RAJAGIRI SCHOOL OF ENGINEERING & TECHNOLOGY

RAJAGIRI VALLEY, KAKKANAD, COCHIN-682039

LAB RECORD

101009/IT700B IT WORKSHOP – MATLAB

SEVENTH SEMESTER

Submitted By

Noel Mathen Eldho (U2109053)

(2021-2025 Batch)

Department of
Computer Science and Business Systems

Rajagiri School of Engineering & Technology (Autonomous)

(Parent University: APJ Abdul Kalam Technological University)

Rajagiri Valley, Kakkanad, Kochi, 682039

November 2024

RAJAGIRI SCHOOL OF ENGINEERING & TECHNOLOGY (AUTONOMOUS)

VISION

To evolve into a premier technological and research institution, molding eminent professionals with creative minds, innovative ideas and sound practical skill, and to shape a future where technology works for the enrichment of mankind.

MISSION

To impart state-of-the-art knowledge to individuals in various technological disciplines and to inculcate in them a high degree of social consciousness and human values, thereby enabling them to face the challenges of life with courage and conviction.

DEPARTMENT OF COMPUTER SCIENCE AND BUSINESS SYSTEMS

VISION

To evolve into a department of excellence in information technology by the creation and exchange of knowledge through leading-edge research, innovation and services, which will in turn contribute towards solving complex societal problems and thus building a peaceful and prosperous mankind.

MISSION

To impart high-quality technical education, research training, professionalism and strong ethical values in the young minds for ensuring their productive careers in industry and academia so as to work with a commitment to the betterment of mankind.

RAJAGIRI SCHOOL OF ENGINEERING & TECHNOLOGY

RAJAGIRI VALLEY, KAKKANAD, COCHIN-682039

CERTIFICATE

This is to certify that this is a bonafide record of the work done by ***Noel Mathen Eldho (U2109053),*** *in the IT WORKSHOP - MATLAB(101009/IT700B) laboratory during the semester S7 in partial fulfillment of the requirements of the degree of Bachelor of Technology (B. Tech.) in "Computer Science and Business Systems" during the academic year 2024-2025 at Rajagiri School of Engineering & Technology (RSET) (Autonomous), Rajagiri Valley, Kochi.*

Ms. Veena Rani Faculty in Charge Associate Professor	Dr.Nikhila T Bhuvan Lab in Charge Associate Professor	Dr.Divya James (HoD) Associate Professor
Dept. of CU RSET	Dept. of CU RSET	Dept. of CU RSET

2021-25 Batch-Lab Cycle

Course Outcomes

After the completion of the course the student will be able to

- CO 1: Understand the Building and managing workspace.
- CO 2: Define and compose matrices and sub matrices to solve linear equations.
- CO 3: Display axis labels and annotations, and testing programming functions.
- CO 4: Understand how to program M-file scripts, M- file functions, Input –output Arguments and program control flow operators, loops, flow structures.
- CO 5: Use the debugging process and debugging M-files.
- CO 6: Implement various image processing techniques.

Experiment 1. Programs using mathematical, relational expressions, and operators

i)Basic Mathematical Operations

Write a MATLAB script that:

- Defines two variables ‘a’ and ‘b’ with values 5 and 3, respectively.
- Computes and displays the sum, difference, product, and quotient of ‘a’ and ‘b’.
- Uses relational operators to compare ‘a’ and ‘b’ and displays the results of the comparisons (e.g., a>b, a==b, etc.).

ii)Complex Mathematical Expressions

Write a MATLAB script that:

- Defines three variables x=2, y=4, and z=6.
- Computes the value of the expression $f(x,y,z)=x^2+y^2-z^2+\sqrt{xy}$ and displays the result.

Experiment 2. Vectors and Matrices: Programs using array operations and matrix operations

i) Vector Operations

Write a MATLAB script that:

- Creates two vectors A=[1,2,3] and B=[4,5,6].
- Computes and displays the dot product and cross product of vectors ‘A’ and ‘B’.

ii) Matrix Operations

Write a MATLAB script that:

- Creates two matrices M1=[1,2;3,4] and M2=[5,6;7,8].
- Performs and displays the result of matrix multiplication M1×M2,inverse of M1,Transpose of M2,matrix concatenation, Determinant of M2

Experiment 3. Programs on input and output of values

i)User Input and Output

Write a MATLAB script that:

- Prompts the user to enter two numbers.
- Calculates the sum, difference, product, and quotient of the two numbers.
- Displays the results in a formatted manner
- Prompts the user to enter strings and try out the string functions for Concatenation, String Comparison, Substring Operations, Case Conversion, **String Length and Splitting,** Padding and Trimming, **Pattern Matching and Replacement**

ii)Matrix Input and Output

Write a MATLAB script that: (don't use loop to enter the values)

- Prompts the user to enter the elements of a 2x2 matrix.
- Displays the entered matrix and its transpose.

Experiment 4. Selection Statements: Experiments on if statements, with else and elseif clauses and switch statements

i)elseif clauses

Write a MATLAB script that:

- Prompts the user to enter a score (0-100).
- Uses if-else and elseif statements to classify the score into grades (A, B, C, D, F) and displays the corresponding grade.
- Prompts the user to enter the coefficients of the quadratic equation and calculate the roots

ii)Using Switch Statement

Write a MATLAB script that:

- Prompts the user to enter the choice (to calculate the area or perimeter)and the radius of the circle
- Uses a switch statement to perform the corresponding operation and displays the result.

Experiment 5. Loop Statements and Vectorizing Code: Programs based on counted (for) and conditional (while) loops

i)Summation using For Loop

Write a MATLAB script that:

- Computes the sum of the first 'n' natural numbers using a for loop, where 'n' is provided by the user.
- Displays the result as a matrix.

ii)Factorial Calculation using While Loop

Write a MATLAB script that:

- Computes the factorial of a number ‘n’ using a while loop, where ‘n’ is provided by the user.
- Displays the result.

iii)Vectorized Operations

Write a MATLAB script that:

- Creates a vector x=[1,2,3,…,10]
- Computes and displays the square of each element in the vector using vectorized operations.

Experiment 6. Programs on Built-in text manipulation functions and conversion between string and number types

i)Text Manipulation

Write a MATLAB script that:

- Prompts the user to enter a string.
- Converts the string to uppercase and lowercase.
- Displays the results.

ii)String to Number Conversion

Write a MATLAB script that:

- Prompts the user to enter a numerical string.
- Converts the string to a number.
- Performs an arithmetic operation (e.g., adds 10 to the number) and displays the result.

iii) Counting Vowels in a String

Write a MATLAB script that:

- Prompts the user to enter a string.
- Uses built-in functions to count the number of vowels (a, e, i, o, u) in the string.
- Displays the count of each vowel.

Experiment 7. Programs based on scripts and user-defined functions

i)Simple Script and Function

Write a MATLAB script that:

- To read the range
- Print all the prime numbers within a range.

Write a MATLAB script that:

- Reads a number
- Check of the number is a Pythagorean triplet using function

- *Hint:The set of numbers (3, 4, 5) is called a Pythagorean triple, meaning the three positive integers satisfy the equation:* $a^2+b^2=c^2$

ii)Function to Compute Fibonacci Sequence

Write a MATLAB script that:

- Defines a function fibonacci to compute the first 'n' terms of the Fibonacci sequence.
- Prompts the user to enter 'n', calls the function, and displays the sequence

Experiment 8. Programs based on Advanced Plotting Techniques

i)Subplots

Write a MATLAB script that:

- Creates a figure with two subplots, the first subplot, plots a sine wave, the second subplot, plots a cosine wave.
- Represent the sine and Cosine waves as two graphs in the same figure
- Adds titles, labels, and legends to the plots.

ii)3D Plotting

Write a MATLAB script that:

- Generates a mesh grid of 'x' and 'y' values.
- Computes $z=\sin(\sqrt{(x^2+y^2)})$.
- Creates a 3D surface plot, line and scatter plot of 'z'.
- Adds titles, labels, and color bar.

Write a MATLAB script that:

- Plots a sphere [x(t,s),y(t,s),z(t,s)]=[cos(t)cos(s),cos(t)sin(s),sin(t)] where [t,s]=[0,2pi]
- Adds titles, labels, and proper titles.

Experiment 9. Programs based on two main data structures: cell arrays and structures

i)Cell Array Manipulation

Write a MATLAB script that:

- Has the cell array C = {3.14, 'MATLAB', true; 7, [1 2 3], 'hello'}
- Access and display the second element of the first row.
- Replace the third element of the second row with a new matrix [4 5 6].
- Modify the content of the first cell to store a structure containing information about a student (name, age, GPA). Display the updated cell array.

ii)Structures

Write a MATLAB script that:

- Defines a structure array to store information about three students, including fields for Name, Age, and marks of 4 subjects.
- Adds data for each student.
- Computes and displays the average grade of each student along with name. Generate a rank list and store it in another structure and display it.

Experiment 10. Programs based on Advanced Functions

i)Anonymous Functions and Function Handles

Write a MATLAB script that:

- Defines an anonymous function to compute the square of a number.
- Creates a function handle to the anonymous function.
- Uses the function handle to compute and display the square of a user-provided number.

ii)Nested Functions

Write a MATLAB script that:

- Defines a main function to compute the area of a rectangle.
- Within the main function, defines a nested function to compute the perimeter of the rectangle.
- Prompts the user for the length and width of the rectangle.
- Call the nested function to compute the perimeter and the main function to compute the area, then display both results.

