a4);
```

```
% Display solutions
[x1_1, x1_2; x2_1, x2_2; x3_1, x3_2; x4_1, x4_2]
```

O/P:

```
ans =

   1.0000 + 0.0000i   1.0000 + 0.0000i
  -1.0000 + 0.0000i  -1.0000 + 0.0000i
  -1.0000 + 2.2361i  -1.0000 - 2.2361i
  -0.5000 + 1.4434i  -0.5000 - 1.4434i
```

Ex 10.3:
Code:

```
LIMITS
syms x

% Example 1: Exponential and Polynomial Function
limit_exp_poly_1 = limit(exp(x)/(x^2 + 1), x, Inf);
limit_exp_poly_2 = limit(x^3/(exp(x) - 1), x, Inf);

% Example 2: Trigonometric and Polynomial Function
limit_trig_poly_1 = limit(sin(x)/x, x, 0);
limit_trig_poly_2 = limit((x^2 + 2*x + 1)/(cos(x) + 2), x, Inf);

% Display the results
[limit_exp_poly_1, limit_exp_poly_2; limit_trig_poly_1, limit_trig_poly_2]
```

O/P:

```
ans =

[Inf,   0]
[  1, Inf]
```

```
syms x

% Example 1: Function k(x) = sin(x)/abs(x)
k = sin(x)/abs(x);
limit_k_left = limit(k, x, 0, 'left');
limit_k_right = limit(k, x, 0, 'right');

% Example 2: Function m(x) = |x|/x
```

```
m = abs(x)/x;
limit_m_left = limit(m, x, 0, 'left');
limit_m_right = limit(m, x, 0, 'right');

% Display the results
[limit_k_left, limit_k_right; limit_m_left, limit_m_right]
```

O/P:

```
ans =

[-1, 1]
[-1, 1]
```

Ex 10.4:
Procedure:
Differentiation and Solving Differential Equations

1. **Differentiate a Function:**
   - Use diff(function) to find the derivative of the function.
2. **Higher Order Differentiation:**
   - Use diff(function, n) to find the n-th order derivative of the function.
3. **Solve a Differential Equation:**
   - Use dsolve('differential_equation') to find the general solution of the differential equation.
4. **Solve Differential Equation with Initial and Boundary Conditions:**
   - Use dsolve('differential_equation', 'initial_condition1', 'initial_condition2', ...) to find a particular solution with specified initial and boundary conditions.
5. **Display Results:**
   - Collect and display the results from differentiation and differential equation solutions.

Code:
DIFFERENTIATION

```
syms x

% Differentiation Example
diff_example = diff(4*x^2);
% Higher Order Differentiation Example
```

```
diff_high_order_example = diff(2*x^5, 2);

% Differential Equation Solution Example
syms y(t)
dsolve_example = dsolve('D2y + 3*y = 0');

% Differential Equation with Initial and Boundary Conditions Example
dsolve_ibc_example = dsolve('D2y + 3*y = 0', 'y(0) = 1', 'Dy(0) = -1');

% Display the results
[diff_example, diff_high_order_example, dsolve_example,
dsolve_ibc_example]
```

O/P:

```
ans =

[8*x, 40*x^3, C1*cos(3^(1/2)*t) - C2*sin(3^(1/2)*t), cos(3^(1/2)*t) -
(3^(1/2)*sin(3^(1/2)*t))/3]
```

```
syms x t

% Non-Linear Differential Equation Example
syms y(t)
nonlinear_ode = (diff(y, t) - y)^2 == 4;
nonlinear_cond = y(0) == 1;
nonlinear_ysol(t) = dsolve(nonlinear_ode, nonlinear_cond);

% Display Non-Linear Differential Equation Solution
disp('Solution to Non-Linear Differential Equation:');
disp(nonlinear_ysol(t));

% Maxima and Minima Example
y_function = x^4 - 6*x^3 + 11*x^2 - 6*x;
m_derivative = diff(y_function);
stationary_points = solve(m_derivative);

% Display Stationary Points for Maxima and Minima
disp('Stationary Points for Maxima and Minima:');
disp(stationary_points);
```

O/P:

