<u>**INDEX**</u>

Sr. No.	Experiment	Signature
1	Program to implement basic activation functions.	
2	Program to implement McCulloch-Pitts Neuron.	
3	Program to implement a simple Neuron with various activation functions	
4	Program to implement a simple perceptron using the perceptron learning algorithm	
5	Program to implement delta and back propagation algorithm.	
6	Program to implement a simple perceptron with radial basis function (RBF) activation function	
7	Program to implement all the fuzzy set operations like max, min, complement, union, intersection.	
8	Program to design fuzzy control system form restaurant tipping problem.	
9	Program to design fuzzy control system form AC temperature.	
10	Program to implement various Genetic operators like crossover, mutation and selection.	
11	Program to implement Genetic Algorithm to maximize the objective function such as f(x)=x^2 where x can have values from 0 to 31.	

PROGRAM 1

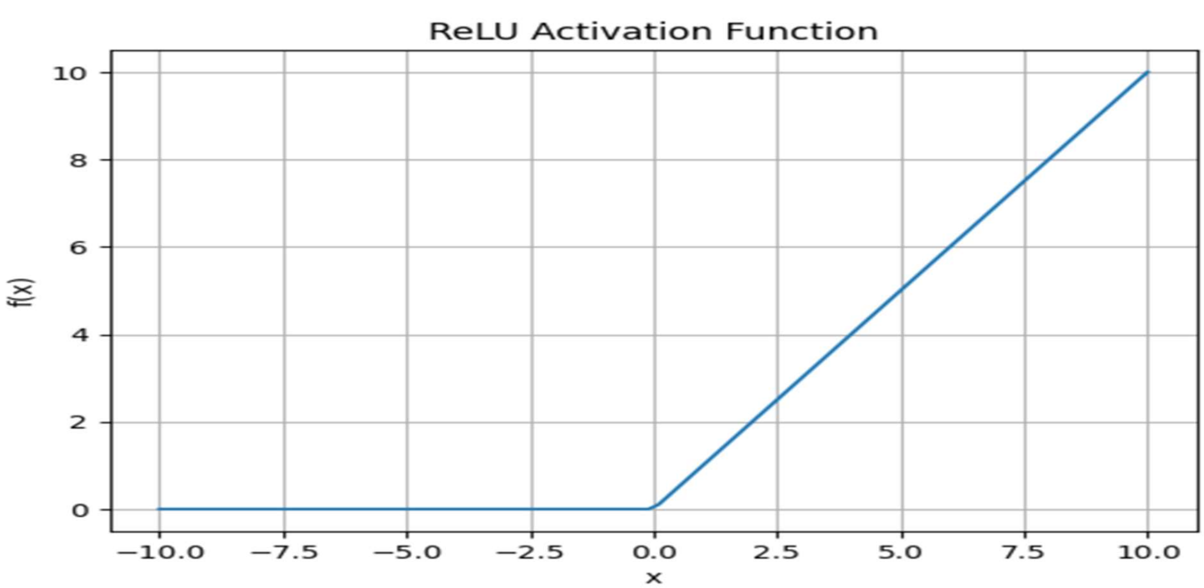
AIM: Program to implement basic activation functions.

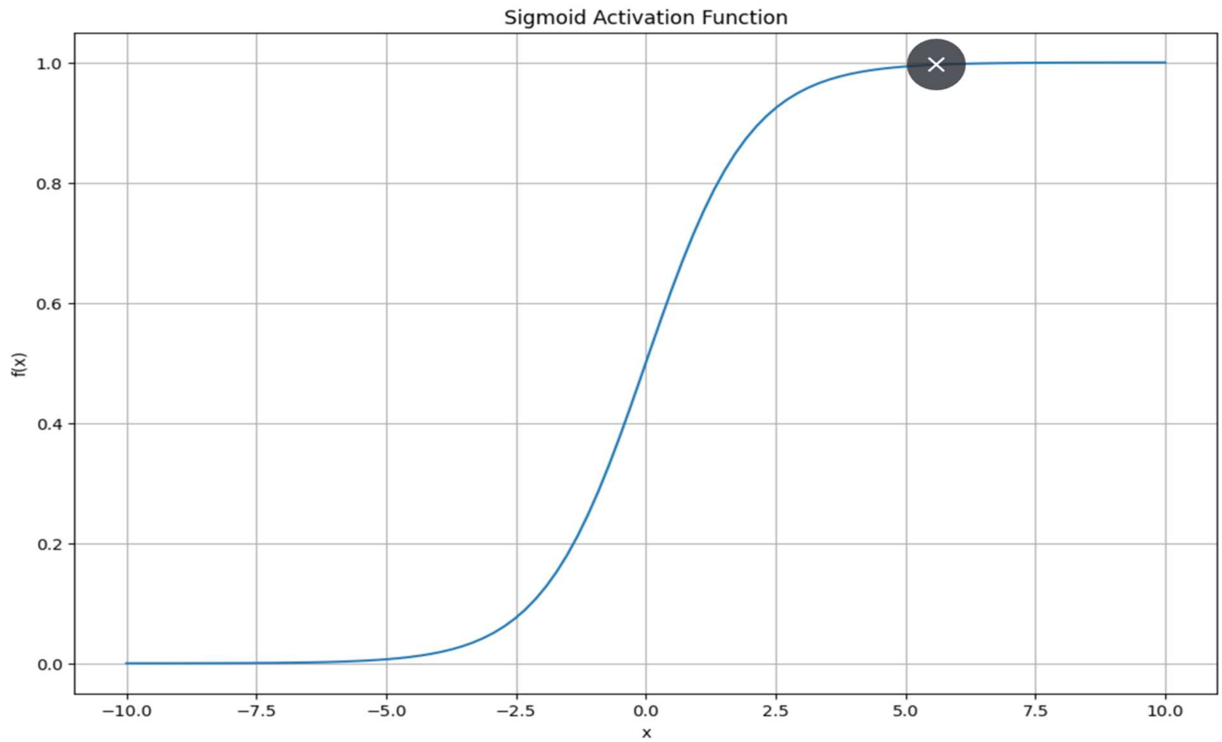
CODE:

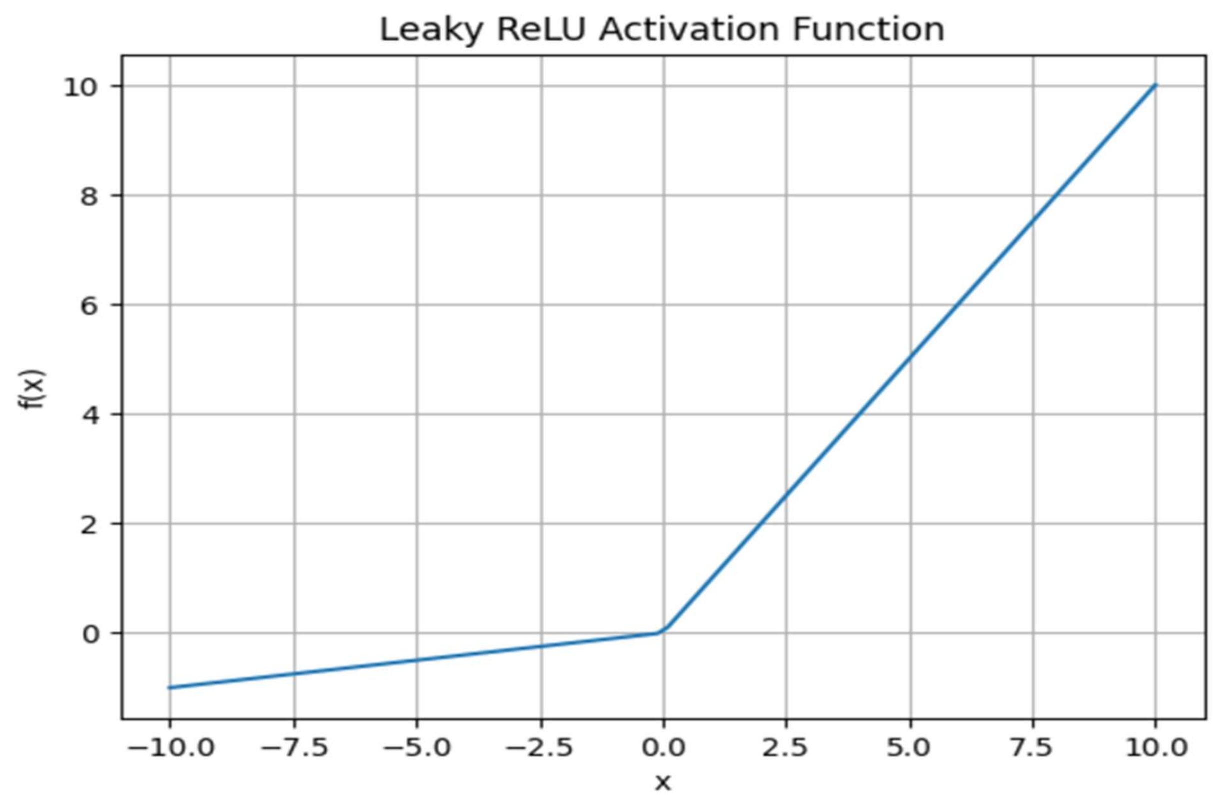
```
import numpy as np
import matplotlib.pyplot as plt
# Activation functions
def sigmoid(x):
    return 1 / (1 + np.exp(-x))
def tanh(x):
    return np.tanh(x)
def relu(x):
    return np.maximum(0, x)
def leaky_relu(x, alpha=0.01):
    return np.where(x > 0, x, alpha * x)
def softmax(x):
    exp_x = np.exp(x - np.max(x))  # Subtract max for numerical stability
    return exp_x / np.sum(exp_x)
# Plotting functions
def plot_activation(func, x, title, **kwargs):
    y = func(x, **kwargs) if kwargs else func(x)
    plt.plot(x, y, label=f'{func.__name__}')
    plt.title(title)
    plt.xlabel('x')
    plt.ylabel('f(x)')
    plt.grid()
    plt.show()
```

```
# Main program
if __name__ == "__main__":
    # Generate input values
    x = np.linspace(-10, 10, 100)

    # Plot activation functions
    plt.figure(figsize=(12, 8))
    plot_activation(sigmoid, x, "Sigmoid Activation Function")
    plot_activation(tanh, x, "Tanh Activation Function")
    plot_activation(relu, x, "ReLU Activation Function")
    plot_activation(leaky_relu, x, "Leaky ReLU Activation Function", alpha=0.1)
    # Softmax example (works on vectors)
    input_vector = np.array([1.0, 2.0, 3.0])
    softmax_result = softmax(input_vector)
    print(f"Softmax Output for {input_vector}: {softmax_result}")
```

OUTPUT:








```
Softmax Output for [1. 2. 3.]: [0.09003057 0.24472847 0.66524096]
```

PROGRAM 2

AIM: Program to implement McCulloch-Pitts Neuron.

CODE:

```
# Function to simulate McCulloch-Pitts neuron
def mcculloch_pitts_neuron(inputs, weights, threshold):
    # Compute the weighted sum of inputs
    weighted_sum = sum(i * w for i, w in zip(inputs, weights))
    # Generate output based on the threshold
    output = 1 if weighted_sum >= threshold else 0
    return output
# Main function
if __name__ == "__main__":
    # Define inputs, weights, and threshold
    inputs = [1, 0, 1]        # Binary inputs (example)
    weights = [2, 1, 2]       # Weights for each input
    threshold = 3             # Threshold value
    # Compute output
    output = mcculloch_pitts_neuron(inputs, weights, threshold)
    # Display results
    print(f"Inputs: {inputs}")
    print(f"Weights: {weights}")
    print(f"Threshold: {threshold}")
    print(f"Output: {output}")
```

OUTPUT:

```
Inputs: [1, 0, 1]
Weights: [2, 1, 2]
Threshold: 3
Output: 1
```

PROGRAM 3

AIM: Program to implement a simple Neuron with various activation functions

CODE:

```
import numpy as np
# Activation functions
def sigmoid(x):
    return 1 / (1 + np.exp(-x)
def tanh(x):
    return np.tanh(x)
def relu(x):
    return np.maximum(0, x)
def leaky_relu(x, alpha=0.01):
    return np.where(x > 0, x, alpha * x)
# Neuron implementation
def simple_neuron(inputs, weights, bias, activation_function):
    # Compute weighted sum + bias
    weighted_sum = np.dot(inputs, weights) + bias
    # Apply activation function
    output = activation_function(weighted_sum)
    return output
# Main program
if __name__ == "__main__":
    # Define inputs, weights, and bias
    inputs = [1.0, 2.0, 3.0]  # Example inputs
    weights = [0.2, 0.8, -0.5]  # Example weights
    bias = 2.0  # Example bias
```

```
# Compute outputs with different activation functions
print("Inputs:", inputs)
print("Weights:", weights)
print("Bias:", bias)
# Using Sigmoid Activation
sigmoid_output = simple_neuron(inputs, weights, bias, sigmoid)
print("Sigmoid Output:", sigmoid_output)
# Using Tanh Activation
tanh_output = simple_neuron(inputs, weights, bias, tanh)
print("Tanh Output:", tanh_output)
# Using ReLU Activation
relu_output = simple_neuron(inputs, weights, bias, relu)
print("ReLU Output:", relu_output)
# Using Leaky ReLU Activation
leaky_relu_output = simple_neuron(inputs, weights, bias, leaky_relu)
print("Leaky ReLU Output:", leaky_relu_output)
```

OUTPUT:

```
Inputs: [1.0, 2.0, 3.0]
Weights: [0.2, 0.8, -0.5]
Bias: 2.0
Sigmoid Output: 0.9088770389851438
Tanh Output: 0.9800963962661914
ReLU Output: 2.3
Leaky ReLU Output: 2.3
```

PROGRAM 4

AIM: Program to implement a simple perceptron using the perceptron learning algorithm

CODE:

```
import numpy as np
class Perceptron:
    def __init__(self, learning_rate=0.1, epochs=100):
        self.learning_rate = learning_rate
        self.epochs = epochs
        self.weights = None
        self.bias = None
    def activation_function(self, weighted_sum):
        return 1 if weighted_sum >= 0 else 0
    def fit(self, X, y):
        # Initialize weights and bias
        n_features = X.shape[1]
        self.weights = np.zeros(n_features)
        self.bias = 0
        for epoch in range(self.epochs):
            for idx, x_i in enumerate(X):
                # Calculate the weighted sum
                weighted_sum = np.dot(x_i, self.weights) + self.bias
                # Predict the output
                y_pred = self.activation_function(weighted_sum)
                # Update weights and bias if there is an error
                error = y[idx] - y_pred
                self.weights += self.learning_rate * error * x_i
```

```
            self.bias += self.learning_rate * error
    def predict(self, X):
        y_pred = [self.activation_function(np.dot(x_i, self.weights) + self.bias) for x_i in X]
        return np.array(y_pred)
# Main program
if __name__ == "__main__":
    # Define the input dataset (AND gate example)
    X = np.array([[0, 0],
                  [0, 1],
                  [1, 0],
                  [1, 1]])  # Input features
    y = np.array([0, 0, 0, 1])  # Target outputs (AND gate)
    # Create a perceptron model
    perceptron = Perceptron(learning_rate=0.1, epochs=10)
    # Train the perceptron
    perceptron.fit(X, y)
    # Test the perceptron
    predictions = perceptron.predict(X)
    print("Predicted outputs:", predictions)
    # Display learned weights and bias
    print("Learned weights:", perceptron.weights)
    print("Learned bias:", perceptron.bias)
```

OUTPUT:

```
Predicted outputs: [0 0 0 1]
Learned weights: [0.2 0.1]
Learned bias: -0.20000000000000004
```

PROGRAM 5

AIM: Program to implement delta and back propagation algorithm.