Experiment 11. Programs based on image processing

i)Basic Image Manipulation

Write a MATLAB script that:

- Read an image file.
- Converts the image to grayscale.
- Displays the original and grayscale images side by side.

ii)Edge Detection

Write a MATLAB script that:

- Read an image file.
- Converts the image to grayscale.
- Applies a different edge detection algorithm to the image.
- Displays the original image and all the edge-detected image side by side.

iii)Image scaling

Write a MATLAB script that:

- Read an image file.
- Converts the image to grayscale.
- Implement a custom function that processes the image pixel by pixel and computes the average of the neighboring pixels replacing each pixel of an image with the average value of its four nearest neighbors (top, bottom, left, and right).
- The border pixels need not be processed because they do not have four neighbors.

Table of Contents

Date: 05/08/2024

Experiment 1
Basic Mathematical Operations

Aim

i)Basic Mathematical Operations
Write a MATLAB script that:

- Defines two variables ‘a’ and ‘b’ with values 5 and 3, respectively.
- Computes and displays the sum, difference, product, and quotient of ‘a’ and ‘b’.
- Uses relational operators to compare ‘a’ and ‘b’ and displays the results of the comparisons (e.g., a>b, a==b, etc.).

ii) Complex Mathematical Expressions
Write a MATLAB script that:

- Defines three variables x=2, y=4, and z=6.
- Computes the value of the expression f(x,y,z)=x2+y2−z2+√xy and displays the result.

Theoretical Background

MATLAB (Matrix Laboratory) is a high-level programming language and interactive environment used for numerical computation, data analysis, visualization, and algorithm development.
Given below are some important commands:

1. `ans`
`ans` is a default variable used to store the result of a computation or command when no other variable is specified. It stands for "answer." For example, if you perform a calculation like `5 + 3` without assigning it to a variable, MATLAB stores the result in `ans`:

```
>> 5 + 3
ans =
        8
```

2. `diary`
record all the input and output of the MATLAB command window to a file. This is useful for keeping a log of your MATLAB session. It toggles both on and off.

3. Who
lists the names of defined variables
>>who
Your variables are:
a b

4.Whos
: lists the names and sizes of defined variables
>>whos a

Name	Size	Bytes	Class	Attributes
a	1x1	8	double	

5. Arithmetic Operators
MATLAB supports a range of arithmetic operators for performing mathematical operations:

i) Addition (`+`): Adds two numbers.

ii) Subtraction (`-`): Subtracts one number from another.

iii) Multiplication (`*`): Multiplies two numbers or matrices.

iv) Division (`/`): Divides one number by another. For element-wise division of matrices, use `./`.

v) Element-wise Multiplication (`.*`): Multiplies corresponding elements of two arrays or matrices.

vi) Element-wise Division (`./`)**: Divides corresponding elements of two arrays or matrices.
>>g = [8 16] ./ [2 4]
g =
[4 4]

vii) Power (`^`)**: Raises a number to the power of another. For element-wise power, use `.^`.
>>h = 2 ^ 3

```
    h =
          8
>>i = [2 3] .^ 2
    i =
          [4 9]
```

viii) Square Root (`sqrt`): Computes the square root of a number or each element of an array.

```
>> j = sqrt(9);
    j =
          3
>>k = sqrt([4 16])
    k =
          [2 4]
```

6. Relational Operators

i) Relational operators are used to compare values and return a logical result (`true` or `false`):

ii) Not equal to (`~=`)**: Checks if two values are not equal.

iii) Greater than (`>`): Checks if one value is greater than another.

iv) Less than (`<`): Checks if one value is less than another.

v) Greater than or equal to (`>=`): Checks if one value is greater than or equal to another.

vi) Less than or equal to (`<=`): Checks if one value is less than or equal to another. c = (4 <= 5);

Code

i)

```
a=5;
   b=3;
   a+b

ans =

   8
```

```
   a-b

ans =

   2

   a*b
```

```
ans =
   15
   a/b
ans =
   1.6667
x = (5 == 5)

x =
 logical
  1
   y = (4 ~= 5)
y =
 logical
  1
    z = (7 > 5)
z =
 logical
  1
   a = (3 < 5)
a =
 logical
  1
    b = (5 >= 5)
b =
 logical
  1
    c = (4 <= 5)
c =
 logical
  1
   x=(5==4)
x =
 logical
     0
```

ii)

```
>>x=2;
>>y=4;
>>z=6;
>>f(x,y,z)=x^2 + y^2 + z^2 + sqrt(x*y);
>>f(x,y,z)
        ans =
                58.8284
```

Conclusion

The basic commands like whos,who and diary have been completed and basic arithmetic and relational expressions have also been completed .

Date: 12/08/2024

Experiment 2
Vectors and Matrices

Aim

i) **Vector Operations**

Write a MATLAB script that:

- Creates two vectors A=[1,2,3] and B=[4,5,6].
- Computes and displays the dot product and cross product of vectors 'A' and 'B'.

ii) **Matrix Operations**

Write a MATLAB script that:

- Creates two matrices M1=[1,2;3,4] and M2=[5,6;7,8].
- Performs and displays the result of matrix multiplication M1×M2,inverse of M1,Transpose of M2,matrix concatenation, Determinant of M2

Theoretical Background

Matrix in MATLAB

A matrix in MATLAB is a two-dimensional array of numbers where each element is identified by its row and column index. MATLAB is particularly designed to handle matrices and matrix operations efficiently, making it a powerful tool for numerical computations.

Matrix Representation

- **Row Vector**: A 1xN matrix where N is the number of columns.
- **Column Vector**: An Nx1 matrix where N is the number of rows.
- **Square Matrix**: An NxN matrix with an equal number of rows and columns.
- **Diagonal Matrix**: A square matrix where only the diagonal elements (from the top left to the bottom right) are non-zero.
- **Identity Matrix**: A special diagonal matrix where all the diagonal elements are 1, denoted as I.

In MATLAB, matrices are defined using square brackets [ ], with semicolons ; separating the rows. For example:

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

This defines a 3x3 matrix A.

Difference Between Vector and Scalar

- **Scalar**: A single number, often representing magnitude or quantity, and is a 1x1 matrix in MATLAB. For example, a = 5;.
- **Vector**: A one-dimensional array of numbers. A vector can be either a row vector (1xN) or a column vector (Nx1) in MATLAB. For example:
    - Row Vector: v_row = [1 2 3];
    - Column Vector: v_col = [1; 2; 3];

Vector Operations in MATLAB

- **Addition and Subtraction**: Element-wise addition or subtraction of vectors of the same size.

```
>> v1 = [1 2 3];
   v2 = [4 5 6];
   result = v1 + v2;  % Result: [5 7 9]
```

- **Scalar Multiplication**: Multiplying each element of a vector by a scalar.

```
>> v = [1 2 3];
   scalar = 2;
   result = scalar * v;  % Result: [2 4 6]
```

- **Dot Product**: The sum of the products of corresponding elements of two vectors.

```
>> v1 = [1 2 3];
   v2 = [4 5 6];
   dot_product = dot(v1, v2);  % Result: 32
```

- **Cross Product**: A vector perpendicular to two vectors in three-dimensional space.

```
>> v1 = [1 2 3];
   v2 = [4 5 6];
   cross_product = cross(v1, v2);  % Result: [-3 6 -3]
```

Matrix Functions in MATLAB

- **Transpose (')**: Transposes a matrix, swapping its rows and columns.

```
>> A = [1 2 3; 4 5 6];
   A_transpose = A';  % Result: [1 4; 2 5; 3 6]
```

- **Inverse (inv)**: Computes the inverse of a square matrix.

```
>> A = [1 2; 3 4];
   A_inv = inv(A);
```

- **Determinant (det)**: Computes the determinant of a square matrix.
  >> A = [1 2; 3 4];
  determinant = det(A); % Result: -2

- **Eigenvalues and Eigenvectors (eig)**: Computes the eigenvalues and eigenvectors of a square matrix.
  >> A = [1 2; 2 1];
  [V, D] = eig(A); % V contains eigenvectors, D contains eigenvalues

- **Matrix Multiplication (*)**: Multiplies two matrices if their dimensions are compatible.
  >> A = [1 2; 3 4];
  B = [5 6; 7 8];
  C = A * B; % Result: [19 22; 43 50]

- **Element-wise Operations (.*, ./, .^)**: Perform element-wise multiplication, division, or exponentiation.
  >> A = [1 2 3];
  B = [4 5 6];
  C = A .* B; % Element-wise multiplication, Result: [4 10 18]

Types of Matrices

- **Zero Matrix:** A matrix filled entirely with zeros.
  zeromatrix = zeros(3)
  zeromatrix =
  0 0 0
  0 0 0
  0 0 0

- **Ones Matrix:** A matrix filled entirely with ones.
  onematrix = ones(3)
  onematrix =
  1 1 1
  1 1 1
  1 1 1

- **Size and Length of Matrix:** Size returns the dimensions of a matrix as [rows, columns] while Length returns the size of the largest dimension of the matrix..

```
size(onematrix)
ans =
    3    2

length(onematrix)
ans =
    3
```

- **Identity Matrix:** A square matrix with ones on the diagonal and zeros elsewhere.

```
identity = eye(3)
identity =
    1    0    0
    0    1    0
    0    0    1
```

