JAMIA MILLIA ISLAMIA UNIVERSITY

NEW DELHI, 110025

COMPUTER ENGINEERING DEPARTMENT,

FACULTY OF ENGINEERING AND TECHNOLOGY,

A Lab File of
ADVANCED COMPUTING

By

Bittu Singh
(23MCS007)

Submitted To:

Dr. Sarfaraz Masood

PROGRAM-1

WAP to implement Ascending / Descending sort on a given list of numbers.

Implementation:

```
% Function to implement ascending and descending sort
function [sorted_asc, sorted_desc] = sort_numbers(numbers)
    % Ascending sort
    sorted_asc = sort(numbers);

    % Descending sort
    sorted_desc = sort(numbers, 'descend');
end

% Example usage
numbers = [5, 2, 9, 1, 7];
[sorted_asc, sorted_desc] = sort_numbers(numbers);

disp('Ascending order:');
disp(sorted_asc);

disp('Descending order:');
disp(sorted_desc);
```

Output:

```
Ascending order:
     1     2     5     7     9
Descending order:
     9     7     5     2     1
```

PROGRAM-2

Implement the following function and plot its curve
f(x) = 0 if x<=0; 1 if x>0

Implementation:

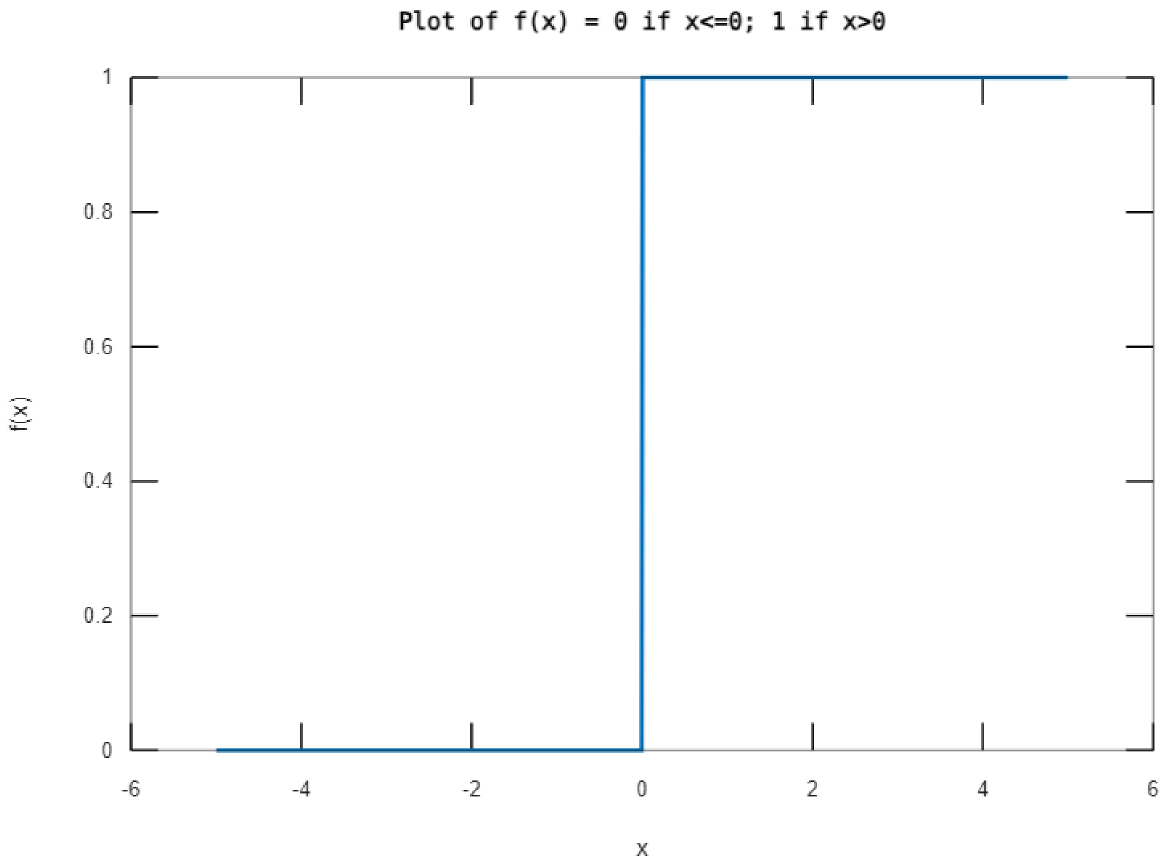
```
% Define the function
function y = f(x)
    y = zeros(size(x)); % Initialize output with zeros
    y(x > 0) = 1; % Set elements to 1 where x > 0
end

% Generate x values
x = linspace(-5, 5, 1000); % Generate 1000 points between -5 and 5

% Compute y values
y = f(x);

% Plot the curve
plot(x, y, 'LineWidth', 2);
xlabel('x');
ylabel('f(x)');
title('Plot of f(x) = 0 if x<=0; 1 if x>0');
grid on;
```