```
Solution to Non-Linear Differential Equation:
3*exp(t) - 2
```

2 - exp(t)

Stationary Points for Maxima and Minima:

3/2
3/2 - 5^(1/2)/2
5^(1/2)/2 + 3/2

Ex 10.5:
Procedure:
Integration

1. **Compute Indefinite Integrals:**
   - Use int(function) to compute the indefinite integral of a function.
2. **Compute Definite Integrals:**
   - Use int(function, lower_limit, upper_limit) to compute the definite integral of a function over a specified range.
3. **Compute Area Under the Curve:**
   - Define the function and use int(function, lower_limit, upper_limit) to find the area under the curve over the specified range.
4. **Plot the Function (Optional):**
   - Use ezplot(function, [lower_limit, upper_limit]) to plot the function over the specified range.
5. **Display Results:**
   - Use disp() to display the results of the integrals and area under the curve.

Code:
INTEGRATION

```
syms x

% Integration Examples
f1 = 2 + x;
int_f1 = int(f1);

f2 = sin(x);
int_f2 = int(f2);

f3 = x^3 + 2*x^2 + 3*x - 16;
int_f3 = int(f3, 1, 2);

% Display Integration Results
disp('Indefinite Integral of 2 + x:');
```

```
disp(int_f1);

disp('Indefinite Integral of sin(x):');
disp(int_f2);

disp('Definite Integral of x^3 + 2*x^2 + 3*x - 16 from 1 to 2:');
disp(int_f3);

% Area Under the Curve Example
f4 = x^2*cos(x);
area_f4 = int(f4, -4, 9);

% Display Area Under the Curve Result
disp('Area under the curve of x^2*cos(x) from -4 to 9:');
disp(area_f4);

% Plot the function
ezplot(f4, [-4, 9]);
```

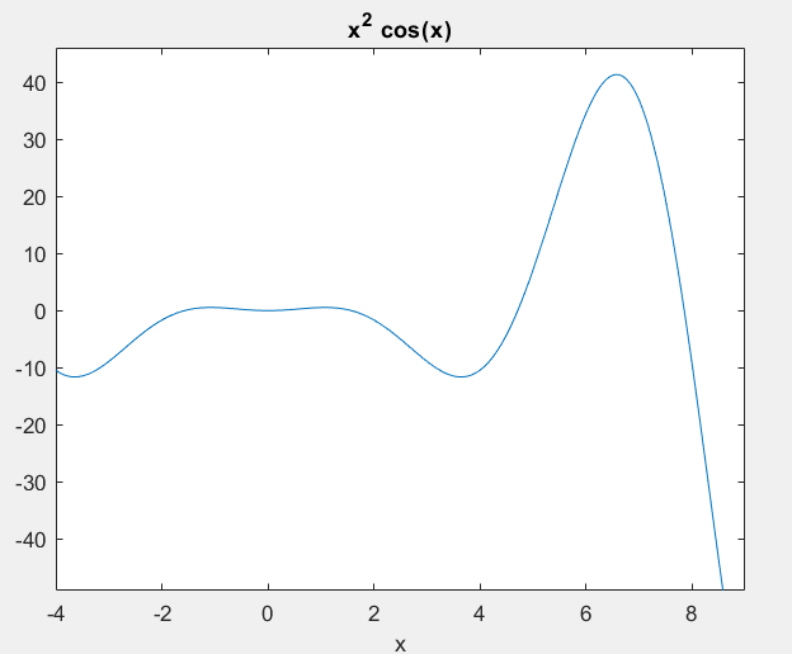
O/P:

Indefinite Integral of 2 + x:(x*(x + 4))/2

Indefinite Integral of sin(x):-cos(x)

Definite Integral of x^3 + 2*x^2 + 3*x - 16 from 1 to 2:-37/12

Area under the curve of x^2*cos(x) from -4 to 9:8



Ex 10.6:
Procedure:
Laplace Transform

1. **Compute Laplace Transform:**
    - Use laplace(function) to compute the Laplace transform of a function with respect to t.
2. **Compute Inverse Laplace Transform:**
    - Use ilaplace(expression, s, t) to compute the inverse Laplace transform of an expression with respect to s, returning a function in terms of t.
3. **Display Results:**
    - Use disp() to display the results of the Laplace and inverse Laplace transforms.

Code:
LAPLACE TRANSFORM