- **Matrix Rank:** The number of linearly independent rows or columns in a matrix.

```
rank(newmatrix)
ans =
    2
```

- **Upper and Lower Triangular Matrices:** A matrix where all the elements below the main diagonal are zero is called Upper triangular matrix. A matrix where all the elements above the main diagonal are zero is called Lower triangular matrix

```
triu(dotprod)
ans =
    1    2    3
    0    4    6
    0    0    9

tril(dotprod)
ans =
    1    0    0
    2    4    0
    3    6    9
```

- **Diagonal Matrix:** A matrix with non-zero elements only on the main diagonal.
  diag([1 2 3])
  ans =
  1 0 0
  0 2 0
  0 0 3

- **Matrix Reshaping:** Changing the dimensions of a matrix without altering its elements.
  shape = 1:10;
  reshape(shape, [5,2])
  ans =
  1 6
  2 7
  3 8
  4 9
  5 10

Code

i)

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

dotproduct = A .* B
dotproduct =
   4    10    18

crossproduct = cross(A, B)
crossproduct =
  -3     6    -3
```

ii)

```
M1 = [1 2; 3 4];
M2 = [5 6; 7 8];

product = M1*M2
product =
  19    22
  43    50

inverse = inv(M1)
inverse =
  -2.0000    1.0000
   1.5000   -0.5000

transpose = M2'
transpose =
   5     7
   6     8
concat = [M1, M2]
concat =
   1     2     5     6
   3     4     7     8

determinant = det(M2)
determinant = -2.0000

diary off
```

Conclusion

This experiment demonstrates the creation and manipulation of various types of matrices in MATLAB, including basic operations on vectors and matrices, as well as specific matrix functions such as identity, diagonal, and triangular matrices. All operations were successfully executed.

Date: 19/08/2024

Experiment 3
Programs on input and output of values

Aim

i) **User Input and Output**

Write a MATLAB script that:

- Prompts the user to enter two numbers.
- Calculates the sum, difference, product, and quotient of the two numbers.
- Displays the results in a formatted manner
- Prompts the user to enter strings and try out the string functions for Concatenation, String Comparison, Substring Operations, Case Conversion, Padding and Trimming,

ii) **Matrix Input and Output**

Write a MATLAB script that: (don't use loop to enter the values)

- Prompts the user to enter the elements of a 2x2 matrix.
- Displays the entered matrix and its transpose.

Theoretical Background

Reading User Input

- *Reading Numeric Values*
  Use the input function to read numeric values from the user. By default, MATLAB treats input as a numeric value unless specified otherwise.
  ```
  >> % Read a scalar numeric value
      userNum = input('Enter a number: ');
  ```
- *Reading Matrices or Arrays*
  To read a matrix or array from the user, you can use the input function and ask the user to input data in MATLAB's matrix format.
  ```
  >> % Read a matrix from the user
     userMatrix = input('Enter a matrix (e.g., [1 2; 3 4]): ');
  ```
  The user needs to provide the matrix in the correct MATLAB syntax.
- *Displaying Numeric Values*
  Use disp or fprintf to display numeric values.
  ```
  >> % Display a number
     disp(['The number you entered is: ', num2str(userNum)]);
  >> % Using fprintf for formatted output
     fprintf('The number you entered is: %.2f\n', userNum);
  ```

- *Displaying Matrices or Arrays*
  Use disp to show matrices and arrays. You can also use fprintf for formatted output, though it's more common for matrices.
  >> % Display a matrix
  disp(userMatrix);
- *Reading User Input as Strings*
  In MATLAB, the input function is used to get input from the user. When reading strings specifically, the input function can take a second argument, 's', to indicate that the input should be treated as a string.
  >> userStr = input('Enter a string: ', 's');
  This line prompts the user to enter a string, and userStr will store the entered string.

Creating and Manipulating Strings

String and Character Arrays

MATLAB provides two primary ways to handle strings:

- **String Arrays**: These are part of MATLAB's string data type introduced in R2016b. They are created using the string function.
  >> str = string(123); % Converts the number 123 to the string "123"

- **Character Arrays**: These are the traditional way of handling strings in MATLAB and are created using the char function.
  >> chr = char("Hello, World!"); % Converts the string to a character array

Concatenation

- **strcat**: This function concatenates strings horizontally. Trailing whitespace is removed in the process.
  >> result = strcat('Hello', ' ', 'World'); % Result: 'Hello World'
- **strjoin**: This function joins elements of a string array into a single string with a specified delimiter.
  >> result = strjoin(["This", "is", "MATLAB"], " "); % Result: "This is MATLAB"

String Comparison

- **strcmp**: This function compares two strings for equality in a case-sensitive manner.
  >> isEqual = strcmp('Hello', 'hello'); % Result: false

- **strcmpi**: This function compares two strings for equality in a case-insensitive manner.
  >> isEqual = strcmpi('Hello', 'hello'); % Result: true

Substring Operations

- **strfind**: This function finds all occurrences of a substring within a string and returns their starting indices.
  >> indices = strfind('This is a test', 'is'); % Result: [3, 6]

- **extractBetween**: This function extracts a substring between two specified substrings.
  >> subStr = extractBetween('This is a test', 'This', 'test'); % Result: " is a "

- **strrep**: This function replaces all occurrences of a substring with another substring.
  >> newStr = strrep('This is a test', 'test', 'demo'); % Result: 'This is a demo'

Case Conversion

- **lower**: Converts all characters in a string to lowercase.
  >> lowerStr = lower('MATLAB'); % Result: 'matlab'

- **upper**: Converts all characters in a string to uppercase.
  >> upperStr = upper('matlab'); % Result: 'MATLAB'

String Length and Splitting

- **strlength**: Returns the number of characters in a string.
  >> len = strlength('MATLAB'); % Result: 6

- **split**: Splits a string into parts using a specified delimiter.
  >> parts = split('one,two,three', ','); % Result: ["one", "two", "three"]

Padding and Trimming

- **strtrim**: Removes leading and trailing whitespace from a string.
  >> trimmedStr = strtrim(' Hello, World! '); % Result: 'Hello, World!'

- **pad**: Adds leading or trailing spaces to a string to achieve a specified length.
  >> paddedStr = pad('Hello', 10); % Result: 'Hello '

String Conversion

- **num2str**: Converts numerical values to a character array (string).
  >> str = num2str(123.45); % Result: '123.45'

- **int2str**: Converts integer values to a character array.
  >> str = int2str(123); % Result: '123'

Pattern Matching and Replacement

- **regexp**: Performs regular expression matching, allowing complex pattern searches within strings.
  >> match = regexp('This is a test', '\w+', 'match'); % Result: {'This', 'is', 'a', 'test'}

- **regexprep**: Replaces substrings that match a regular expression with a specified replacement.
  >> result = regexprep('This is a test', '\s', '_'); % Result: 'This_is_a_test'

Code

i)

```
format("compact")

num1 = input('Enter first number: ');
Enter first number: 1
num2 = input('Enter the second number: ');
Enter the second number: 9

sum = num1 + num2;
diff = num2 - num1;
prod = num1 * num2;
quot = num2/num1;

fprintf('Sum: %f\nDifference: %f\nProduct: %f\nQuotient: %f\n', sum, diff, prod, quot);
Sum: 10.000000
Difference: 8.000000
Product: 9.000000
Quotient: 9.000000

str1 = input('Enter string 1: ', 's');
Enter string 1: hello
str2 = input('Enter string 2: ', 's');
Enter string 2: world

concat = strcat(str1, str2);
compare = strcmp(str1, str2);
substr = extractBetween(concat, 'l', 'r');
substra = extractAfter(concat, 'o');
substrb = extractBefore(concat, 'w');
internal.matlab.datatoolsservices.VariableUtils.saveWorkspace
ustr = upper(concat);
lstr = lower(ustr);
padded = pad(concat, 20);
trimmed = strtrim(padded);

fprintf('Concatenated string: "%s"\nString comparison: %s\nSubstring: %s\nSubstring after: %s\nSubstring before: %s\nUppercase: %s\nLowercase: %s\n', concat, mat2str(compare), substr{1}, substra, substrb, ustr, lstr);
Concatenated string: "helloworld"
String comparison: false
```

```
Substring: lowo
Substring after: world
Substring before: hello
Uppercase: HELLOWORLD
Lowercase: helloworld

fprintf('Size of padded string: %d\nSize of trimmed string: %d', strlength(padded), strlength(trimmed));
Size of padded string: 20
Size of trimmed string: 10
```

ii)