Output:



PROGRAM-3

Implement the following function and plot it's curve

f(x) = -1 if x<0; 0 if x = 0; 1 if x>0

Implementation:

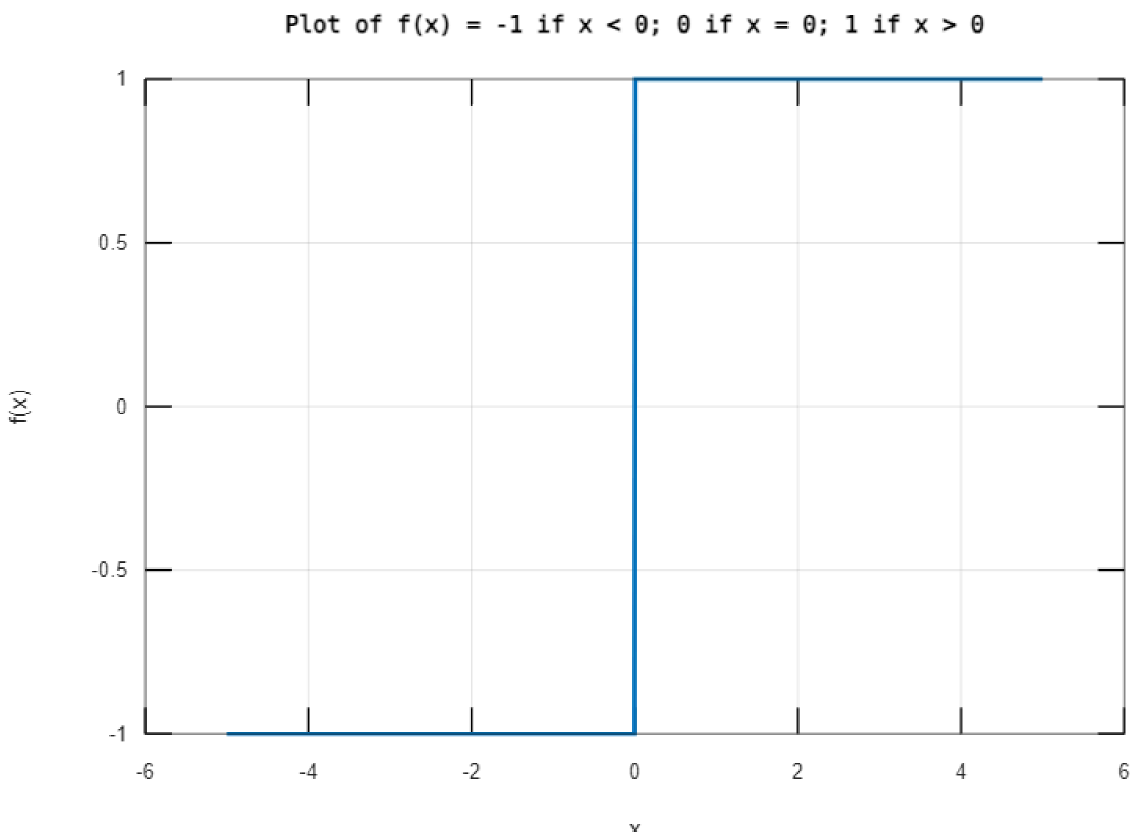
```
% Define the function
function y = f(x)
    y = zeros(size(x)); % Initialize output with zeros
    y(x < 0) = -1; % Set elements to -1 where x < 0
    y(x == 0) = 0; % Set elements to 0 where x = 0
    y(x > 0) = 1; % Set elements to 1 where x > 0
end

% Generate x values
x = linspace(-5, 5, 1000); % Generate 1000 points between -5 and 5

% Compute y values
y = f(x);

% Plot the curve
plot(x, y, 'LineWidth', 2);
xlabel('x');
ylabel('f(x)');
title('Plot of f(x) = -1 if x < 0; 0 if x = 0; 1 if x > 0');
grid on;
```