```
syms t s

% Laplace Transform Example
laplace_t_example = laplace(t^3);

% Display Laplace Transform Result
disp('Laplace Transform of t^3:');
disp(laplace_t_example);

% Inverse Laplace Transform Examples
ilaplace_x_example = ilaplace(cos(s), s, t);
ilaplace_s_example = ilaplace(1/s^3, s, t);

% Display Inverse Laplace Transform Results
disp('Inverse Laplace Transform of cos(s):');
disp(ilaplace_x_example);

disp('Inverse Laplace Transform of 1/s^3:');
disp(ilaplace_s_example);
```

O/P:

```
Laplace Transform of t^3:6/s^4

Inverse Laplace Transform of cos(s):ilaplace(cos(s), s, t)

Inverse Laplace Transform of 1/s^3:t^2/2
```

Ex 10.7:
Procedure:
Fourier Transform

1. **Compute Fourier Transform:**
    - Use fourier(function) to compute the Fourier transform of a function with respect to x.
2. **Compute Inverse Fourier Transform:**
    - Use ifourier(expression) to compute the inverse Fourier transform of an expression with respect to w.
3. **Display Results:**
    - Use disp() to display the results of the Fourier and inverse Fourier transforms.

Code:

FOURIER TRANSFORM

```
syms x w

% Fourier Transform Examples
fourier_x_example = fourier(cos(x));
fourier_x2_example = fourier(sin(x));

% Display Fourier Transform Results
disp('Fourier Transform of cos(x):');
disp(fourier_x_example);

disp('Fourier Transform of sin(x):');
disp(fourier_x2_example);

% Inverse Fourier Transform Examples
ifourier_x_example = ifourier(exp(-w^2));
ifourier_x2_example = ifourier(1/(1 + w^2));

% Display Inverse Fourier Transform Results
disp('Inverse Fourier Transform of exp(-w^2):');
disp(ifourier_x_example);

disp('Inverse Fourier Transform of 1/(1 + w^2):');
disp(ifourier_x2_example);
```

O/P:

```
Fourier Transform of cos(x):pi*(dirac(w - 1) + dirac(w + 1))
```

Fourier Transform of sin(x):-pi*(dirac(w - 1) - dirac(w + 1))*1i

Inverse Fourier Transform of exp(-w^2):exp(-x^2/4)/(2*pi^(1/2))

Inverse Fourier Transform of 1/(1 + w^2):exp(-abs(x))/2

Expt No. 11	REGRESSION AND INTERPOLATION	Date of Expt:

Ex 11.1:
Procedure:
Newton Divided Difference Method

1. **Define Variables:**
   - Set the number of data points, nnn.
   - Define the vectors x and fx for the data points and their corresponding function values.
2. **Initialize Divided Difference Table:**
   - Create an $(n+1) \times (n+1)$ matrix F and set the first column to fx.
3. **Compute Divided Differences:**
   - Use nested loops to fill in the matrix FFF using the divided difference formula:

$$F(i+1, j+1) = \frac{F(i+1, j) - F(i, j)}{x(i+1) - x(i-j+1)}$$

4. **Extract Coefficients:**
   - Extract the coefficients of the Newton polynomial from the diagonal of matrix F.
5. **Evaluate Newton Polynomial:**
   - Generate values for the variable (e.g., $plotx$).
   - Compute the corresponding values using the Newton polynomial formula with the coefficients obtained.
6. **Plot Results:**
   - Plot the Newton polynomial and the original data points to visualize the interpolation.

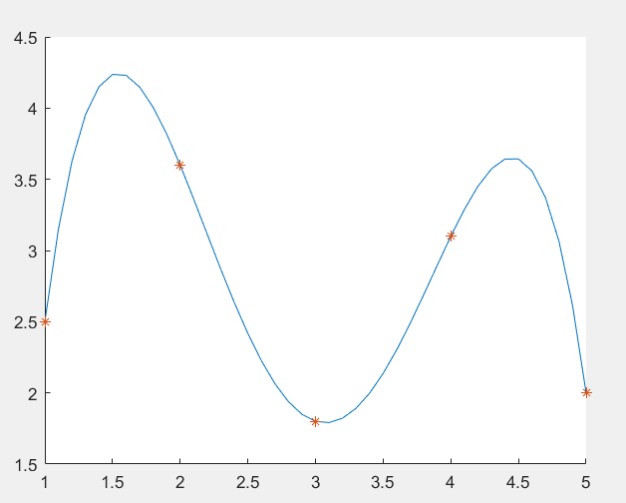
Code:
NEWTON DIVIDED DIFFERENCE METHOD