```
a11 = input('Enter element(1,1): ');
Enter element(1,1): 12
a12 = input('Enter element(1,2): ');
Enter element(1,2): 24
a21 = input('Enter element(2,1): ');
Enter element(2,1): 36
a22 = input('Enter element(2,2): ');
Enter element(2,2): 48

matrix = [a11 a12; a21 a22];
matrixt = matrix';

fprintf('Matrix inputted: ');
Matrix inputted: disp(matrix)
   12    24
   36    48
fprintf('Matrix transpose: ');
Matrix transpose: disp(matrixt)
   12    36
   24    48
```

Conclusion

In MATLAB, user input is handled flexibly using the input function for numeric values, strings, and matrices. Outputs are displayed using disp or fprintf for formatted results. This capability enables interactive and dynamic data handling in MATLAB scripts and functions.

Date: 02/09/2024

Experiment 4

Selection Statements: Experiments on if statements, with else and elseif clauses and switch statements

Aim

i) **Grade Classification**

Write a MATLAB script that:

- Prompts the user to enter a score (0-100).
- Uses if-else and elseif statements to classify the score into grades (A, B, C, D, F) and displays the corresponding grade.

ii) **Quadratic equation**

- Prompts the user to enter the coefficients of the quadratic equation and calculate the roots

iii) **Switch Statement**

Write a MATLAB script that:

- Prompts the user to enter the choice (to calculate the area or perimeter)and the radius of the circle
- Uses a switch statement to perform the corresponding operation and displays the result.

Theoretical Background

Selection statements are fundamental constructs in programming languages that allow for decision-making. They enable a program to execute different sections of code based on specific conditions. MATLAB, a high-level programming language used extensively in engineering and scientific computing, includes several types of selection statements for controlling the flow of execution in scripts and functions.

Types of Selection Statements in MATLAB

MATLAB supports the following primary types of selection statements:

1. if Statements
2. switch Statements

if Statements

- The if statement is a fundamental control structure that executes a block of code if a specified condition is true. MATLAB provides a versatile if statement syntax that can handle multiple conditions through elseif clauses and an optional else clause.
- Syntax:

```
if condition
    % Code to execute if the condition is true
elseif another_condition
    % Code to execute if the another_condition is true
else
    % Code to execute if none of the above conditions are true
end
```

- Example:

```
x = 10;
if x > 0
    disp('x is positive');
elseif x < 0
    disp('x is negative');
else
    disp('x is zero');
end
```

switch Statements

- The switch statement provides a way to handle multiple discrete cases based on the value of a single expression. This is particularly useful when a variable can take one of several possible values, and you want to execute different code for each possible value.
- Syntax:

```
switch expression
    case value1
        % Code to execute if expression equals value1
    case value2
        % Code to execute if expression equals value2
    otherwise
        % Code to execute if expression does not match any case
end
```

- Example:

```
day = 'Monday';
switch day
   case 'Monday'
      disp('Start of the work week');
   case 'Friday'
      disp('End of the work week');
   otherwise
      disp('Midweek day');
end
```

Usage and Applications

- Conditional Execution: Selection statements are used to execute code conditionally based on the value of variables, user input, or other factors.
- Branching Logic: They allow for the implementation of complex branching logic where different paths in the code are executed based on varying conditions.
- Error Handling: if statements are often used to check for errors or invalid inputs and to handle exceptions gracefully.

Code

i)

```
canContinue = true;
while canContinue
   score = input("Enter Score: ");
   if(score>=90 && score<=100)
      disp("Grade: A");
   elseif(score>=80 && score<90)
      disp("Grade: B");
   elseif(score>=70 && score<80)
      disp("Grade: C");
   elseif(score>=60 && score<70)
      disp("Grade: D");
   elseif(score>=50 && score<60)
      disp("Grade: E");
   else
      disp("Grade: F");
   end
     loop = input("Do you want to add more scores?(Y/N) ","s");
   if(loop=="Y")
      canContinue = true;
   else
      canContinue = false;
   end
end
```

```
Enter Score: 95
Grade: A
Do you want to add more scores?(Y/N)
Y
Enter Score: 85
Grade: B
Do you want to add more scores?(Y/N)
Y
Enter Score: 75
Grade: C
Do you want to add more scores?(Y/N)
Y
Enter Score: 65
Grade: D
Do you want to add more scores?(Y/N)
Y
Enter Score: 55
Grade: E
```

```
Do you want to add more scores?(Y/N)
Y
Enter Score: 45
Grade: F
Do you want to add more scores?(Y/N)
Y
Enter Score: 25
Grade: F
Do you want to add more scores?(Y/N)
N
```

ii)

```
canContinue = true;
while canContinue
   a = input("Enter coefficient of x^2: ")
   b = input("Enter coefficient of x: ")
   c = input("Enter constant value: ")
   if(a==0)
         disp("The entered values do not
form a quadratic equation!")
   else
              fprintf('The given quadratic
equation is %dx^2 + %dx + %d\n', a, b,
c);
      d = b^2 - 4*a*c;
      if d > 0
         root1 = (-b + sqrt(d)) / (2*a);
         root2 = (-b - sqrt(d)) / (2*a);
             fprintf('The roots are real and
distinct: %.2f and %.2f\n', root1, root2);
      elseif d == 0
         root1 = -b / (2*a);
              fprintf('The root is real and
equal: %.2f\n', root1);
      else
         realPart = -b / (2*a);
         imagPart = sqrt(-d) / (2*a);
           fprintf('The roots are imaginary:
%.2f + %.2fi and %.2f - %.2fi\n',
realPart, imagPart, realPart, imagPart);
      end
   end
       loop = input("Do you want to
continue?(Y/N) ","s");
   if(loop=="Y")
      canContinue = true;
   else
   canContinue = false;
      end
```

```
end

Enter coefficient of x^2: 1
a =
    1
Enter coefficient of x: -5
b =
   -5
Enter constant value: 6
c =
    6
The given quadratic equation is 1x^2 +
-5x + 6
The roots are real and distinct: 3.00 and
2.00
Do you want to continue?(Y/N) Y
Enter coefficient of x^2: 1
a =
    1
Enter coefficient of x: -4
b =
   -4
Enter constant value: 4
c =
    4
The given quadratic equation is 1x^2 +
-4x + 4
The root is real and equal: 2.00
Do you want to continue?(Y/N) Y
Enter coefficient of x^2: 1
a =
    1
Enter coefficient of x: 1
b =
    1
Enter constant value: 1
c =
    1
The given quadratic equation is 1x^2 +
1x + 1
The roots are imaginary: -0.50 + 0.87i
and -0.50 - 0.87i
Do you want to continue?(Y/N) N
```

iii)

```
canContinue = true;
while canContinue
   r = input('Enter radius of circle: ');
   disp('Operation:');
```

```
   disp('1. Perimeter of circle');
   disp('2. Area of circle');
     choice = input('Enter your choice of
operation: ');
   switch choice
      case 1
         p = 2 * pi * r;
          fprintf('Perimeter of the circle is:
%.2f\n', p);
      case 2
         A = pi * r^2;

fprintf('Area of the circle is: %.2f\n', A);
      otherwise
              fprintf('Invalid choice. Please
enter 1 or 2.\n');
   end

loop = input('Do you want to continue?
(Y/N): ', 's');
   if strcmpi(loop, 'Y')
      canContinue = true;
 else
      canContinue = false;
   end
end
```

```
Enter radius of circle: 3
Operation:
1. Perimeter of circle
2. Area of circle
Enter your choice of operation: 1
Perimeter of the circle is: 18.85
Do you want to continue? (Y/N): Y
Enter radius of circle: 3
Operation:
1. Perimeter of circle
2. Area of circle
Enter your choice of operation: 2
Area of the circle is: 28.27
Do you want to continue? (Y/N): N
```

Conclusion

Selection statements in MATLAB provide essential control over program flow, enabling developers to write flexible and adaptive code. The if and switch statements offer powerful mechanisms for decision-making, making it possible to execute different code blocks based on variable values or conditions. Understanding and utilizing these constructs effectively is crucial for writing robust MATLAB programs.

Date: 02/09/2024

Experiment 5

Programs based on counted (for) and conditional (while) loops

Aim

i) **Summation using For Loop**

Write a MATLAB script that:

- Computes the sum of the first 'n' natural numbers using a for loop, where 'n' is provided by the user.
- Displays the result

ii) **Factorial Calculation using While Loop**

Write a MATLAB script that:

- Computes the factorial of number 'n' using a while loop, where 'n' is provided by the user.
- Displays the result.

iii) **Switch Statement**

Write a MATLAB script that:

- Prompts the user to enter the choice (to calculate the area or perimeter)and the radius of the circle
- Uses a switch statement to perform the corresponding operation and displays the result.

Theoretical Background

Loops are essential programming constructs that allow for the repeated execution of code blocks. They are fundamental for performing repetitive tasks, iterating over arrays or matrices, and automating processes. MATLAB, a high-level programming language primarily used for numerical computing, supports several types of loops to handle different iterative tasks.

Types of Loops in MATLAB

MATLAB supports the following primary loop constructs:

1. for Loop
    - The for loop in MATLAB is used to iterate over a range of values or elements in an array. It executes a block of code a specific number of times, with each iteration using a different value from the specified range.
    - Syntax:

```
for index = startValue:endValue
    % Code to execute in each iteration
end
```

index: The loop variable that takes on each value in the specified range.
startValue: The initial value of the loop variable.
endValue: The final value of the loop variable.

- Example:

```
for i = 1:5
    disp(['Iteration number: ', num2str(i)]);
end
```

This loop will display the iteration number from 1 to 5.

2. while Loop
   - The while loop in MATLAB executes a block of code as long as a specified condition remains true. It is more flexible than the for loop as it allows for conditions that are not necessarily based on a fixed range.
   - Syntax:

```
while condition
    % Code to execute as long as the condition is true
end
```

condition: A logical expression that determines whether the loop body should execute.

   - Example:

```
count = 1;
while count <= 5
    disp(['Iteration number: ', num2str(count)]);
    count = count + 1; % Update the loop variable
end
```

This loop will also display the iteration number from 1 to 5, but it uses a condition to control the loop.

Practical Applications of Loops

- Repetitive Calculations: Loops are used to perform repetitive calculations or operations, such as summing elements or computing factorials.
- Iterating Over Arrays: Loops facilitate processing each element in arrays or matrices, allowing for operations like filtering or transformation.
- Simulations and Modeling: In simulations, loops can run through multiple iterations or time steps to model dynamic systems.

Performance Considerations

- Vectorization: In MATLAB, operations that can be vectorized (i.e., performed on entire arrays at once) are usually preferred over loops for performance reasons. MATLAB is optimized for matrix operations, and vectorized code is often more efficient.
- Loop Efficiency: When using loops, especially with large datasets, ensure that the code inside the loop is optimized, and consider preallocating arrays to improve performance.

Code

i)

```
n = input("Enter number: ");
sum = 0;
for i=1:n
sum = sum + i;
end
fprintf('Sum of the first %d numbers is %d\n', n, sum);
```

```
Enter number:
8
Sum of the first 8 numbers is 36
```

ii)

```
n = input("Enter number: ");
factorial = 1;
if(n == 1 || n == 0)
   factorial = 1;
else
   for i=1:n
      factorial = factorial * i;
   end
end
fprintf('Factorial of the %d is %d\n', n, factorial);
```

```
Enter number:
5
Factorial of the 5 is 120
```

iii)

```
v = [1 2 3 4 5 6 7 8 9 10]
squares = zeros(1, length(v));
for i = 1:length(v)
   squares(i) = v(i)^2;
end
disp("Square of each element in the vector is: ");
disp(squares);
```

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

Square of each element in the vector is:
     1     4     9    16    25    36    49    64
81   100
```

Conclusion

Loops are a crucial component of MATLAB programming, enabling the execution of repetitive tasks and complex iterative processes. Understanding how to effectively use for and while loops, and knowing when to leverage vectorized operations, is essential for efficient and effective programming in MATLAB.

Date: 30/09/2024

Experiment 6

Programs on Built-in text manipulation functions and conversion between string and number types

Aim

i) **Text Manipulation**

Write a MATLAB script that:

- Prompts the user to enter a string.
- Converts the string to uppercase and lowercase.
- Displays the results.

ii) **String to Number Conversion**

Write a MATLAB script that:

- Prompts the user to enter a numerical string.
- Converts the string to a number.
- Performs an arithmetic operation (e.g., adds 10 to the number) and displays the result.

iii) **Counting Vowels in a String**

Write a MATLAB script that:

- Prompts the user to enter a string.
- Uses built-in functions to count the number of vowels (a, e, i, o, u) in the string.
- Displays the count of each vowel.

Theoretical Background

MATLAB (Matrix Laboratory) is a high-level programming language and environment primarily designed for numerical and matrix computations. One of its strengths lies in its extensive library of built-in functions, which facilitate a wide range of tasks. Understanding these functions is crucial for effective programming and data manipulation in MATLAB.

Mathematical Functions

MATLAB includes numerous mathematical functions that perform basic to advanced computations:

- Basic Math Functions: Functions like abs(), sqrt(), exp(), and logarithmic functions (log(), log10()) provide essential mathematical operations for scalar and vector calculations.
- Trigonometric Functions: Functions such as sin(), cos(), and tan() allow users to perform trigonometric calculations, crucial for engineering and physics applications.
- Matrix Operations: Functions like det() for determinants, inv() for inverses, and eig() for eigenvalues are fundamental in linear algebra, enabling users to manipulate and analyze matrices efficiently.

Statistical Functions

MATLAB provides a suite of functions for statistical analysis, including:

- Descriptive Statistics: Functions like mean(), median(), and std() help summarize data sets, while var() computes variance.
- Statistical Testing: Functions such as ttest() and anova1() enable hypothesis testing and analysis of variance, essential for data-driven decision-making.

Logical and Relational Functions

Logical operations are vital in programming for control flow and data evaluation:

- Logical Functions: Functions like all(), any(), and not() allow for evaluating conditions across arrays, aiding in decision-making processes.
- Relational Functions: Functions such as eq(), lt(), and gt() facilitate comparisons between elements, useful for filtering and conditional logic.

File I/O Functions

MATLAB provides robust tools for reading from and writing to files, essential for data management:

- File Handling: Functions like fopen(), fclose(), fread(), and fwrite() enable users to handle file operations efficiently.
- Data Persistence: Functions such as load() and save() allow for storing and retrieving variables, facilitating long-term data storage and access.

String Functions

String manipulation is critical in many applications, and MATLAB offers several built-in functions for this purpose:

- Basic String Operations: Functions like strcat() for concatenation, strcmp() for comparison, and length() for obtaining the length of a string help in managing text data.

- Case Conversion: Functions upper() and lower() convert strings to uppercase and lowercase, respectively, aiding in text normalization.

Conversion Functions

Data type conversions are often necessary in programming:

- String to Number Conversion: Functions like str2double() and str2num() allow for converting string representations of numbers into numeric types.
- Number to String Conversion: Functions such as num2str() and sprintf() enable formatting numbers as strings for display or text manipulation.

Control Flow Functions

MATLAB includes standard control flow structures:

- Conditional Statements: The if, elseif, and else constructs allow for executing code based on logical conditions.
- Loops: The for and while loops enable iteration over data, making it possible to perform repeated operations efficiently.

Code

i)

```
canContinue = true;
userString = input('Enter a string: ', 's');
upperString = upper(userString);
lowerString = lower(userString);
fprintf('Uppercase: %s\n', upperString);
fprintf('Lowercase: %s\n', lowerString);
```

```
Enter a string:
heya
Uppercase: HEYA
Lowercase: heya
```

ii)

```
numString = input('Enter a numerical string: ', 's');
numberValue = str2double(numString);
if isnan(numberValue)
    fprintf('Error: The input is not a valid number.\n');
else
   result = numberValue + 10;
    fprintf('The result after adding 10 is: %.2f\n', result);
end
```

```
Enter a numerical string:
120
The result after adding 10 is: 130.00
```

iii)

```
vowelString = input('Enter a string: ', 's');
vowels = 'aeiou';
vowelCount = zeros(1, length(vowels));
for i = 1:length(vowels)
                    vowelCount(i) = sum(lower(vowelString) == vowels(i));
end
for i = 1:length(vowels)
   fprintf('Count of %s: %d\n', vowels(i), vowelCount(i));
end
```

```
Enter a string:
heya
Count of a: 1
Count of e: 1
Count of i: 0
Count of o: 0
Count of u: 0
```

Conclusion

The extensive library of built-in functions in MATLAB empowers users to perform complex calculations, analyze data, manipulate strings, and handle files with ease. Mastery of these functions is essential for efficient programming and effective problem-solving in various scientific and engineering disciplines. Understanding how to leverage these tools can significantly enhance productivity and the quality of code in MATLAB.

Date: 30/09/2024

Experiment 7

Programs based on scripts and user-defined functions

Aim

Simple Script and Function

i) Write a MATLAB script that:

- To read the range
- Print all the prime numbers within a range.

ii) Write a MATLAB script that:

- Reads a number
- Check of the number is a Pythagorean triplet using function
- Hint:The set of numbers (3, 4, 5) is called a Pythagorean triple, meaning the three positive integers satisfy the equation: a 2 +b 2 =c 2

iii) **Function to Compute Fibonacci Sequence**

Write a MATLAB script that:

- Defines a function fibonacci to compute the first 'n' terms of the Fibonacci sequence.
- Prompts the user to enter 'n', calls the function, and displays the sequence

Theoretical Background

Structure of a User-Defined Function in MATLAB

A user-defined function in MATLAB allows users to encapsulate code for specific tasks, promoting modularity and reusability. Understanding the structure and components of a user-defined function is essential for effective programming in MATLAB.

Basic Syntax

- A MATLAB function is typically written in a separate file with a .m extension. The basic syntax for defining a function is:

```
function [output1, output2, ...] = functionName(input1, input2, ...)
% Function code goes here
end
```

Anatomy of a MATLAB Function

Function Declaration:

- The function keyword is used to declare the function.

- Outputs are specified in square brackets. If there is only one output, the square brackets are optional.
- Inputs are provided within parentheses.
- Example:

```
function [result] = squareNumber(x)
result = x^2;
end
```

Input and Output Arguments:

- Inputs: These are the values passed to the function when it is called. They act as local variables within the function.
- Outputs: These are the results that the function returns upon completion.
- Example with Multiple Inputs and Outputs:

```
function [sum, product] = calculate(a, b)
sum = a + b;
product = a * b;
end
```

Function File:

- The filename must match the function name. For instance, if the function is named calculate, it should be saved as calculate.m.