Output:



PROGRAM-4

Implement the following function also plot its curve

f(x) = 1/(1 + exp(-x))

Implementation:

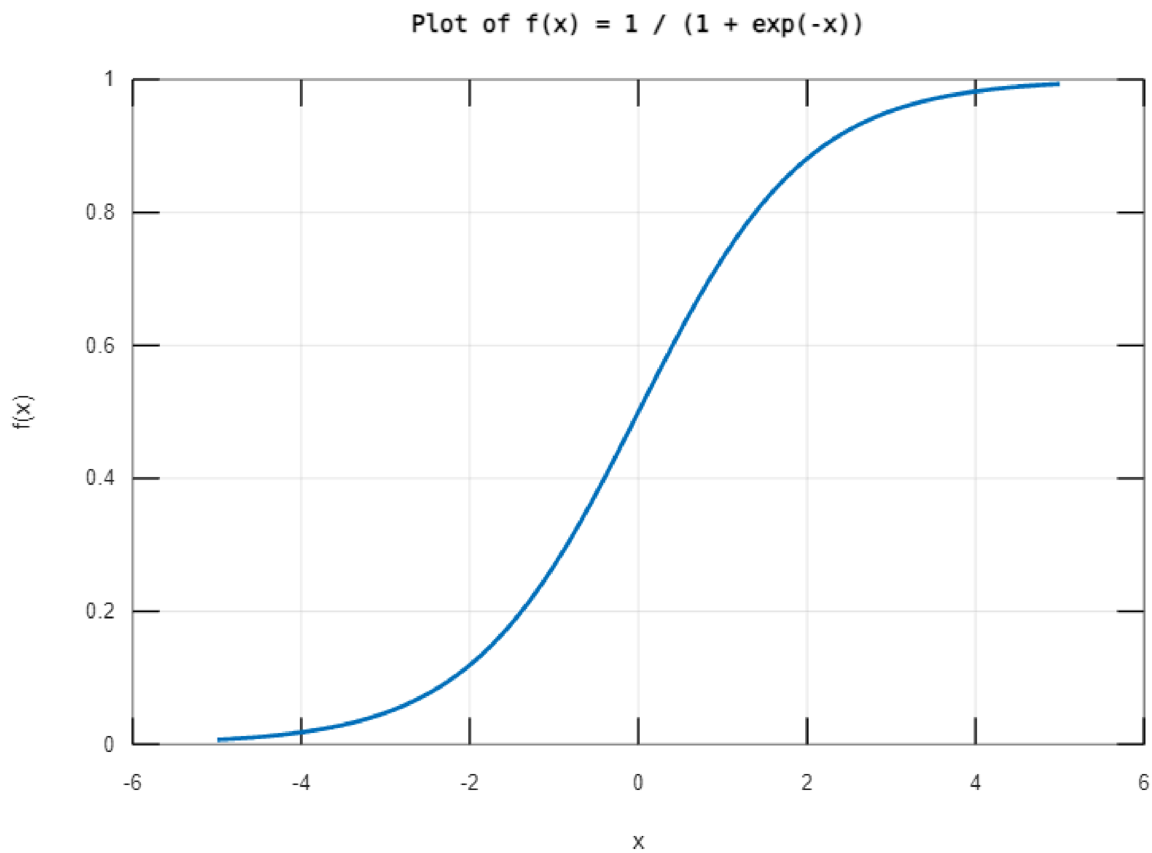
```
% Define the function
function y = f(x)
    y = 1 ./ (1 + exp(-x));
end

% Generate x values
x = linspace(-5, 5, 1000); % Generate 1000 points between -5 and 5

% Compute y values
y = f(x);

% Plot the curve
plot(x, y, 'LineWidth', 2);
xlabel('x');
ylabel('f(x)');
title('Plot of f(x) = 1 / (1 + exp(-x))');
grid on;
```