```
n=4;
x=[1;2;3;4;5];
fx=[2.5;3.6;1.8;3.1;2.0];
F=zeros(n+1, n+1);
F(:,1)=fx;
% Compute the Newton divided differences.
for i=1:n
```

```
    for j=1:i
        F(i+1,j+1)=(F(i+1,j)-F(i,j))/(x(i+1)-x(i-j+1));
    end
end
a=diag(F)
hold on
plotx=1:0.1:5;
ploty = a(n+1)*ones(size(plotx));
for i=n:(-1):1
    ploty = a(i) + ploty.*(plotx-x(i));
end
figure(1);
plot(plotx,ploty,'-',x,fx,'*');
```

O/P:

```
a =
    2.5000
    1.1000
   -1.4500
    1.0000
   -0.4792
```



Ex 11.2:

Procedure:

Regression Method

1. **Define Variables:**
    - Set up the vectors X and Y for the independent and dependent variables.
    - Determine the number of data points, n.

2. **Construct the Matrix A and Vector b:**
   - Matrix A contains sums related to X and X^2:

$$A = \begin{bmatrix} n & \sum X \\ \sum X & \sum X^2 \end{bmatrix}$$

   - Vector b contains sums related to Y and $X \cdot Y$:

$$b = \begin{bmatrix} \sum X \\ \sum (X \cdot Y) \end{bmatrix}$$

3. **Compute the Coefficients:**
   - Solve for the coefficients φ by computing:

$$\varphi = inv(A) \cdot b$$

   - φ will contain the intercept and slope of the regression line.
4. **Plot Results:**
   - Plot the original data points (X, Y) as blue squares.
   - Plot the regression line using the computed coefficients φ in red.

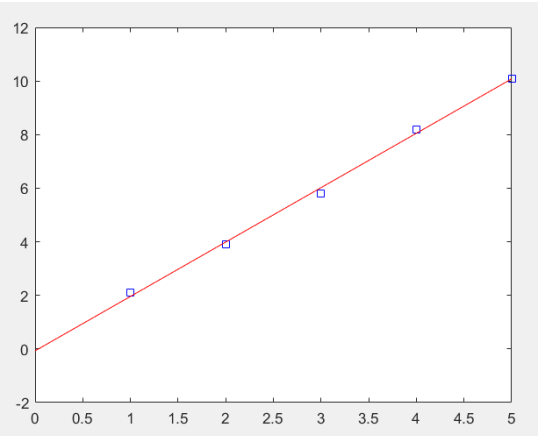
This method fits a linear regression model to the given data.

Code:

REGRESSION METHOD

```
X = [1; 2; 3; 4; 5];