CODE:

```
import numpy as np

class NeuralNetwork:

    def __init__(self, input_size, hidden_size, output_size):

        # Initialize weights and biases with random values

        self.weights_input_hidden = np.random.randn(input_size, hidden_size)

        self.bias_hidden = np.zeros((1, hidden_size))

        self.weights_hidden_output = np.random.randn(hidden_size, output_size)

        self.bias_output = np.zeros((1, output_size))

        # Hyperparameters

        self.learning_rate = 0.1

    def sigmoid(self, x):

        return 1 / (1 + np.exp(-x))

    def sigmoid_derivative(self, x):

        return x * (1 - x)

    def forward_propagation(self, X):

        # Hidden layer

        self.hidden_sum = np.dot(X, self.weights_input_hidden) + self.bias_hidden

        self.hidden_activation = self.sigmoid(self.hidden_sum)

        # Output layer

        self.output_sum = np.dot(self.hidden_activation, self.weights_hidden_output) +
self.bias_output

        self.output_activation = self.sigmoid(self.output_sum)

        return self.hidden_activation, self.output_activation
```

```
def backpropagation(self, X, y):
        # Forward propagation
        _, predicted = self.forward_propagation(X)
        # Calculate output layer error
        output_error = y - predicted
        output_delta = output_error * self.sigmoid_derivative(predicted)
        # Calculate hidden layer error
        hidden_error = np.dot(output_delta, self.weights_hidden_output.T)
        hidden_delta = hidden_error * self.sigmoid_derivative(self.hidden_activation)
        # Update weights and biases
        # Output layer weights
        self.weights_hidden_output += self.learning_rate * np.dot(self.hidden_activation.T,
output_delta)
        self.bias_output += self.learning_rate * np.sum(output_delta, axis=0, keepdims=True)
        # Hidden layer weights
        self.weights_input_hidden += self.learning_rate * np.dot(X.T, hidden_delta)
        self.bias_hidden += self.learning_rate * np.sum(hidden_delta, axis=0, keepdims=True)
    def train(self, X, y, epochs):
        for _ in range(epochs):
            self.backpropagation(X, y)
    def predict(self, X):
        _, predicted = self.forward_propagation(X)
        return predicted
# Example usage
def main():
    # XOR problem example
    X = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])
    y = np.array([[0], [1], [1], [0]])
```

```
    # Create and train neural network
    nn = NeuralNetwork(input_size=2, hidden_size=4, output_size=1)
    nn.train(X, y, epochs=10000)
    # Make predictions
    predictions = nn.predict(X)
    print("Predictions:")
    print(predictions)
    print("\nTarget:")
    print(y)
if __name__ == "__main__":
    main()
```

OUTPUT:

```
Predictions:
[[0.02905082]
 [0.95278358]
 [0.95520555]
 [0.05588007]]

Target:
[[0]
 [1]
 [1]
 [0]]
```

PROGRAM 6

AIM: Program to implement a simple perceptron with radial basis function (RBF) activation function

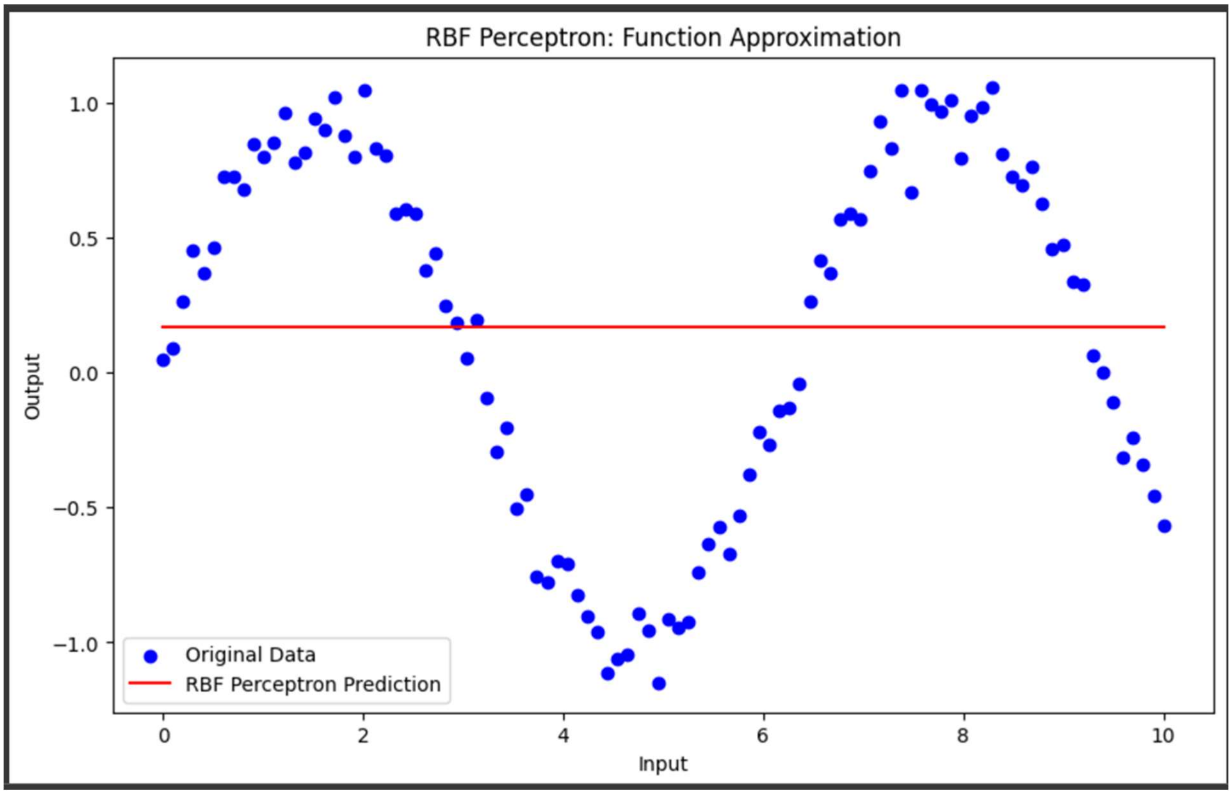
CODE:

```
import numpy as np

import matplotlib.pyplot as plt

class RBFPerceptron:

    def __init__(self, input_size, num_centers):

        # Randomly initialize centers

        self.centers = np.random.randn(num_centers, input_size)

        # Initialize spreads (width of RBF)

        # Spread determines the influence of each center

        self.spreads = np.ones(num_centers)

        # Initialize output layer weights

        self.weights = np.random.randn(num_centers)

    def rbf_activation(self, x):

        # Calculate squared Euclidean distances from centers

        distances = np.sum((self.centers - x)**2, axis=1)

        # Apply RBF (Gaussian) activation

        # exp(-||x - center||^2 / (2 * spread^2))

        activations = np.exp(-distances / (2 * self.spreads**2))

        return activations

    def train(self, X, y, learning_rate=0.01, epochs=100):

        for _ in range(epochs):

            # Compute RBF activations for training samples

            rbf_outputs = np.array([self.rbf_activation(sample) for sample in X])

            # Update weights using least squares
```

```
        self.weights = np.linalg.lstsq(rbf_outputs, y, rcond=None)[0]
        # Optional: Adjust centers and spreads
        self._optimize_centers_and_spreads(X, y)
def _optimize_centers_and_spreads(self, X, y, learning_rate=0.01):
    # Compute current predictions
    current_predictions = self.predict(X)
    # Simple gradient-based center optimization
    for i in range(len(self.centers)):
        # Compute gradient for center position
        center_gradient = np.mean(
            2 * (current_predictions - y) *
            self.weights[i] *
            (X - self.centers[i]) /
            (self.spreads[i]**2)
        )
        # Update center position
        self.centers[i] -= learning_rate * center_gradient
    # Simple gradient-based spread optimization
    for i in range(len(self.spreads)):
        # Compute gradient for spread
        spread_gradient = np.mean(
            2 * (current_predictions - y) *
            self.weights[i] *
            np.sum((X - self.centers[i])**2) /
            (self.spreads[i]**3)
        )
        # Update spread (ensure positive)
        self.spreads[i] = max(0.1, self.spreads[i] - learning_rate * spread_gradient)
```

```
    def predict(self, X):
        # Compute RBF activations for input samples
        rbf_outputs = np.array([self.rbf_activation(sample) for sample in X])
        # Compute weighted sum of RBF activations
        predictions = np.dot(rbf_outputs, self.weights)
        return predictions
def main():
    # Generate synthetic regression data
    np.random.seed(42)
    X = np.linspace(0, 10, 100).reshape(-1, 1)
    y = np.sin(X).ravel() + np.random.normal(0, 0.1, X.shape[0])
    # Create and train RBF Perceptron
    rbf_perceptron = RBFPerceptron(input_size=1, num_centers=10)
    rbf_perceptron.train(X, y, epochs=200)
    # Make predictions
    y_pred = rbf_perceptron.predict(X)
    # Visualize results
    plt.figure(figsize=(10, 6))
    plt.scatter(X, y, color='blue', label='Original Data')
    plt.plot(X, y_pred, color='red', label='RBF Perceptron Prediction')
    plt.title('RBF Perceptron: Function Approximation')
    plt.xlabel('Input')
    plt.ylabel('Output')
    plt.legend()
    plt.show()
if __name__ == "__main__":
    main()
```