Output:



PROGRAM-5

Implement the following function also plot its curve

f(x) = (exp(x) - exp(-x))/(exp(x) + exp(-x))

Implementation:

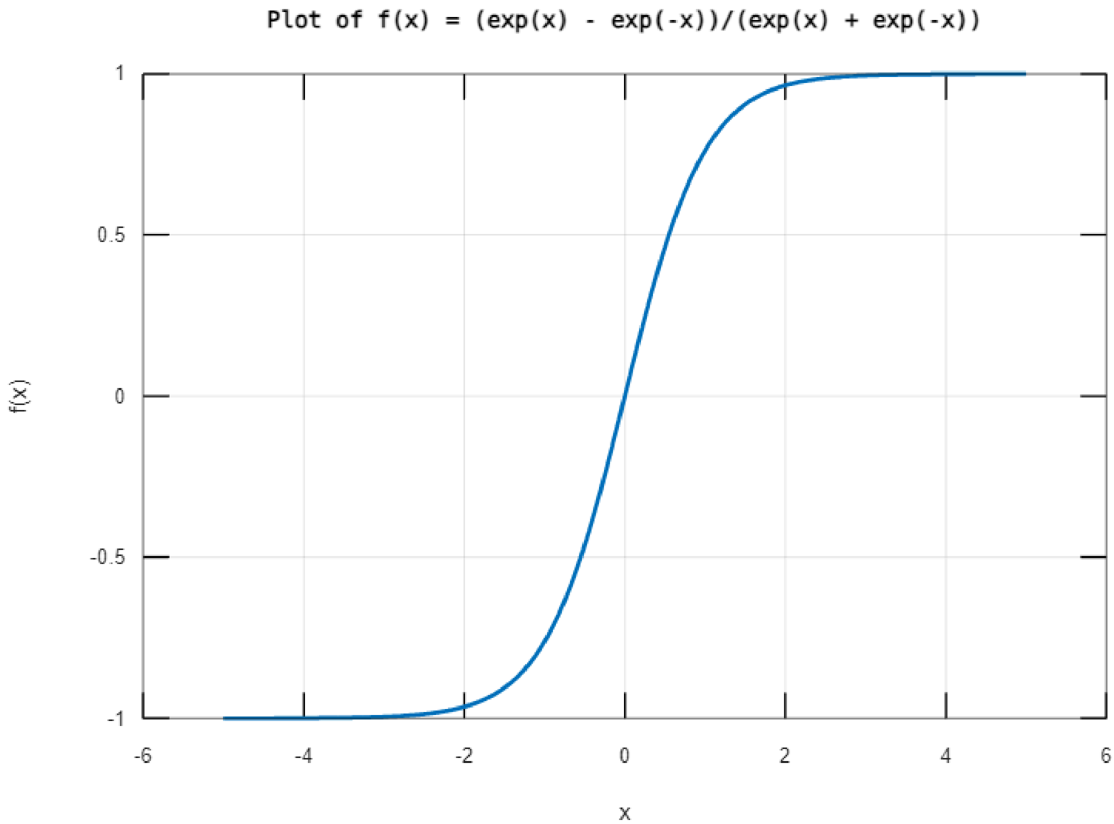
```
% Define the function
function y = f(x)
   y = (exp(x) - exp(-x)) ./ (exp(x) + exp(-x));
end

% Generate x values
x = linspace(-5, 5, 1000); % Generate 1000 points between -5 and 5

% Compute y values
y = f(x);

% Plot the curve
plot(x, y, 'LineWidth', 2);
xlabel('x');
ylabel('f(x)');
title('Plot of f(x) = (exp(x) - exp(-x))/(exp(x) + exp(-x))');
grid on;
```

Output:



PROGRAM-6

WAP to GENERATE and PLOT RANDOM nos with Uniform and Normal Distributions.