Y = [2.1; 3.9; 5.8; 8.2; 10.1];
n = length(X);
A = [n, sum(X); sum(X), sum(X.*X)];
b = [sum(Y); sum(X.*Y)];
phi = inv(A)*b;
plot(X, Y, 'bs', [0 5], phi(1) + phi(2)*[0 5], '-r');
```

O/P:



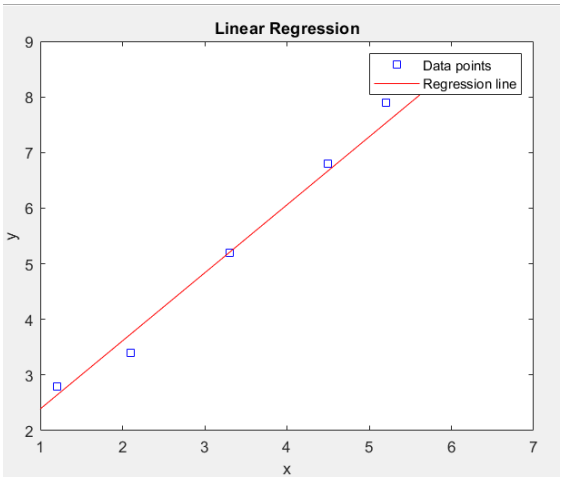
Ex 11.3:
Procedure:
Multi-Regression Method

1. **Initialize Data**:
    - Define the independent variable and dependent variable datasets.
2. **Set Up the Regression Matrix**:
    - Construct a matrix that includes a column of ones (to account for the intercept) and the independent variable data.
3. **Compute Regression Coefficients**:
    - Use the ordinary least squares method to calculate the regression coefficients.
4. **Plot Data and Regression Line**:
    - Create a plot showing the original data points and overlay the regression line.
5. **Output the Coefficients**:
    - Display the calculated regression coefficients.

Code:
MULTI-REGRESSION METHOD

```
% Data

x = [1.2; 2.1; 3.3; 4.5; 5.2; 6.1];
y = [2.8; 3.4; 5.2; 6.8; 7.9; 8.3];
n = length(x);

% Regression setup
X = [ones(n,1), x];
Y = y;

% Compute regression coefficients
phi = inv(X'*X)*X'*Y;
```

```
% Plot
plot(x, y, 'bs', [1 6], phi(1) + phi(2)*[1 6], '-r');
title('Linear Regression');
xlabel('x');
ylabel('y');
legend('Data points', 'Regression line');

% Output
phi
```



O/P:

```
phi =
   1.1670
   1.2231
```

Ex 11.4:

Procedure:

Spline and PCHIP Interpolation

1. **Initialize Data**:
   - Define the time vector T and corresponding values vector S.
2. **Plot Original Data**:
   - Plot the original data points using a specified line style and color.
3. **Perform Spline Interpolation**:
   - Compute spline interpolation values for a finer time vector.
   - Plot the spline interpolation results using a different marker and color.
4. **Perform PCHIP Interpolation**:
   - Compute PCHIP (Piecewise Cubic Hermite Interpolating Polynomial) interpolation values for the same finer time vector.
   - Plot the PCHIP interpolation results using a different line style and marker.
5. **Enhance the Plot**:
   - Add title, axis labels, and a legend to distinguish between the original data, spline interpolation, and PCHIP interpolation.

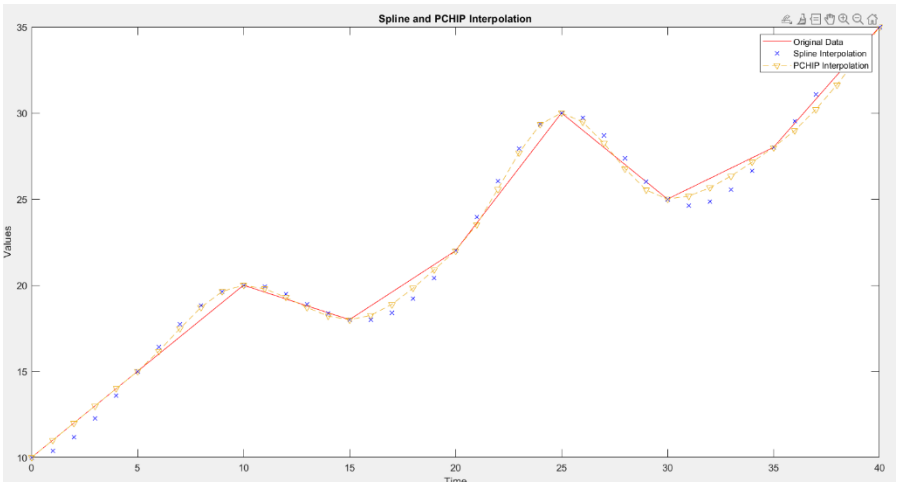
Code:

SPLINE AND PCHIP INTERPOLATION