Calling Functions:

- A function is called by using its name followed by input arguments in parentheses. It will return output values, if specified.
- Example of Calling a Function

```
[sumValue, productValue] = calculate(3, 5);
```

Code

i)

```
lowerLimit = input('Enter the lower limit of the range: ');
upperLimit = input('Enter the upper limit of the range: ');
primes = [];
for num = lowerLimit:upperLimit
   if isprime(num)
      primes(end + 1) = num;
   end
end
fprintf('Prime numbers between %d and %d are:\n', lowerLimit, upperLimit);
disp(primes);
```

```
Enter a number (format: [a, b, c]):
[2,2,3]
The numbers [2, 2, 3] do not form a Pythagorean triplet.
```

iii)

```
number = input('Enter a number (format: [a, b, c]): ');
if isPythagoreanTriplet(number)
    fprintf('The numbers [%d, %d, %d] form a Pythagorean triplet.\n', number(1), number(2), number(3));
else
   fprintf('The numbers [%d, %d, %d] do not form a Pythagorean triplet.\n', number(1), number(2), number(3));
end
function isTriplet = isPythagoreanTriplet(nums)
   a = nums(1);
   b = nums(2);
   c = nums(3);
   isTriplet = (a^2 + b^2 == c^2);
end
```

```
Enter the lower limit of the range:
1
Enter the upper limit of the range:
10
Prime numbers between 1 and 10 are:
    2     3     5     7
```

ii)

```
n = input('Enter the number of terms in the Fibonacci sequence: ');
fibonacciSequence = fibonacci(n);
fprintf('The first %d terms of the Fibonacci sequence are:\n', n);
disp(fibonacciSequence);
function fib = fibonacci(n)
   fib = zeros(1, n);
   if n >= 1, fib(1) = 0; end
   if n >= 2, fib(2) = 1; end
   for i = 3:n
      fib(i) = fib(i-1) + fib(i-2);
   end
end
```

```
Enter the number of terms in the Fibonacci sequence:
10
The first 10 terms of the Fibonacci sequence are:
    0    1    1    2    3    5    8   13   21   34
```

Conclusion

Understanding the structure of user-defined functions in MATLAB is crucial for efficient programming. By encapsulating code into functions, users can create reusable components that simplify complex tasks, enhance code organization, and improve readability. Mastering these elements enables programmers to write more efficient and maintainable MATLAB code.

Date: 15/10/2024

Experiment 8

Programs based on Advanced Plotting Techniques

Aim

Subplots

i) Write a MATLAB script that:

- Creates a figure with two subplots, the first subplot, plots a sine wave, the second subplot, plots a cosine wave.
- Represent the sine and Cosine waves as two graphs in the same figure
- Adds titles, labels, and legends to the plots.

3D Plotting

ii) Write a MATLAB script that:

- Generates a mesh grid of 'x' and 'y' values.
- Computes z=sin(√(x 2 +y 2)).
- Creates a 3D surface plot, line and scatter plot of 'z'.
- Adds titles, labels, and color bar.

iii) Write a MATLAB script that:

- Plots a sphere [x(t,s),y(t,s),z(t,s)]=[cos(t)cos(s),cos(t)sin(s),sin(t)] where [t,s]=[0,2pi]
- Adds titles, labels, and proper titles.

Theoretical Background

In MATLAB, plotting functions allow users to visualize mathematical functions and data. MATLAB's powerful plotting tools, like plot(), surf(), and scatter3(), are widely used for data representation.

2D Plotting (Sine and Cosine Functions):

The sine (sin(x)) and cosine (cos(x)) functions are standard trigonometric functions used to model periodic waveforms. In MATLAB, the plot() function is used to display continuous 2D graphs of these functions. The ability to create subplots using subplot() allows multiple graphs to be displayed within the same figure window, which is useful for comparison. Adding titles, axis labels, and legends using title(), xlabel(), ylabel(), and legend() helps in making the plots informative and visually organized.

Meshgrid and 3D Plotting:

MATLAB's meshgrid() function is essential for generating a grid of x and y coordinates over a 2D space, allowing for the evaluation of a function at each grid point. This is

crucial when working with functions of two variables (like $z=f(x,y)$ z=f(x,y)) to compute the corresponding z-values.

MATLAB provides several 3D plotting options:

- Surface Plots (surf()): These are used to plot a continuous 3D surface where the color of the surface can represent the height (z value). Surface plots provide a smooth and comprehensive visual representation of functions with two independent variables.
- Line Plots (plot3()): This function is used for plotting 3D lines by specifying x, y, and z coordinates, offering a way to visualize how a function or data evolves along a path.
- Scatter Plots (scatter3()): The scatter plot is a way to represent discrete points in 3D space, with optional coloring based on a fourth variable (e.g., z). This method is ideal for visualizing discrete data or sample points.

Parametric Equations and Sphere Plotting:

Parametric equations are commonly used in MATLAB for plotting curves and surfaces by defining each coordinate (x, y, z) as a function of one or two parameters (such as angles or time). For instance, to plot a sphere, parametric equations for the x, y, and z coordinates are used, and MATLAB's surf() function can display the 3D surface based on these parametric equations.

By using color shading options such as shading interp and colorbar(), MATLAB enhances the visualization, providing smooth transitions in color and aiding in the interpretation of the 3D surfaces or scatter plots.

Code

i)

```
x = 0:0.01:2*pi;
y_sin = sin(x);
y_cos = cos(x);
% First figure: Sine wave
figure;
plot(x, y_sin, 'b', 'LineWidth', 2);
title('Sine Wave');
xlabel('x (radians)');
ylabel('sin(x)');
legend('sin(x)');
grid on;
% Second figure: Cosine wave
figure;
plot(x, y_cos, 'r', 'LineWidth', 2);
title('Cosine Wave');
xlabel('x (radians)');
ylabel('cos(x)');
legend('cos(x)');
grid on;
```

ii)

```
x=linspace(-5,5,50);
y=linspace(-5,5,50);
[X,Y]=meshgrid(x,y);
z=sin(sqrt(X.^2 + Y.^2));
surf(X,Y,z)
```

```
colorbar
title('3D Surface Plot');
xlabel('x Axis');
ylabel('y Axis');
zlabel('z Axis');
figure;
plot3(X,Y,z);
title('3D line Plot');
xlabel('x Axis');
ylabel('y Axis');
zlabel('z Axis');
figure;
scatter3(X,Y,z);
title('3D scatter Plot');
xlabel('x Axis');
ylabel('y Axis');
zlabel('z Axis');
%c)
t = linspace(0, pi, 50);
s = linspace(0, 2*pi, 100);
[t, s] = meshgrid(t, s);
x = cos(t) .* cos(s);
y = cos(t) .* sin(s);
z = sin(t);
figure;
surf(x, y, z, 'FaceAlpha', 0.7);
title('3D Sphere');
xlabel('X-axis');
ylabel('Y-axis');
zlabel('Z-axis');
axis equal;
grid on;
light;
lighting phong;
```

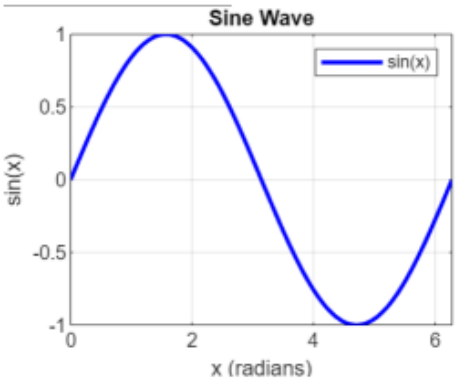
iii)

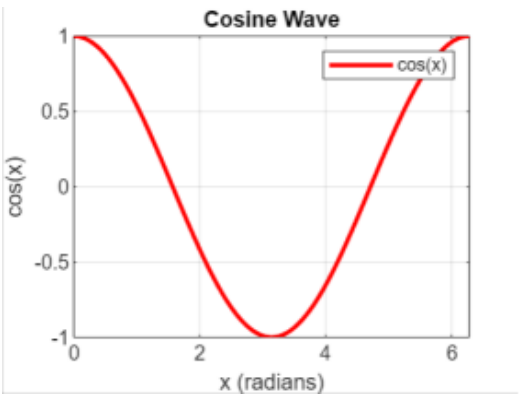
```
t = linspace(0, 2*pi, 50); % from 0 to pi
s = linspace(0, 2*pi, 100); % from 0 to
2*pi
[t, s] = meshgrid(t, s);

x = cos(t) .* cos(s);
y = cos(t) .* sin(s);
z = sin(t);
% Create the sphere plot
figure;
surf(x, y, z, 'FaceAlpha', 0.7);
title('3D Sphere');
xlabel('X-axis');
ylabel('Y-axis');
zlabel('Z-axis');
axis equal; % Equal scaling for all axes
grid on;
light;
lighting phong;
```

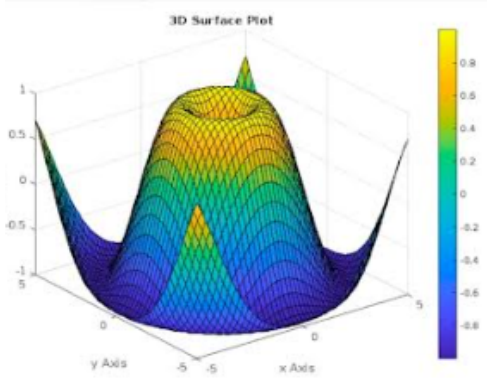
Output

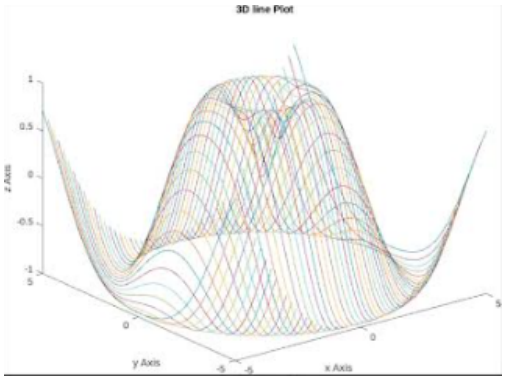
i)