Implementation:

```
% Define the number of random numbers to generate
num_samples = 1000;

% Generate random numbers with uniform distribution
uniform_numbers = rand(num_samples, 1);

% Generate random numbers with normal distribution (mean = 0, standard deviation = 1)
normal_numbers = randn(num_samples, 1);

% Plot histograms for both distributions
figure;

% Plot histogram for uniform distribution
subplot(2, 1, 1);
histogram(uniform_numbers, 20, 'Normalization', 'probability');
title('Uniform Distribution');
xlabel('Value');
ylabel('Probability');

% Plot histogram for normal distribution
subplot(2, 1, 2);
histogram(normal_numbers, 20, 'Normalization', 'probability');
title('Normal Distribution');
xlabel('Value');
ylabel('Probability');
```

Output:

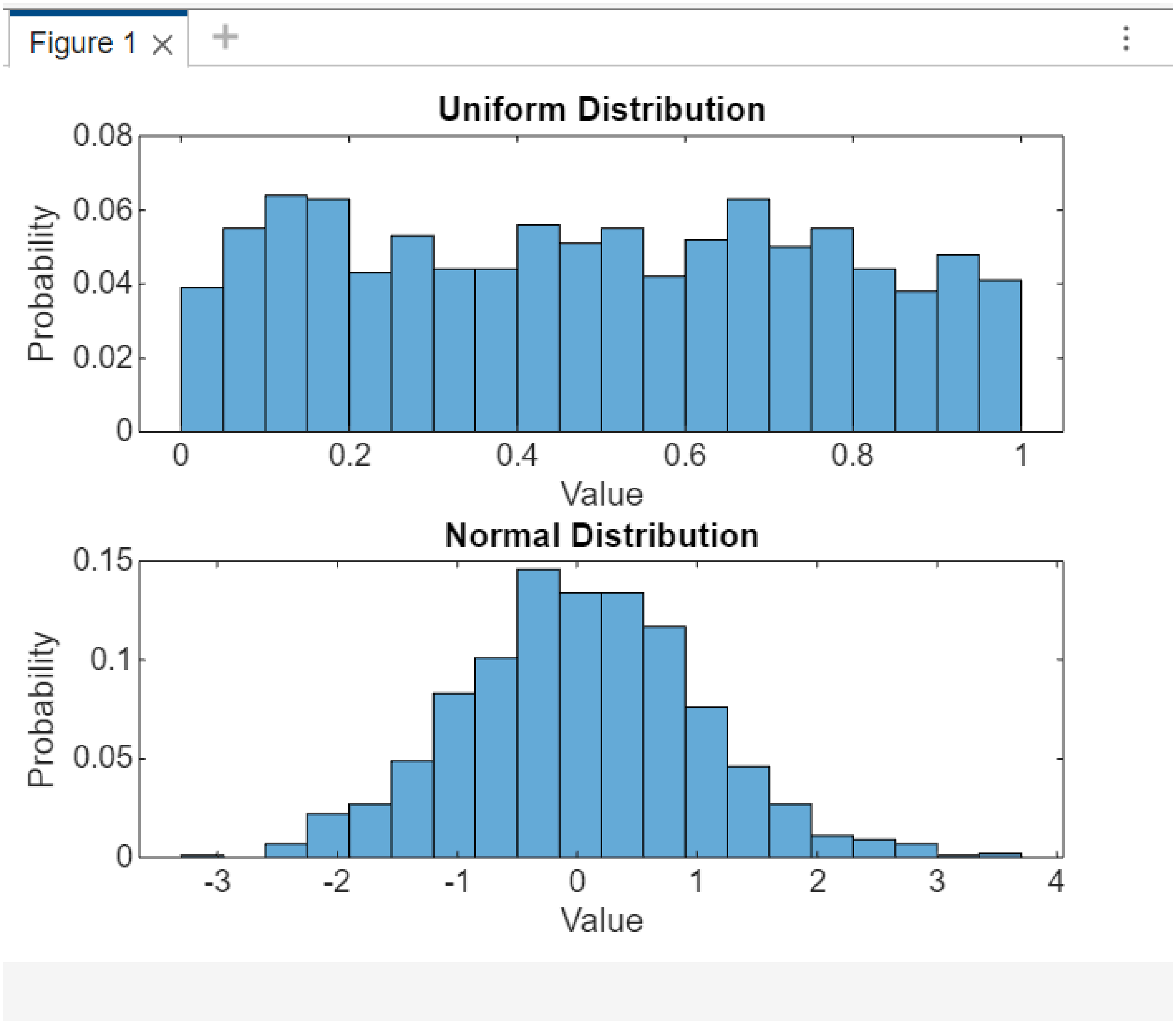


Figure 1
Uniform Distribution
Probability
Value
0.08
0.06
0.04
0.02
0
0
0.2
0.4
0.6
0.8
1
Normal Distribution
Probability
Value
0.15
0.1
0.05
0
-3
-2
-1
0
1
2
3
4

PROGRAM-7

WAP to Read and Write Data from Excel Files.