```
% Data
T = 0:5:40;
```

```
S = [10, 15, 20, 18, 22, 30, 25, 28, 35];

% Plot original data
plot(T, S, '-r');
hold on;

% Spline Interpolation
TI = 0:40;
SI = spline(T, S, TI);
plot(TI, SI, 'xb');
% PCHIP Interpolation
TIP = 0:40;
SI_P = pchip(T, S, TIP);
plot(TI, SI_P, '--v');

% Enhance the plot
title('Spline and PCHIP Interpolation');
xlabel('Time');
ylabel('Values');
legend('Original Data', 'Spline Interpolation', 'PCHIP Interpolation');
```

O/P:



Expt No. 12	IMAGE PROCESSING	Date of Expt:

The **Image Processing Toolbox** in MATLAB is a powerful collection of functions and tools designed to assist with a wide range of image processing tasks. It provides an extensive set of algorithms and workflows for processing, analyzing, visualizing, and algorithm development in the field of image and video data. Here are some of its key features:

1. Image Importing and Exporting:

- Supports various image formats such as JPEG, PNG, TIFF, BMP, and others.
- Allows for importing data from specialized formats like DICOM (for medical imaging) and GeoTIFF (for geospatial data).

2. Image Enhancement:

- Functions for improving image quality, such as contrast enhancement, histogram equalization, noise reduction, and filtering.
- Includes advanced methods like adaptive histogram equalization and Wiener filtering.

3. Geometric Transformations:

- Enables transformations like scaling, rotating, cropping, and translating images.
- Tools for image registration, aligning different images for comparison, and combining data from multiple sources.

4. Filtering and Convolution:

- Offers a variety of linear and nonlinear filtering options, such as Gaussian, median, and Sobel filters.
- Convolution operations to enhance or detect features in an image.

5. Segmentation:

- Tools for dividing an image into meaningful regions, such as thresholding, edge detection, and watershed segmentation.
- Advanced segmentation techniques include active contours (snakes) and region-growing methods.

6. Morphological Operations:

- Functions for image morphology such as dilation, erosion, opening, closing, and skeletonization, useful for analyzing shapes and structures within an image.

7. Feature Detection and Extraction:

- Supports feature detection methods such as edge detection, corner detection, and blob analysis.
- Functions to find shapes, objects, and boundaries in images, including the Hough transform and region properties analysis.

8. Image Registration:

- Tools for aligning images taken at different times or from different perspectives.
- Techniques include intensity-based, feature-based, and multimodal registration.

9. Object Detection and Measurement:

- Supports measuring and analyzing objects within an image, including area, centroid, perimeter, and shape characteristics.
- Tools for object detection, classification, and tracking in sequences of images or videos.

10. 3D Image Processing:

- Capabilities to handle 3D image volumes, such as medical scans (MRI or CT) and microscopy data.
- Functions for visualizing, processing, and analyzing volumetric data.

11. Image Annotation and Visualization:

- Tools for marking images with text, shapes, and lines.
- Visualization tools that can overlay images, display histograms, and generate 3D plots.

12. GPU Acceleration:

- Many functions in the toolbox are optimized to leverage GPU acceleration, making it easier to process large images or perform computations faster.

13. Machine Learning and Deep Learning Integration:

- Supports integrating image processing with machine learning and deep learning workflows.
- Includes pretrained networks for tasks such as image classification, object detection, and semantic segmentation.

Ex 12. 1: Reading and Displaying Images