OUTPUT:



RBF Perceptron: Function Approximation
1.0
0.5
0.0
−0.5
−1.0
Output
0
2
4
6
8
10
Input
Original Data
RBF Perceptron Prediction

PROGRAM 7

AIM: Program to implement all the fuzzy set operations like max, min, complement, union, intersection.

CODE:

```
import numpy as np
class FuzzySetOperations:
    @staticmethod
    def complement(fuzzy_set):
        return [1 - x for x in fuzzy_set]
    @staticmethod
    def union(fuzzy_set1, fuzzy_set2):
        return [max(x, y) for x, y in zip(fuzzy_set1, fuzzy_set2)]
    @staticmethod
    def intersection(fuzzy_set1, fuzzy_set2):
        return [min(x, y) for x, y in zip(fuzzy_set1, fuzzy_set2)]
    @staticmethod
    def max_operation(fuzzy_set1, fuzzy_set2):
        return [max(x, y) for x, y in zip(fuzzy_set1, fuzzy_set2)]
    @staticmethod
    def min_operation(fuzzy_set1, fuzzy_set2):
        return [min(x, y) for x, y in zip(fuzzy_set1, fuzzy_set2)]
# Main program
if __name__ == "__main__":
    # Example fuzzy sets
    fuzzy_set1 = [0.1, 0.4, 0.7, 1.0]
    fuzzy_set2 = [0.2, 0.6, 0.5, 0.8]
    print("Fuzzy Set 1:", fuzzy_set1)
```

```
print("Fuzzy Set 2:", fuzzy_set2)
# Complement
complement1 = FuzzySetOperations.complement(fuzzy_set1)
print("\nComplement of Fuzzy Set 1:", complement1)
# Union
union_result = FuzzySetOperations.union(fuzzy_set1, fuzzy_set2)
print("Union of Fuzzy Set 1 and Fuzzy Set 2:", union_result)
# Intersection
intersection_result = FuzzySetOperations.intersection(fuzzy_set1, fuzzy_set2)
print("Intersection of Fuzzy Set 1 and Fuzzy Set 2:", intersection_result)
# Max operation
max_result = FuzzySetOperations.max_operation(fuzzy_set1, fuzzy_set2)
print("Max operation between Fuzzy Set 1 and Fuzzy Set 2:", max_result)
# Min operation
min_result = FuzzySetOperations.min_operation(fuzzy_set1, fuzzy_set2)
print("Min operation between Fuzzy Set 1 and Fuzzy Set 2:", min_result)
```

OUTPUT:

```
Fuzzy Set 1: [0.1, 0.4, 0.7, 1.0]
Fuzzy Set 2: [0.2, 0.6, 0.5, 0.8]

Complement of Fuzzy Set 1: [0.9, 0.6, 0.30000000000000004, 0.0]
Union of Fuzzy Set 1 and Fuzzy Set 2: [0.2, 0.6, 0.7, 1.0]
Intersection of Fuzzy Set 1 and Fuzzy Set 2: [0.1, 0.4, 0.5, 0.8]
Max operation between Fuzzy Set 1 and Fuzzy Set 2: [0.2, 0.6, 0.7, 1.0]
Min operation between Fuzzy Set 1 and Fuzzy Set 2: [0.1, 0.4, 0.5, 0.8]
```

PROGRAM 8

AIM: Program to design fuzzy control system form restaurant tipping problem.

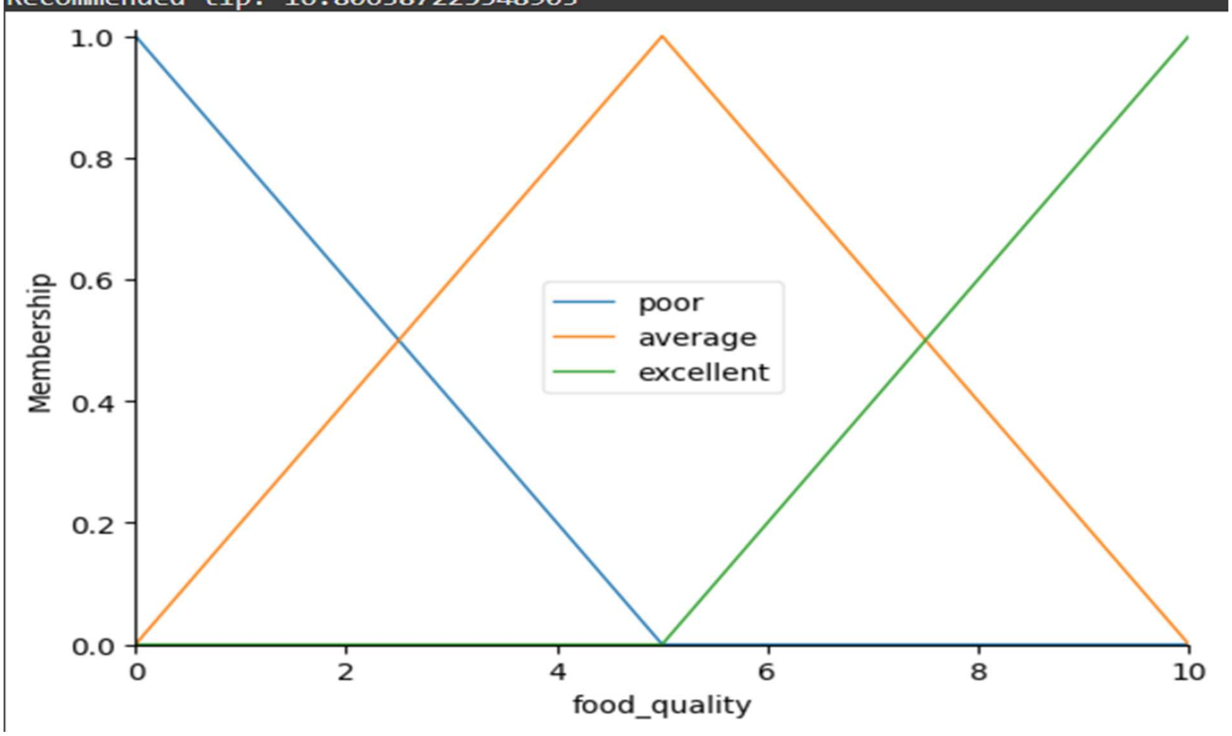
CODE:

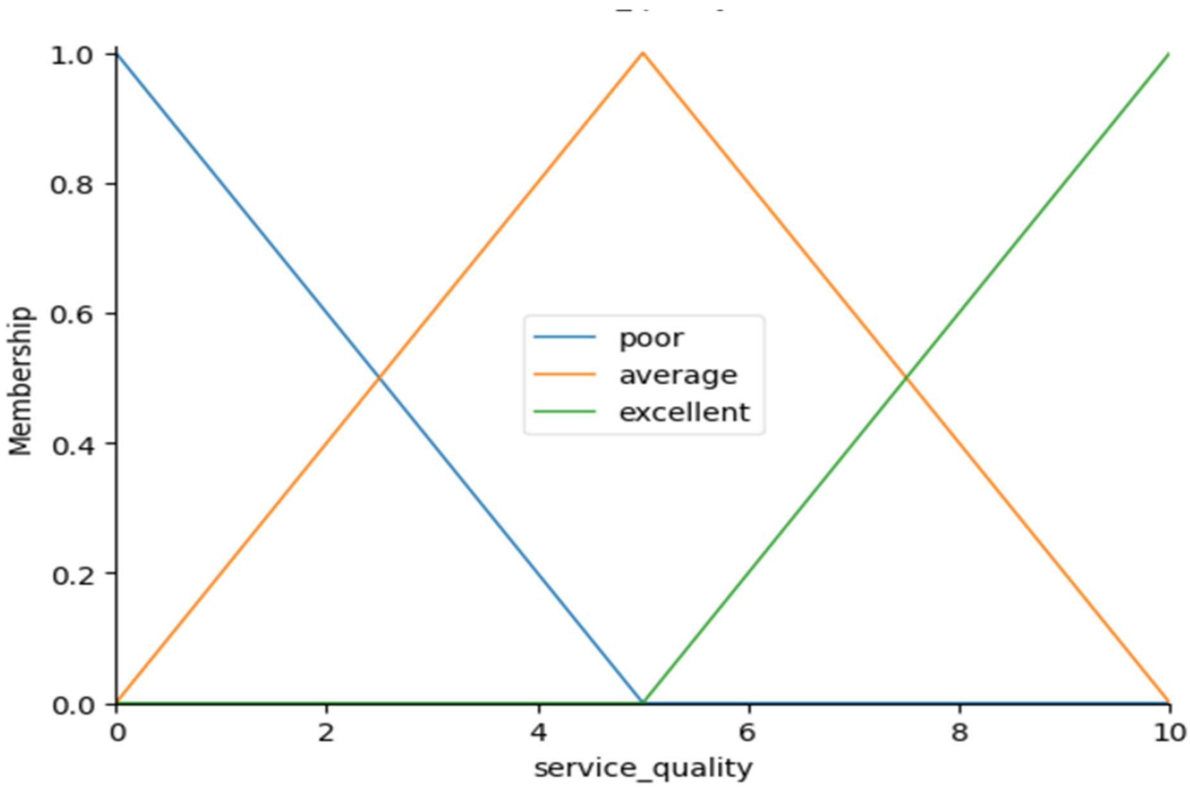
```
#Code 2
import numpy as np
import skfuzzy as fuzz
from skfuzzy import control as ctrl
# Step 1: Define fuzzy variables
food_quality = ctrl.Antecedent(np.arange(0, 11, 1), 'food_quality')
service_quality = ctrl.Antecedent(np.arange(0, 11, 1), 'service_quality')
tip = ctrl.Consequent(np.arange(0, 26, 1), 'tip')
# Step 2: Define fuzzy membership functions
food_quality['poor'] = fuzz.trimf(food_quality.universe, [0, 0, 5])
food_quality['average'] = fuzz.trimf(food_quality.universe, [0, 5, 10])
food_quality['excellent'] = fuzz.trimf(food_quality.universe, [5, 10, 10])
service_quality['poor'] = fuzz.trimf(service_quality.universe, [0, 0, 5])
service_quality['average'] = fuzz.trimf(service_quality.universe, [0, 5, 10])
service_quality['excellent'] = fuzz.trimf(service_quality.universe, [5, 10, 10])
tip['low'] = fuzz.trimf(tip.universe, [0, 0, 13])
tip['medium'] = fuzz.trimf(tip.universe, [0, 13, 25])
tip['high'] = fuzz.trimf(tip.universe, [13, 25, 25])
# Step 3: Define fuzzy rules
rule1 = ctrl.Rule(food_quality['poor'] | service_quality['poor'], tip['low'])
rule2 = ctrl.Rule(service_quality['average'], tip['medium'])
rule3 = ctrl.Rule(food_quality['excellent'] | service_quality['excellent'], tip['high'])
# Step 4: Create the control system
tipping_ctrl = ctrl.ControlSystem([rule1, rule2, rule3])
```

```
tipping = ctrl.ControlSystemSimulation(tipping_ctrl)
# Step 5: Provide inputs
tipping.input['food_quality'] = 6.5  # Input: Food quality score
tipping.input['service_quality'] = 9.0  # Input: Service quality score
# Step 6: Perform fuzzy inference
tipping.compute()
# Output the result
print("Recommended tip:", tipping.output['tip'])
# Optional: Visualize the results
food_quality.view()
service_quality.view()
tip.view(sim=tipping)
```