Implementation:

```
% Define the file names
input_file = 'input_data.xlsx'; % Excel file to read from
output_file = 'output_data.xlsx'; % Excel file to write to

% Read data from Excel file
data = xlsread(input_file);

% Display the read data
disp('Data read from Excel file:');
disp(data);

% Manipulate the data (optional)
% For example, let's add 1 to each element of the data
data = data + 1;

% Write data to Excel file
xlswrite(output_file, data);

disp('Data has been written to Excel file.');
```

PROGRAM-8

WAP to find missing entries from Excel Datasheet and replace these missing (Nan) entries with the MEAN of that column.

Implementation:

```
% Define the file name
file_name = 'data_sheet.xlsx'; % Excel file to read from and write to

% Read data from Excel file
data = xlsread(file_name);

% Find missing entries (NaN values) and replace them with the mean of that column
for col = 1:size(data, 2)
  nan_indices = isnan(data(:, col)); % Find NaN indices in each column
  if any(nan_indices)
    column_mean = mean(data(~nan_indices, col)); % Calculate mean of non-NaN
values in the column
    data(nan_indices, col) = column_mean; % Replace NaN values with the mean
  end
end

% Write updated data to Excel file
xlswrite(file_name, data);

disp('Missing entries replaced with column means and data written to Excel file.');
```

PROGRAM-9

WAP to Implement McCulloch Pitts Neuron.

Implementation:

```
inputs=[1,0,1,0; 1,1,1,1; 1,1,1,0; 1,0,0,1; 1,1,1,1;];
%x1 = [1; 0; 1; 0; 1; 0; 1; 0];
%x2 = [1; 1; 0; 0; 1; 1; 0; 0];
%x3 = [1; 1; 1; 1; 0; 0; 0; 0];
target=[0;1;0;0;1];
weights=[0.4,0.8,0.4,0.8];

output = zeros(size(inputs,1),1);
threshold=2;

for i = 1:size(inputs,1);
    disp(inputs(i,:))
    disp(inputs(i,:).*weights);
    if  sum (inputs(i,:).*weights) > threshold;
        output(i) = 1;
    else
        output(i) = 0;
    end;
end;
disp(output)
result= horzcat(output,target)
```

Output:

```
result =

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

PROGRAM-10

WAP to implement a Hebb's Neuron.

Implementation:

```
inputs = [1, 0, 1, 0;
          1, 1, 1, 1;
          1, 1, 1, 0;
          1, 0, 0, 1;
          1, 1, 1, 1];
target = [0; 1; 0; 0; 1];
weights = [0.4, 0.8, 0.4, 0.8];
threshold = 2;

output = zeros(size(inputs, 1), 1);

% Hebb's Learning Rule
for i = 1:size(inputs, 1)
   disp(inputs(i, :));
   disp(inputs(i, :) .* weights);

   % Update weights using Hebb's learning rule
   weights = weights + (inputs(i, :) .* target(i));

   if sum(inputs(i, :) .* weights) > threshold
      output(i) = 1;
   else
      output(i) = 0;
   end
end

disp('Output:');
disp(output);

result = horzcat(output, target);
disp('Result:');
disp(result);
```

Output:

```
Result:
    0   0
    1   1
    1   0
    1   0
    1   1
```

PROGRAM-11

WAP to implement PERCEPTRON Network.

Implementation:

```
% Perceptron Learning Algorithm

% Define training data
X = [0 0; 0 1; 1 0; 1 1]; % input features
y = [0; 0; 0; 1]; % target labels

% Initialize weights and bias
w = randn(1, size(X, 2));  weights
b = randn(); % bias
learning_rate = 0.1;
disp('Weights before training:');
disp(w);
disp('Bias before training:');
disp(b);