- **Objective**: Learn how to read and display images in MATLAB.
- **Procedure**:
    1. Read an image using imread.
       A = imread('nature.jpg');
    2. Display the image using imshow.
       imshow(A);
    3. Check the size of the image.
       size(A);
    4. Open the image in a separate figure.
       figure; imshow(A);

Ex 12.2: Image Resizing

- **Objective**: Resize an image and visualize the results.
- **Procedure**:

1. Resize the image to twice its size.
   B = imresize(A, 2);
2. Display the original and resized images.
   subplot(1, 2, 1); imshow(A); title('Original Image');
   subplot(1, 2, 2); imshow(B); title('Resized Image');

Ex 12. 3: Image Rotation

- **Objective**: Rotate an image by 45 degrees.
- **Procedure**:
  1. Rotate the image by 45 degrees.
     C = imrotate(A, 45);
  2. Display the rotated image.
     imshow(C);

Ex 12.4: Grayscale Conversion

- **Objective**: Convert a color image to grayscale.
- **Procedure**:
  1. Convert the image to grayscale.
     gray_A = rgb2gray(A);
  2. Display the grayscale image.
     imshow(gray_A);

Ex 12. 5: Image Histogram and Histogram Equalization

- **Objective**: Equalize the histogram of an image to enhance contrast.
- **Procedure**:
  1. Convert the image to grayscale.
     gray_A = rgb2gray(A);
  2. Apply histogram equalization.
     hist_A = histeq(gray_A);
  3. Display the original and histogram-equalized images along with their histograms.
     subplot(2, 2, 1); imshow(gray_A); title('Original Image');
     subplot(2, 2, 2); imshow(hist_A); title('Histogram Equalized Image');
     subplot(2, 2, 3); imhist(gray_A); title('Original Histogram');

subplot(2, 2, 4); imhist(hist_A); title('Equalized Histogram');

Ex 12. 6: Filtering: Gaussian and Median

- **Objective**: Apply different filters to remove noise.
- **Procedure**:
    1. Convert the image to grayscale.
       gray_A = rgb2gray(A);
    2. Apply Gaussian and Median filters.
       h_gaussian = fspecial('gaussian', 3, 0.5);
       A_gaussian = imfilter(gray_A, h_gaussian);
       A_median = medfilt2(gray_A);
    3. Display the original, Gaussian, and median filtered images.
       subplot(1, 3, 1); imshow(gray_A); title('Original');
       subplot(1, 3, 2); imshow(A_gaussian); title('Gaussian Filter');
       subplot(1, 3, 3); imshow(A_median); title('Median Filter');

Ex 12.7: Edge Detection (Sobel, Prewitt, Canny)

- **Objective**: Detect edges using different operators.
- **Procedure**:
    1. Convert the image to grayscale.
       gray_A = rgb2gray(A);
    2. Apply Sobel, Prewitt, and Canny edge detection.
       A_sobel = edge(gray_A, 'sobel');
       A_prewitt = edge(gray_A, 'prewitt');
       A_canny = edge(gray_A, 'canny');
    3. Display all results in a subplot.
       subplot(2, 2, 1); imshow(A); title('Original Image');
       subplot(2, 2, 2); imshow(A_sobel); title('Sobel');
       subplot(2, 2, 3); imshow(A_prewitt); title('Prewitt');
       subplot(2, 2, 4); imshow(A_canny); title('Canny');

Ex 12. 8: Morphological Operations (Erosion and Dilation)

- **Objective**: Perform erosion and dilation operations.
- **Procedure**:
    1. Convert the image to grayscale.
       gray_A = rgb2gray(A);

2. Define a structuring element.
   se = strel('disk', 5);

3. Apply erosion and dilation.
   A_eroded = imerode(gray_A, se);
   A_dilated = imdilate(gray_A, se);

4. Display the original, eroded, and dilated images.
   subplot(1, 3, 1); imshow(gray_A); title('Original');
   subplot(1, 3, 2); imshow(A_eroded); title('Eroded');
   subplot(1, 3, 3); imshow(A_dilated); title('Dilated');

Ex 12.9: Image Thresholding

- **Objective**: Convert an image to binary using thresholding.
- **Procedure**:
  1. Convert the image to grayscale and double format.