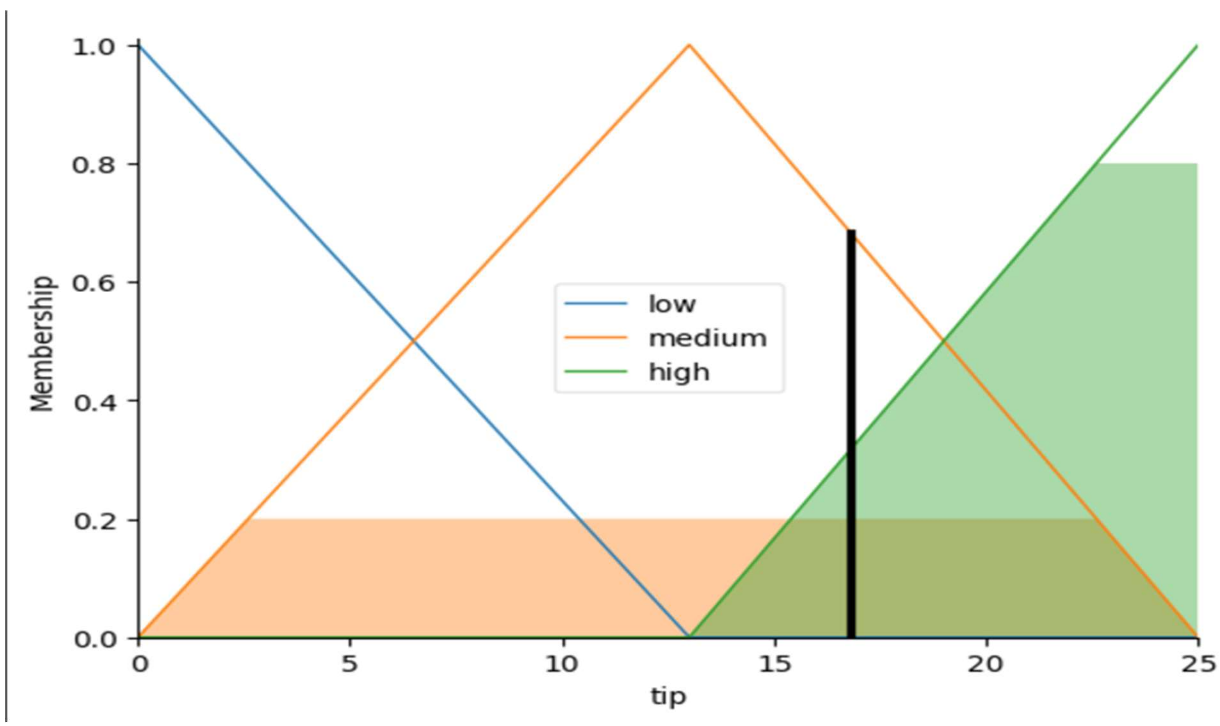
OUTPUT:

```
Recommended tip: 16.806387225548903
```





1.0
0.8
0.6
0.4
0.2
0.0
Membership
0
2
4
6
8
10
service_quality
poor
average
excellent



1.0
0.8
0.6
0.4
0.2
0.0
Membership
0
5
10
15
20
25
tip
low
medium
high

PROGRAM 9

AIM: Program to design fuzzy control system form AC temperature.

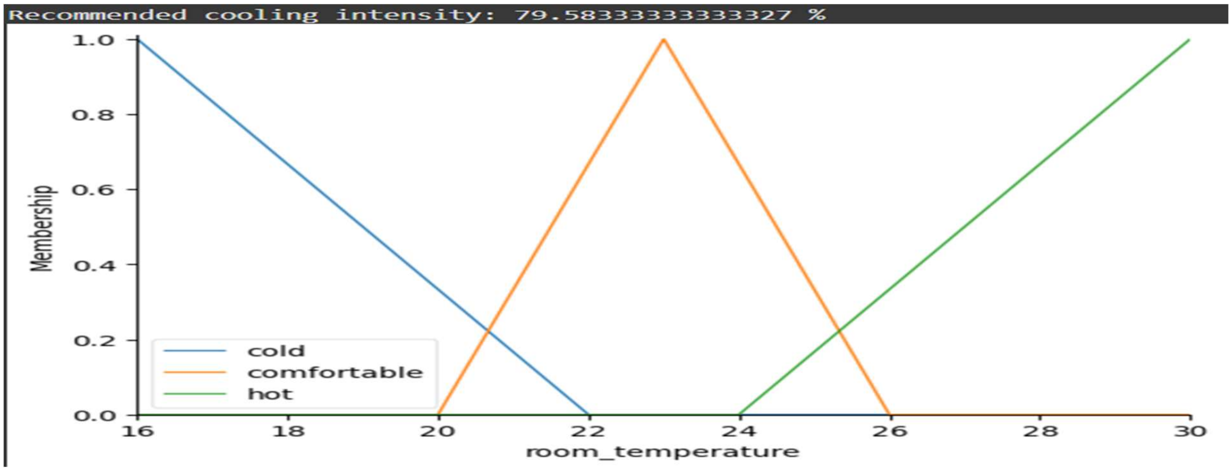
CODE:

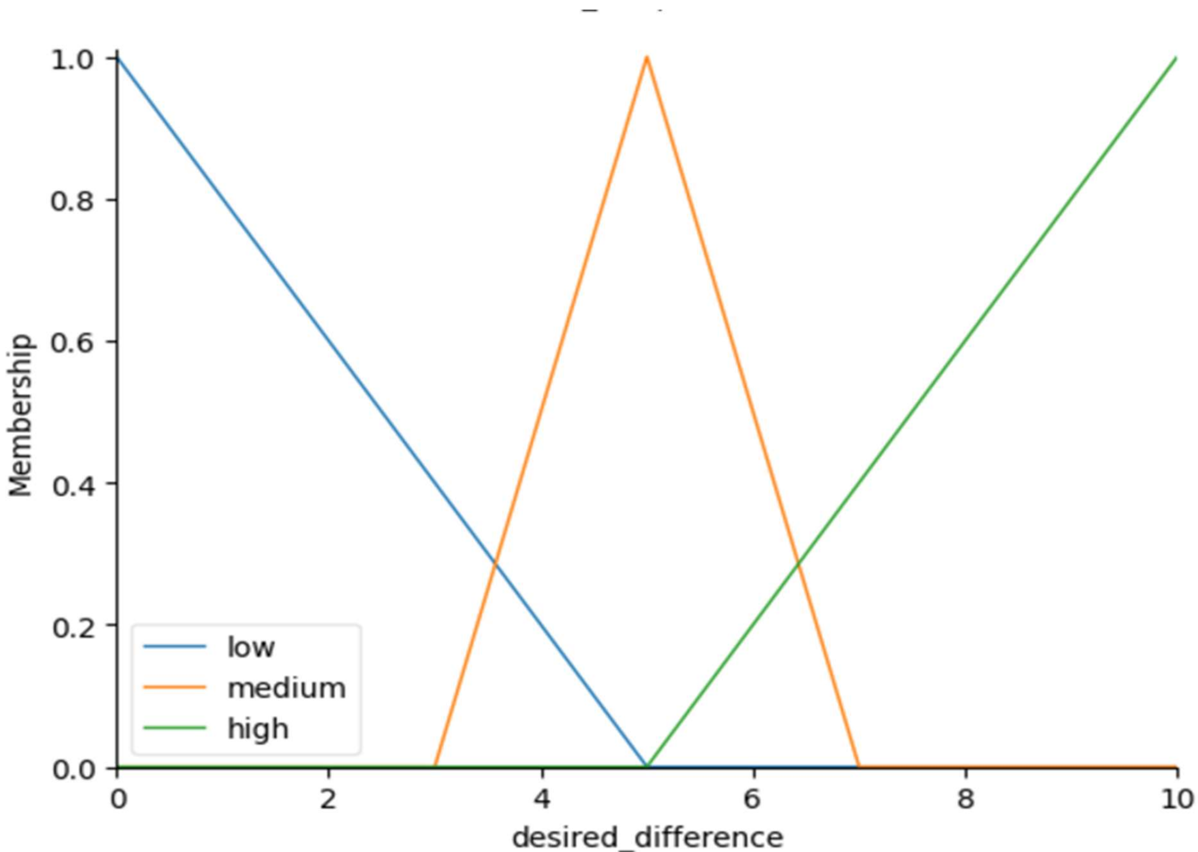
```
import numpy as np

import skfuzzy as fuzz

from skfuzzy import control as ctrl

# Step 1: Define fuzzy variables

room_temperature = ctrl.Antecedent(np.arange(16, 31, 1), 'room_temperature')  # Room
temperature in °C

desired_difference = ctrl.Antecedent(np.arange(0, 11, 1), 'desired_difference')  # Desired
temperature difference

cooling = ctrl.Consequent(np.arange(0, 101, 1), 'cooling')  # Cooling intensity in %

# Step 2: Define fuzzy membership functions

# Room temperature

room_temperature['cold'] = fuzz.trimf(room_temperature.universe, [16, 16, 22])

room_temperature['comfortable'] = fuzz.trimf(room_temperature.universe, [20, 23, 26])

room_temperature['hot'] = fuzz.trimf(room_temperature.universe, [24, 30, 30])

# Desired temperature difference

desired_difference['low'] = fuzz.trimf(desired_difference.universe, [0, 0, 5])

desired_difference['medium'] = fuzz.trimf(desired_difference.universe, [3, 5, 7])

desired_difference['high'] = fuzz.trimf(desired_difference.universe, [5, 10, 10])

# Cooling intensity

cooling['low'] = fuzz.trimf(cooling.universe, [0, 0, 50])

cooling['medium'] = fuzz.trimf(cooling.universe, [30, 50, 70])

cooling['high'] = fuzz.trimf(cooling.universe, [50, 100, 100])

# Step 3: Define fuzzy rules

rule1 = ctrl.Rule(room_temperature['cold'] & desired_difference['low'], cooling['low'])

rule2 = ctrl.Rule(room_temperature['cold'] & desired_difference['medium'], cooling['low'])
```

```
rule3 = ctrl.Rule(room_temperature['comfortable'] & desired_difference['low'], cooling['low'])

rule4 = ctrl.Rule(room_temperature['comfortable'] & desired_difference['medium'],
cooling['medium'])

rule5 = ctrl.Rule(room_temperature['hot'] & desired_difference['medium'], cooling['medium'])

rule6 = ctrl.Rule(room_temperature['hot'] & desired_difference['high'], cooling['high'])

# Step 4: Create and simulate the control system

ac_control = ctrl.ControlSystem([rule1, rule2, rule3, rule4, rule5, rule6])

ac_simulation = ctrl.ControlSystemSimulation(ac_control)

# Step 5: Provide inputs

ac_simulation.input['room_temperature'] = 28  # Current room temperature

ac_simulation.input['desired_difference'] = 7  # Desired temperature difference

# Step 6: Perform fuzzy inference

ac_simulation.compute()

# Output the result

print("Recommended cooling intensity:", ac_simulation.output['cooling'], "%")

# Optional: Visualize the results

room_temperature.view()

desired_difference.view()

cooling.view(sim=ac_simulation)
```