% Training the perceptron
epochs = 70;
for epoch = 1:epochs
   for i = 1:size(X, 1)
      % Compute the output of the perceptron
      output = X(i, :) * w' + b;

      % Update weights and bias using perceptron learning rule
      w = w + learning_rate * (y(i) - output) * X(i, :);
      b = b + learning_rate * (y(i) - output);
   end
end

% Test the perceptron
disp('Weights after training:');
disp(w);
disp('Bias after training:');
disp(b);

% Predictions
disp('Predictions:');
```

```
for i = 1:size(X, 1)
    pred = X(i, :) * w' + b;
    disp(['Input: ', num2str(X(i, :)), ', Prediction: ', num2str(pred)]);
end
```

Output:

```
weights =

   2.4000   2.8000   2.4000   2.8000

Weights before training:
   0.4271  -0.2810
Bias before training:
0.1762
Weights after training:
   0.5524   0.5242
Bias after training:
-0.2732
Predictions:
Input: 0  0, Prediction: -0.27324
Input: 0  1, Prediction: 0.25094
Input: 1  0, Prediction: 0.27918
Input: 1  1, Prediction: 0.80336
```

PROGRAM-12

WAP to implement ADALINE Network.

Implementation:

```
% Define inputs and targets for XOR gate
inputs = [0, 0;
          0, 1;
          1, 0;
          1, 1];
target = [0; 1; 1; 0];

% Initialize weights and bias
weights = rand(1, size(inputs, 2));
bias = rand;

% Learning rate
learning_rate = 0.1;

% Maximum number of iterations
max_iterations = 1000;

% Train the ADALINE network
for iter = 1:max_iterations
   % Compute output (activation) for each input pattern
   outputs = inputs * weights' + bias;

   % Compute errors
   errors = target - outputs;

   % Update weights and bias using gradient descent
   weights = weights + learning_rate * errors' * inputs;
   bias = bias + learning_rate * sum(errors);

   % Compute mean squared error
   mse = mean(errors.^2);

   % Check if convergence criteria met
   if mse < 0.01
      disp(['Convergence reached after ', num2str(iter), ' iterations']);
      break;
   end
```

```
end

% Display final weights and bias
disp('Final weights:');
disp(weights);
disp('Final bias:');
disp(bias);

% Test the trained ADALINE network
predicted_output = inputs * weights' + bias;

% Display predicted output
disp('Predicted output:');
disp(predicted_output);
```

Output:

```
Final weights:
   1.0408e-16   1.0408e-16
Final bias:
0.5000
Predicted output:
   0.5000
   0.5000
   0.5000
   0.5000
```

PROGRAM-13

WAP to implement MADALINE Network.

Implementation:

```
% Define inputs and targets for XOR gate
inputs = [0, 0;
          0, 1;
          1, 0;
          1, 1];
target = [0; 1; 1; 0];

% Initialize weights and biases for first ADALINE layer
weights_layer1 = rand(2, size(inputs, 2)); % Weights for first layer
bias_layer1 = rand(2, 1); % Biases for first layer

% Initialize weights and bias for output ADALINE
weights_output = rand(1, 2); % Weights for output layer
bias_output = rand; % Bias for output layer

% Learning rate
learning_rate = 0.1;

% Maximum number of iterations
max_iterations = 1000;

% Train the MADALINE network
for iter = 1:max_iterations
   % Forward pass through first layer
   output_layer1 = inputs * weights_layer1' + bias_layer1';
   output_layer1 = 1 ./ (1 + exp(-output_layer1)); % Sigmoid activation

   % Forward pass through output layer
   output = output_layer1 * weights_output' + bias_output;
   output = 1 ./ (1 + exp(-output)); % Sigmoid activation

   % Compute errors
   errors = target - output;

   % Backpropagation through output layer
   delta_output = errors .* output .* (1 - output); % Error * derivative of sigmoid
```

```
    % Backpropagation through first layer
    delta_layer1 = (delta_output * weights_output) .* output_layer1 .* (1 - output_layer1);

    % Update weights and biases for output layer
    weights_output = weights_output + learning_rate * delta_output' * output_layer1;
    bias_output = bias_output + learning_rate * sum(delta_output);

    % Update weights and biases for first layer
    weights_layer1 = weights_layer1 + learning_rate * delta_layer1' * inputs;
    bias_layer1 = bias_layer1 + learning_rate * sum(delta_layer1)';