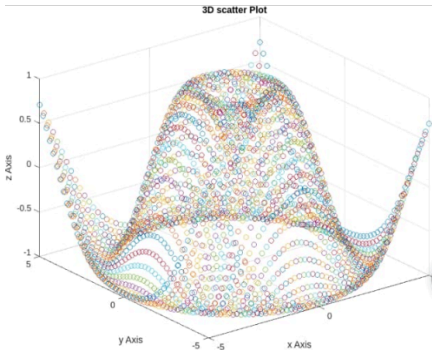


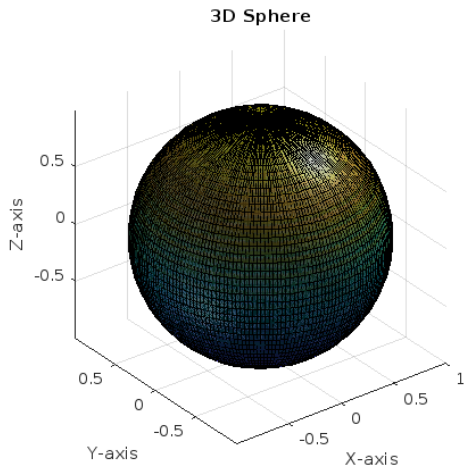
ii)





iii)





Conclusion

This experiment demonstrated MATLAB's powerful plotting capabilities for visualizing mathematical functions in 2D and 3D. By creating sine and cosine plots, using meshgrid() for 3D plots, and working with parametric equations to generate a sphere, we explored key tools like plot(), surf(), and scatter3(). These methods showcased MATLAB's versatility in representing data and complex functions, providing valuable insights into its visualization capabilities.

Date: 23/10/2024

Experiment 9

Programs based on two main data structures: cell arrays and structures

Aim

Cell Array Manipulation

i) Write a MATLAB script that:

- Has the cell array C = {3.14, "MATLAB", true; 7, [1 2 3], "hello";}
- Access and display the second element of the first row.
- Replace the third element of the second row with a new matrix [4 5 6].
- Modify the content of the first cell to store a structure containing information about a student (name, age, GPA). Display the updated cell array.

Structures

ii) Write a MATLAB script that:

- Defines a structure array to store information about three students, including fields for Name, Age, and marks of 4 subjects.
- Adds data for each student.
- Computes and displays the average grade of each student along with name. Generate a ranklist and store it in another structure and display it.

Theoretical Background

This experiment focuses on two fundamental data structures in MATLAB: cell arrays and structures. Both structures are essential for organizing and manipulating data, especially when dealing with heterogeneous types (like strings, numbers, and arrays).

Cell Arrays:

A cell array is a data type in MATLAB that allows you to store an array of different types of data. Each cell in a cell array can contain any type of data, including numbers, strings, arrays, and even other cell arrays.

Key Features:

- Heterogeneous Data: Each cell can contain different data types and sizes, making cell arrays versatile.
- Accessing Data: Data is accessed using curly braces {}. For example, C{1, 2} accesses the element in the first row and second column.

- Dynamic Size: Cell arrays can grow or shrink in size dynamically, which provides flexibility when managing data.

Use Cases:

- Storing mixed data types (e.g., strings, numbers, arrays).
- Organizing data in a tabular format where each row can have different data types.
- Useful for situations where traditional arrays (which require uniform data types) are inadequate.

Structures:

A structure is a data type in MATLAB that organizes data into fields. Each field can contain different types of data, and the fields can vary in size.

Key Features:

- Field Names: Structures allow for descriptive field names, making the data more interpretable. For example, a structure representing a student can have fields like name, age, and marks.
- Accessing Data: Data is accessed using dot notation, e.g., student.name accesses the name field of a structure variable student.
- Arrays of Structures: You can create arrays of structures, where each structure in the array can represent a different entity (e.g., multiple students).

Use Cases:

- Storing related data of varying types under a single entity, such as a student's name, age, and grades.
- Facilitating the organization of complex data types, especially in applications involving databases or records.

Code

i)

```
C = {3.14, 'MATLAB', true; 7, [1 2 3], 'hello'};
secondElementFirstRow = C{1, 2};
disp('Second element of the first row:');
disp(secondElementFirstRow);
C{2, 3} = [4 5 6];
student.name = 'John Doe';
student.age = 21;
student.GPA = 3.8;
C{1, 1} = student;
disp('Updated cell array contents:');
for row = 1:size(C, 1)
    for col = 1:size(C, 2)
        if isstruct(C{row, col})
            fprintf('Name:%s, Age:%d, GPA:%.2f ', ...
                C{row, col}.name, C{row, col}.age, C{row, col}.GPA);
        else
            fprintf('%s ', mat2str(C{row, col}));
        end
    end
    fprintf('\n');
end
```

Second element of the first row:

```
MATLAB
Updated cell array contents:
Name:John Doe, Age:21, GPA:3.80
'MATLAB'  true
7  [1 2 3]  [4 5 6]
```

ii)

```
students(1).Name = 'Alice';
students(1).Age = 20;
students(1).Marks = [85, 90, 78, 92];
students(2).Name = 'Bob';
students(2).Age = 22;
students(2).Marks = [75, 80, 85, 70];
students(3).Name = 'Charlie';
students(3).Age = 21;
students(3).Marks = [90, 88, 94, 95];
averageGrades = zeros(1, length(students));
for i = 1:length(students)
    averageGrades(i) = mean(students(i).Marks);
end
disp('Average Grades of Students:');
for i = 1:length(students)
    fprintf('%s: %.2f\n', students(i).Name, averageGrades(i));
end
[~, rankIndices] = sort(averageGrades, 'descend');
rankList = struct('Name', {}, 'AverageGrade', {});
for i = 1:length(rankIndices)
    rankList(i).Name = students(rankIndices(i)).Name;
    rankList(i).AverageGrade = averageGrades(rankIndices(i));
end
disp('Rank List:');
for i = 1:length(rankList)
    fprintf('%d. %s: %.2f\n', i, rankList(i).Name, rankList(i).AverageGrade);
end
```

```
Average Grades of Students:
Alice: 86.25
Bob: 77.50
Charlie: 91.75
Rank List:
1. Charlie: 91.75
2. Alice: 86.25
3. Bob: 77.50
```

Conclusion

Both cell arrays and structures are crucial for effective data management in MATLAB. They provide the flexibility to handle complex data types and relationships, which is essential in programming for data analysis, scientific computing, and more. Understanding these data structures is fundamental for students and professionals who work with MATLAB in various fields such as engineering, statistics, and data science.

Date: 23/10/2024

Experiment 10

Programs based on Advanced Functions

Aim

Anonymous Functions and Function Handles

i) Write a MATLAB script that:

- Defines an anonymous function to compute the square of a number.
- Creates a function handle to the anonymous function.
- Uses the function handle to compute and display the square of a user-provided number.

Nested Functions

ii) Write a MATLAB script that:

- Defines a main function to compute the area of a rectangle.
- Within the main function, defines a nested function to compute the perimeter of the rectangle.
- Prompts the user for the length and width of the rectangle.
- Call the nested function to compute the perimeter and the main function to compute the area, then displays both results.

Theoretical Background

In MATLAB, effective programming often involves the use of various function types to optimize code organization, enhance readability, and improve flexibility. This experiment focuses on three pivotal concepts: anonymous functions, function handles, and nested functions. Understanding these concepts is crucial for developing sophisticated algorithms, simplifying data processing, and ensuring clean code structures.

Anonymous Functions:

Anonymous functions are a feature in MATLAB that allows users to define functions without creating a separate file. They are typically used for simple operations and can be created in a single line using the @ symbol.

Key Features:

- Concise Syntax: The syntax for defining an anonymous function is straightforward, enabling quick creation of functions for simple calculations or operations.
- Multiple Inputs: Anonymous functions can accept multiple input arguments, which increases their utility in various scenarios.

- No Formal Declaration: Unlike regular functions that require a file and a function name, anonymous functions are defined inline, which reduces overhead when the functionality is straightforward.

Use Cases:

- Mathematical Operations: Frequently used in mathematical modeling, optimization, or data analysis where functions can be defined inline for operations like squaring numbers or computing exponential functions.
- Callbacks in GUI: In graphical user interfaces (GUIs), anonymous functions can serve as callbacks, providing a way to respond to user actions without cluttering the code with multiple small functions.

Example

An anonymous function for squaring a number can be defined as follows:

```
square = @(x) x^2;
result = square(5);  % This will output 25
```

Function Handles:

A function handle is a MATLAB data type that represents a function, allowing you to pass functions as arguments, store them in variables, and invoke them dynamically. Function handles are essential for creating flexible and modular code.

Key Features:

- Dynamic Invocation: Function handles facilitate the dynamic execution of functions, allowing them to be passed as parameters to other functions, enhancing code modularity.
- Versatility: They can reference not just anonymous functions, but also named functions, enabling more complex function manipulations.

Use Cases:

- Higher-Order Functions: In mathematical contexts, function handles enable the implementation of higher-order functions, which take other functions as input. This is useful for operations like integration and optimization.
- Callbacks in Events: Function handles are frequently used in event-driven programming, where a function is called in response to certain events (e.g., mouse clicks, timer events).