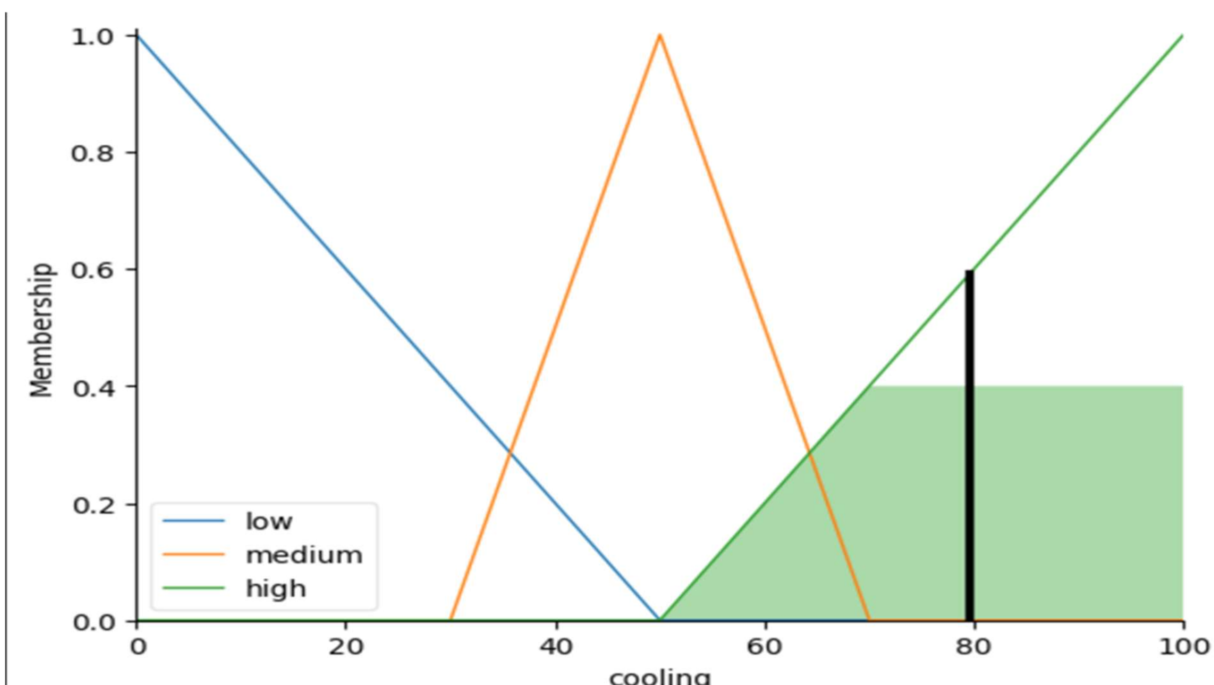
OUTPUT:

Recommended cooling intensity: 79.58333333333327 %





Membership
1.0
0.8
0.6
0.4
0.2
0.0
0
2
4
6
8
10
desired_difference
low
medium
high



Membership
1.0
0.8
0.6
0.4
0.2
0.0
0
20
40
60
80
100
cooling
low
medium
high

PROGRAM 10

AIM: Program to implement various Genetic operators like crossover, mutation and selection.

CODE:

```
import random

# Target solution

TARGET = "1010101010"

POPULATION_SIZE = 6

GENE_LENGTH = len(TARGET)

MUTATION_RATE = 0.1  # Probability of mutation (10%)

# Generate initial population

def generate_population(size, gene_length):

    return [''.join(random.choice("01") for _ in range(gene_length)) for _ in range(size)]

# Fitness function: Count matching bits with the target

def fitness(individual):

    return sum(1 for i, j in zip(individual, TARGET) if i == j)

# Selection: Roulette wheel selection

def selection(population, fitness_scores):

    total_fitness = sum(fitness_scores)

    probabilities = [score / total_fitness for score in fitness_scores]

    selected = random.choices(population, weights=probabilities, k=2)

    return selected

# Crossover: Single-point crossover

def crossover(parent1, parent2):

    crossover_point = random.randint(1, GENE_LENGTH - 1)

    offspring1 = parent1[:crossover_point] + parent2[crossover_point:]

    offspring2 = parent2[:crossover_point] + parent1[crossover_point:]
```

```
    return offspring1, offspring2
# Mutation: Flip a random bit
def mutate(individual):
    individual = list(individual)
    for i in range(len(individual)):
        if random.random() < MUTATION_RATE:
            individual[i] = '1' if individual[i] == '0' else '0'
    return ''.join(individual)
# Main Genetic Algorithm function
def genetic_algorithm():
    # Generate initial population
    population = generate_population(POPULATION_SIZE, GENE_LENGTH)
    generation = 0
    print(f"Generation {generation}: {population}")
    while True:
        # Evaluate fitness
        fitness_scores = [fitness(individual) for individual in population]
        # Check for target match
        if TARGET in population:
            print(f"Solution found in generation {generation}: {TARGET}")
            break
        # Selection
        new_population = []
        for _ in range(POPULATION_SIZE // 2):  # Produce pairs of offspring
            parent1, parent2 = selection(population, fitness_scores)
            # Crossover
            offspring1, offspring2 = crossover(parent1, parent2)
            # Mutation
```

```
            offspring1 = mutate(offspring1)
            offspring2 = mutate(offspring2)

            new_population.extend([offspring1, offspring2])
        # Update population
        population = new_population
        generation += 1
        print(f"Generation {generation}: {population}")
# Run the Genetic Algorithm
if __name__ == "__main__":
    genetic_algorithm()
```