     gray_A = im2double(rgb2gray(A));
  2. Apply different threshold levels.
     B_thresh100 = im2bw(gray_A, 100/255);
  3. Display the original and thresholded images.
     subplot(1, 2, 1); imshow(gray_A); title('Original Image');
     subplot(1, 2, 2); imshow(B_thresh100); title('Thresholded Image');

Ex 12.10: Noise Addition and Removal

- **Objective**: Add different types of noise and remove them using filters.
- **Procedure**:
  1. Add Gaussian and salt & pepper noise.
     B_gaussian = imnoise(A, 'gaussian');
     B_saltpepper = imnoise(A, 'salt & pepper');
  2. Apply a median filter to remove noise.
     B_filtered = medfilt2(rgb2gray(B_saltpepper));
  3. Display the noisy and filtered images.
     subplot(1, 2, 1); imshow(B_saltpepper); title('Salt & Pepper Noise');
     subplot(1, 2, 2); imshow(B_filtered); title('Filtered Image');

Ex 12.11: Slope Calculation Using DEM

- **Objective**: Calculate the slope from a Digital Elevation Model (DEM).
- **Procedure**:
    1. Load the DEM file (ensure you have a DEM image or .tif file).
       DEM = imread('dem.tif');
       DEM = double(DEM); % Convert to double for calculation
    2. Calculate the gradient in the x and y directions using the gradient function.
       [Gx, Gy] = gradient(DEM);
    3. Compute the slope in degrees.
       slope = atan(sqrt(Gx.^2 + Gy.^2)) * (180/pi); % Convert radians to degrees
    4. Display the slope map.
       imagesc(slope);
       colorbar;
       title('Slope Map');

Ex 12.12: Slope and Aspect Calculation

- **Objective**: Compute both the slope and aspect from a DEM.
- **Procedure**:
    1. Load the DEM and compute its gradients.
       DEM = imread('dem.tif');
       DEM = double(DEM);
       [Gx, Gy] = gradient(DEM);
    2. Calculate the slope.
       slope = atan(sqrt(Gx.^2 + Gy.^2)) * (180/pi);
    3. Calculate the aspect (direction of the steepest slope).
       aspect = atan2(Gy, Gx) * (180/pi); % Convert to degrees
       aspect(aspect < 0) = aspect(aspect < 0) + 360; % Convert to 0-360 degrees
    4. Display both slope and aspect maps.
       subplot(1, 2, 1);
       imagesc(slope); colorbar; title('Slope Map');
       subplot(1, 2, 2);
       imagesc(aspect); colorbar; title('Aspect Map');

Ex 12.13: Slope Generation Using 3D Surface

- **Objective**: Generate a synthetic 3D surface and calculate its slope.
- **Procedure**:
    1. Create a synthetic surface using a function (e.g., a Gaussian surface).
       ```
       [X, Y] = meshgrid(-5:0.1:5, -5:0.1:5);
       Z = exp(-X.^2 - Y.^2);  % Gaussian surface
       ```
    2. Compute the slope of the surface.
       ```
       [Gx, Gy] = gradient(Z);
       slope = atan(sqrt(Gx.^2 + Gy.^2)) * (180/pi);  % Slope in degrees
       ```
    3. Display the 3D surface and its slope map.
       ```
       figure;
       subplot(1, 2, 1);
       surf(X, Y, Z); title('3D Surface');
       subplot(1, 2, 2);
       imagesc(slope); colorbar; title('Slope Map');
       ```

Ex 12.14: Slope Classification Based on Terrain Types

- **Objective**: Classify slopes into different categories (e.g., gentle, moderate, steep).
- **Procedure**:
    1. Calculate slope from DEM as in previous experiments.
    2. Define slope categories (e.g., gentle: 0-10°, moderate: 10-30°, steep: >30°).
       ```
       gentle = slope < 10;
       moderate = slope >= 10 & slope <= 30;
       steep = slope > 30;
       ```
    3. Create a classified slope map.
       ```
       slope_class = zeros(size(slope));
       slope_class(gentle) = 1;  % Gentle
       slope_class(moderate) = 2;  % Moderate
       slope_class(steep) = 3;  % Steep
       imagesc(slope_class);
       colormap([0.8 1 0.8; 1 1 0.5; 1 0.5 0.5]);  % Color coding
       colorbar; title('Classified Slope Map');
       ```