Example:

Using a function handle for the previously defined anonymous function:

```
fhandle = @square;   % Assign the anonymous function to a function handle
result = fhandle(3); % This computes 9
```

Nested Functions:

Nested functions are functions defined within the body of another function. They provide a mechanism for encapsulating functionality and accessing the parent function's workspace.

Key Features:

- Access to Parent Variables: Nested functions can access and modify variables in their parent function's scope, making it easier to work with shared data.
- Encapsulation and Organization: By keeping related functions together, nested functions promote cleaner and more maintainable code. They help localize functionality that is only relevant to the parent function.

Use Cases:

- Helper Functions: When a function needs to perform several related calculations, defining a nested function allows for better organization without exposing helper functions to the entire program.
- Complex Algorithms: In more complex algorithms, nested functions can streamline code by grouping related operations, improving readability and reducing the risk of naming conflicts.

Code

i)

```
squareFunc = @(x) x.^2;
squareHandle = squareFunc;
userInput = input('Enter a number to compute its square: ');
result = squareHandle(userInput);
fprintf('The square of %.2f is %.2f\n', userInput, result);
```

```
Enter a number to compute its square:
10
The square of 10.00 is 100.00
```

ii)

```
function rectangleAreaAndPerimeter()
    length = input('Enter the length of the rectangle: ');
     width = input('Enter the width of the rectangle: ');
    perimeter = computePerimeter(length, width);
    area = length * width;
     fprintf('Area of the rectangle: %.2f\n', area);
       fprintf('Perimeter of the rectangle: %.2f\n', perimeter);
   function p = computePerimeter(l, w)
       p = 2 * (l + w);
   end
end
rectangleAreaAndPerimeter()
```

```
Enter the length of the rectangle:
8
Enter the width of the rectangle:
10
Area of the rectangle: 80.00
Perimeter of the rectangle: 36.00
```

Conclusion

Understanding and utilizing anonymous functions, function handles, and nested functions enhances the capability to write clean, efficient, and organized code in MATLAB. These features allow for flexibility in programming, making it easier to handle complex operations and improve overall code readability. Mastery of these concepts is crucial for anyone looking to leverage MATLAB for mathematical computations, data analysis, and algorithm development.

Date: 04/11/2024

Experiment 11

Programs based on image processing

Aim

Basic Image Manipulation

i) Write a MATLAB script that:

- Reads an image file.
- Converts the image to grayscale.
- Displays the original and grayscale images side by side.

Edge Detection

ii) Write a MATLAB script that:

- Reads an image file
- Converts the image to grayscale.
- Applies a different edge detection algorithm to the image
- Displays the original image and all the edge-detected images side by side.

Image scaling

iii) Write a MATLAB script that:

- Reads an image file
- Converts the image to grayscale.
- Implement a custom function that processes the image pixel by pixel and computes the average of the neighboring pixels replacing each pixel of an image with the average value of its four nearest neighbors (top, bottom, left, and right).
- The border pixels need not be processed because they do not have four neighbors.

Theoretical Background

1. Basic Image Manipulation: Digital images are represented as matrices of pixel values. When an image is loaded, MATLAB stores it as a matrix, with color images represented by three color channels (Red, Green, and Blue) and grayscale images by a single intensity channel. Converting a color image to grayscale reduces data complexity, focusing only on brightness values.

- **Basic Image Manipulation**: This function reads an image file and stores it as a matrix in MATLAB. It supports various formats (like JPG, PNG, TIFF). For example, image = imread('file.jpg') loads the image file "file.jpg" into the workspace

- **rgb2gray**: This function converts an RGB image to grayscale. Each pixel's color information is converted into a single intensity value, creating a matrix with one intensity value per pixel, which simplifies further processing. For example, grayImage = rgb2gray(image) converts the RGB image to grayscale.
- **imshow**:This function displays an image in a MATLAB figure window, which helps visualize images at different processing stages. For example, imshow(image) shows the image matrix as a picture in the figure window

2. Edge Detection: Edge detection is the process of finding areas in an image where intensity values change significantly, which typically marks the boundaries of objects. MATLABprovides various algorithms to highlight these changes, each suitable for different scenarios

- **edge**: This function applies edge detection to an image, offering multiple algorithms. Its syntax is edgeImage = edge(grayImage, 'method'), where 'method' specifies the algorithm:
    - **Sobel**: Detects edges by evaluating intensity gradients in horizontal and vertical directions using convolution filters.
    - **Canny**:Amulti-step process involving gradient calculation, non-maximum suppression, and edge thresholding, which results in precise edge detection with fewer false edges.
    - **Prewitt**: Similar to Sobel, using a different filter to detect changes in intensity, mainly for simpler edge detection tasks.
- **subplot**: This function divides the figure window into a grid, allowing multiple images to be displayed simultaneously for comparison. It helps visualize the original and processed images side by side. For instance, subplot(1, 3, 2) divides the figure into a 1-row by 3-column grid and selects the middle position for display

3. Image Scaling: Image scaling or smoothing often involves adjusting pixel values based on neighbouring pixels. This technique helps reduce noise by averaging pixel values, creating a smoother appearance. By averaging each pixel with its four nearest neighbours, high-frequency variations (noise) are minimized.

- **Custom function and indexing**: MATLAB allows you to define custom functions to process images at the pixel level. Using loops and matrix indexing, each pixel can be replaced with the average value of its top, bottom, left, and right neighbours. This type of function typically accesses individual pixels by row and column indices, calculating the average of neighbouring pixels and replacing the central pixel value
- **Borderhandling**: When processing pixels, border pixels usually don't have a full set of neighbours, so they're often left unprocessed or retained as-is in the scaled image.

Code

i)

```
fileimage = imread('/MATLAB Drive/WALLPAPERS/lenna.jpg');
grayImage = rgb2gray(image);
images side by side figure;
subplot(1, 2, 1);
imshow(image);
title('Original Image');
subplot(1, 2, 2);
imshow(grayImage);
title('Grayscale Image');
```

ii)

```
image = imread('/MATLAB Drive/WALLPAPERS/C9.jpg');
grayImage = rgb2gray(image);
edgesSobel = edge(grayImage, 'sobel');
edgesCanny = edge(grayImage, 'canny');
edgesPrewitt = edge(grayImage, 'prewitt');
figure('Units', 'normalized', 'Position', [0.1 0.1 0.8 0.4]);
subplot(1, 4, 1);
imshow(image);
title('Original Image', 'FontSize', 14, 'FontWeight', 'bold');
subplot(1, 4, 2);
imshow(edgesSobel);
title('Sobel Edge Detection', 'FontSize', 14, 'FontWeight', 'bold');
subplot(1, 4, 3);
imshow(edgesCanny);
title('Canny Edge Detection', 'FontSize', 14, 'FontWeight', 'bold');
subplot(1, 4, 4);
imshow(edgesPrewitt);
title('Prewitt Edge Detection', 'FontSize', 14, 'FontWeight', 'bold');
subplotHandles = findall(gcf, 'type', 'axes');
for i = 1:numel(subplotHandles)
subplotHandles(i).Position = subplotHandles(i).Position + [0 0.1 0 0];
end
```

iii)

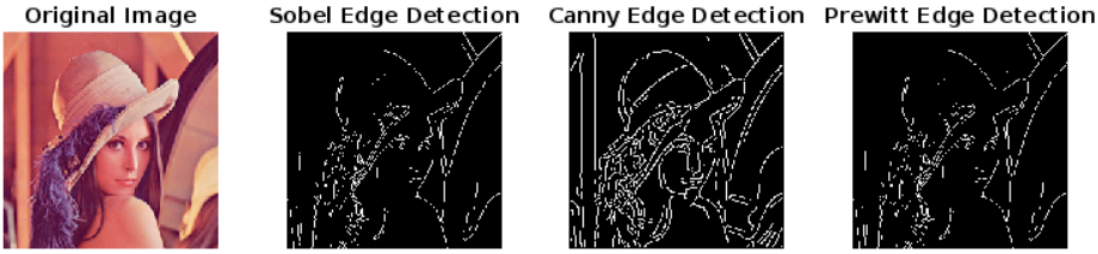
```
image = imread('/MATLAB Drive/WALLPAPERS/C9.jpg');
grayImage = double(rgb2gray(image));
[rows, cols] = size(grayImage);
scaledImage = grayImage;
% Process each pixel (ignoring the borders)
for i = 2:rows-1
```

```
  for j = 2:cols-1
     averageValue = (grayImage(i-1, j) +
grayImage(i+1, j) + grayImage(i, j-1) +
grayImage(i, j+1)) / 4;
     scaledImage(i, j) = averageValue;
end
end
scaledImage = uint8(scaledImage);
figure;
subplot(1, 2, 1);
imshow(uint8(grayImage));
title('Original Grayscale Image');
subplot(1, 2, 2); imshow(scaledImage);
title('Scaled Image Averaging)');
```

Output

i)

ii)



iii)

Conclusion

This experiment covered basic image processing techniques using MATLAB, including grayscale conversion, edge detection, and smoothing through pixel averaging. Grayscale images simplify data, while edge detection algorithms (Sobel, Prewitt, and Canny) highlight object boundaries. Custom scaling reduced noise by averaging pixels with their neighbours, enhancing clarity. Using MATLAB functions like imread, rgb2gray, and edge, these techniques provide a solid foundation for image analysis and applications in fields like object detection and medical imaging.