    % Compute mean squared error
    mse = mean(errors.^2);

    % Check if convergence criteria met
    if mse < 0.01
        disp(['Convergence reached after ', num2str(iter), ' iterations']);
        break;
    end
end

% Display final weights and biases
disp('Final weights for first layer:');
disp(weights_layer1);
disp('Final biases for first layer:');
disp(bias_layer1);
disp('Final weights for output layer:');
disp(weights_output);
disp('Final bias for output layer:');
disp(bias_output);

% Test the trained MADALINE network
output_layer1_test = inputs * weights_layer1' + bias_layer1';
output_layer1_test = 1 ./ (1 + exp(-output_layer1_test)); % Sigmoid activation
predicted_output = output_layer1_test * weights_output' + bias_output;
predicted_output = 1 ./ (1 + exp(-predicted_output)); % Sigmoid activation

% Display predicted output
disp('Predicted output:');
disp(predicted_output);
```

Output:

```
Final weights for first layer:
   0.2383   0.2141
   1.0141   0.2202
Final biases for first layer:
   0.3600
   0.7250
Final weights for output layer:
   0.1337   0.1938
Final bias for output layer:
-0.2333
Predicted output:
   0.4940
   0.4980
   0.5045
   0.5073
```

PROGRAM-14

WAP to implement BACKPROPAGATION ALGORITHM for an MLFANN.

Implementation:

```
% Define inputs and targets for XOR gate
inputs = [0, 0;
          0, 1;
          1, 0;
          1, 1];
target = [0; 1; 1; 0];

% Initialize weights and biases for hidden layer and output layer
hidden_layer_weights = rand(2, size(inputs, 2)); % Weights for hidden layer
output_layer_weights = rand(1, 2); % Weights for output layer
hidden_layer_bias = rand(2, 1); % Bias for hidden layer
output_layer_bias = rand; % Bias for output layer

% Learning rate
learning_rate = 0.1;

% Maximum number of iterations
max_iterations = 1000;

% Train the MLFFANN using Backpropagation algorithm
for iter = 1:max_iterations
   % Forward pass
   hidden_layer_output = 1 ./ (1 + exp(-(inputs * hidden_layer_weights' +
hidden_layer_bias')));
   output_layer_output = 1 ./ (1 + exp(-(hidden_layer_output * output_layer_weights' +
output_layer_bias)));

   % Compute errors
   output_error = target - output_layer_output;
   hidden_error = (output_error * output_layer_weights) .* hidden_layer_output .* (1 -
hidden_layer_output);

   % Backpropagation
   output_layer_weights = output_layer_weights + learning_rate * output_error' *
hidden_layer_output;
   output_layer_bias = output_layer_bias + learning_rate * sum(output_error);
```

```
    hidden_layer_weights = hidden_layer_weights + learning_rate * hidden_error' *
inputs;
    hidden_layer_bias = hidden_layer_bias + learning_rate * sum(hidden_error)';

    % Compute mean squared error
    mse = mean(output_error.^2);

    % Check if convergence criteria met
    if mse < 0.01
        disp(['Convergence reached after ', num2str(iter), ' iterations']);
        break;
    end
end

% Display final weights and biases
disp('Final weights for hidden layer:');
disp(hidden_layer_weights);
disp('Final biases for hidden layer:');
disp(hidden_layer_bias);
disp('Final weights for output layer:');
disp(output_layer_weights);
disp('Final bias for output layer:');
disp(output_layer_bias);

% Test the trained MLFFANN
hidden_layer_output_test = 1 ./ (1 + exp(-(inputs * hidden_layer_weights' +
hidden_layer_bias')));
predicted_output = 1 ./ (1 + exp(-(hidden_layer_output_test * output_layer_weights' +
output_layer_bias)));

% Display predicted output
disp('Predicted output:');
disp(predicted_output);
```

Output:

```
Final weights for hidden layer:
   4.8242   4.4753
   2.0780   0.1090
Final biases for hidden layer:
  -1.0928
  -0.2792
Final weights for output layer:
   4.7050  -2.7280
Final bias for output layer:
-2.0479
Predicted output:
   0.1149
   0.7781
   0.5514
   0.5696
```