OUTPUT:

```
Generation 0: ['0101100101', '0101000011', '1011100100', '1001101110', '0011101001', '1011011100']
Generation 1: ['1011111100', '1011011100', '0101000110', '1011100110', '1001101001', '0001101110']
Generation 2: ['1011011100', '1111011100', '1001101000', '0001101111', '1101100110', '0101000111']
Generation 3: ['1111000111', '0101011100', '1101001110', '1011011100', '0001000110', '1101101011']
Generation 4: ['0101011111', '1101101100', '1010010100', '1011011100', '1111000110', '1101001011']
Generation 5: ['0101101100', '1111001011', '1010011100', '1011010100', '1111101111', '1101001000']
Generation 6: ['1011111101', '1101001000', '1010001100', '1010001100', '1111001111', '1111101001']
Generation 7: ['1100111000', '1111001101', '1010000100', '1010000100', '1110001110', '1010001101']
Generation 8: ['0011001100', '1111000111', '1110010100', '0010001010', '1110000101', '1010000111']
Generation 9: ['1011000111', '1110011100', '1110100111', '1010000111', '0010001110', '0011011010']
Generation 10: ['1010011100', '1110000111', '0110010011', '1011000111', '1011000111', '1110100111']
Generation 11: ['0110010100', '1110011011', '1011000111', '1010100111', '1110110011', '0111101111']
Generation 12: ['1010101011', '1010100011', '1011110011', '1110000110', '1010100111', '0010100110']
Generation 13: ['1011100110', '0010110011', '1010100110', '0010110110', '1011111001', '1111110010']
Generation 14: ['1011100110', '0010100110', '1010100010', '1011110100', '0010110110', '0000100011']
Generation 15: ['1001100010', '1010100011', '1011100110', '0101100011', '1010100110', '1010110100']
Generation 16: ['1001100010', '1011100110', '1010110100', '1010100001', '1001100010', '1011000110']
Generation 17: ['1100100001', '0010110100', '1011000010', '1001100011', '1010001100', '1001100001']
Generation 18: ['1001100011', '1001100011', '1010100101', '1100100100', '0001100011', '1000010100']
Generation 19: ['0011000011', '1001100010', '1001101100', '1010010111', '0001100011', '1001100011']
Generation 20: ['0011100011', '1001000011', '1000000001', '0011100100', '0000100111', '1101100011']
Generation 21: ['1000000011', '0111100000', '1000100111', '0011100111', '0001100111', '1000000001']
Generation 22: ['0000010011', '1011110111', '0011100111', '0011100111', '1000000011', '1000000001']
Generation 23: ['0011010001', '1000100111', '0011100011', '1000001111', '0000010011', '1011110011']
Generation 24: ['0000010011', '0000100111', '1000001110', '1000010001', '1000001111', '1111110011']
Generation 25: ['1111101110', '1000010011', '0010010111', '0000100111', '1000001111', '1000001110']
Generation 26: ['0010010110', '1000000110', '1000010011', '0000110111', '1010010111', '0000001110']
Generation 27: ['0000001110', '0000001010', '0100000110', '1101001110', '1000001110', '0000010011']
Generation 28: ['0000001010', '1011101010', '0000101010', '0000001010', '0000101110', '0000001010']
Generation 29: ['0000001000', '1010101010', '0000100010', '0000001010', '0010001010', '0000001010']
Solution found in generation 29: 1010101010
```

PROGRAM 11

AIM: Program to implement Genetic Algorithm to maximize the objective function such as f(x)=x^2 where x can have values from 0 to 31.

CODE:

```
import random
# Parameters
POPULATION_SIZE = 6  # Number of individuals in the population
CHROMOSOME_LENGTH = 5  # Binary representation of x (5 bits for values 0 to 31)
MUTATION_RATE = 0.1  # Probability of mutation (10%)
GENERATIONS = 15  # Maximum number of generations
# Objective function
def objective_function(x):
    return x**2
# Generate initial population
def generate_population(size, length):
    return [''.join(random.choice("01") for _ in range(length)) for _ in range(size)]
# Decode binary chromosome to integer value
def decode(chromosome):
    return int(chromosome, 2)
# Fitness function
def fitness(chromosome):
    x = decode(chromosome)
    return objective_function(x)
# Selection: Roulette wheel selection
def selection(population, fitness_scores):
    total_fitness = sum(fitness_scores)
    probabilities = [score / total_fitness for score in fitness_scores]
```

```
    return random.choices(population, weights=probabilities, k=2)
# Crossover: Single-point crossover
def crossover(parent1, parent2):
    point = random.randint(1, CHROMOSOME_LENGTH - 1)
    offspring1 = parent1[:point] + parent2[point:]
    offspring2 = parent2[:point] + parent1[point:]
    return offspring1, offspring
# Mutation: Flip a random bit
def mutate(chromosome):
    chromosome = list(chromosome)
    for i in range(len(chromosome)):
        if random.random() < MUTATION_RATE:
            chromosome[i] = '1' if chromosome[i] == '0' else '0'
    return ''.join(chromosome)
# Main Genetic Algorithm function
def genetic_algorithm():
    # Generate initial population
    population = generate_population(POPULATION_SIZE, CHROMOSOME_LENGTH)
    generation = 0
    print(f"Generation {generation}: {population}")
    for _ in range(GENERATIONS):
        # Evaluate fitness
        fitness_scores = [fitness(chromosome) for chromosome in population]
        # Find and print the best solution
        best_chromosome = max(population, key=lambda chrom: fitness(chrom))
        best_x = decode(best_chromosome)
        best_fitness = fitness(best_chromosome)
        print(f"Best solution in generation {generation}: x = {best_x}, f(x) = {best_fitness}")
```

```
        # Selection
        new_population = []
        for _ in range(POPULATION_SIZE // 2):  # Produce pairs of offspring
            parent1, parent2 = selection(population, fitness_scores)
            # Crossover
            offspring1, offspring2 = crossover(parent1, parent2)
            # Mutation
            offspring1 = mutate(offspring1)
            offspring2 = mutate(offspring2)
            new_population.extend([offspring1, offspring2])
        # Update population
        population = new_population
        generation += 1
# Run the Genetic Algorithm
if __name__ == "__main__":
    genetic_algorithm()
```

OUTPUT:

```
Generation 0: ['10001', '10111', '01011', '01010', '00101', '11111']
Best solution in generation 0: x = 31, f(x) = 961
Best solution in generation 1: x = 31, f(x) = 961
Best solution in generation 2: x = 31, f(x) = 961
Best solution in generation 3: x = 31, f(x) = 961
Best solution in generation 4: x = 29, f(x) = 841
Best solution in generation 5: x = 31, f(x) = 961
Best solution in generation 6: x = 31, f(x) = 961
Best solution in generation 7: x = 30, f(x) = 900
Best solution in generation 8: x = 31, f(x) = 961
Best solution in generation 9: x = 31, f(x) = 961
Best solution in generation 10: x = 30, f(x) = 900
Best solution in generation 11: x = 30, f(x) = 900
Best solution in generation 12: x = 30, f(x) = 900
Best solution in generation 13: x = 31, f(x) = 961
Best solution in generation 14: x = 27, f(x) = 729
```