CONTENT

EXP1	Implementation of fuzzy control inference system	DATE:

AIM: Understand the concept of fuzzy control inference system using python programming language.

Algorithm:
Step 1: Define Fuzzy Sets input and output variables.
Step 2: Create Fuzzy Rules
Step 3: Perform Fuzzy Inference
Step 4: Defuzzify the output fuzzy sets to obtain a crisp output value.
Step 5: Use the defuzzified output as the control action.
Step 6: Implement Control Action.
Step 7: Repeat the above steps in a loop as needed for real-time control.
End of the fuzzy control algorithm.

First, you'll need to install the scikit-fuzzy library if you haven't already. You can install it using the following command:

```
pip install scikit-fuzzy
```

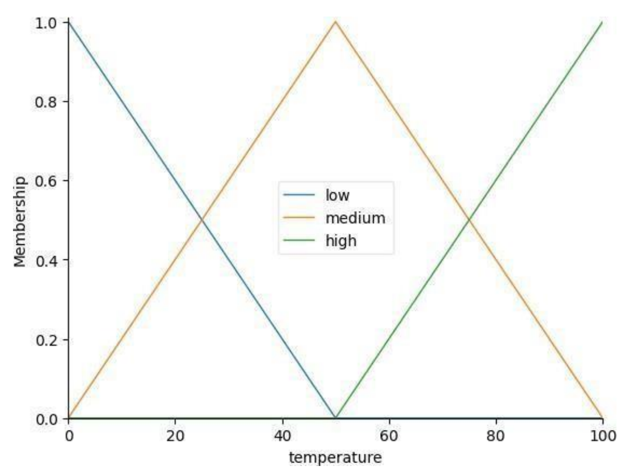
Now, let's implement the fuzzy inference system:

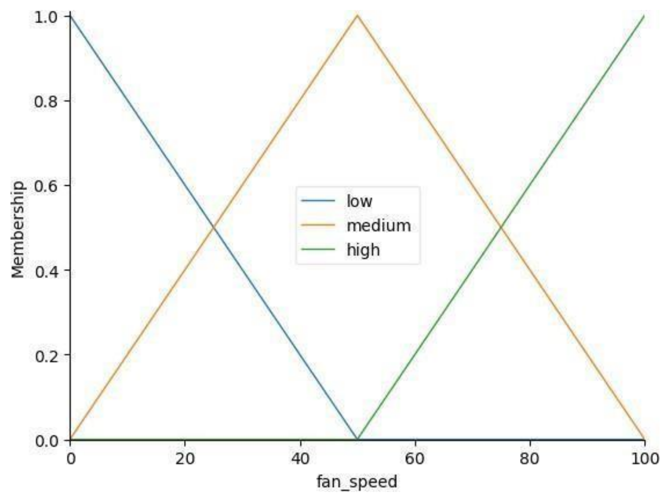
PROGRAM:

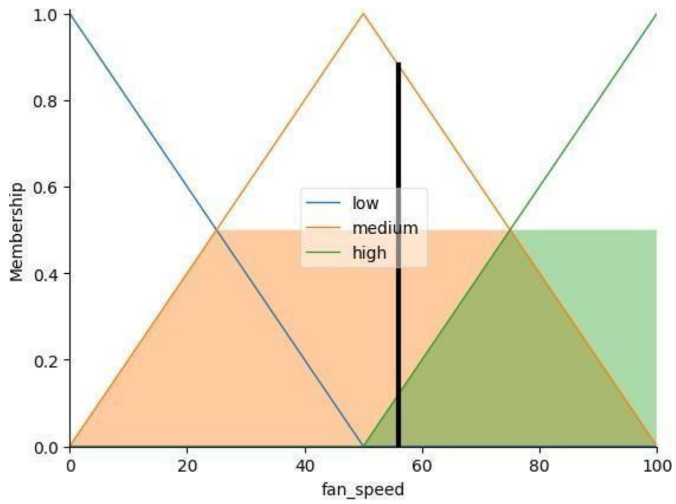
```
import numpy as np
import skfuzzy as fuzz
from skfuzzy import control as ctrl

# Create Antecedent/Consequent objects for temperature and fan speed
temperature = ctrl.Antecedent(np.arange(0, 101, 1), 'temperature')
fan_speed = ctrl.Consequent(np.arange(0, 101, 1), 'fan_speed')

# Define membership functions for temperature
temperature['low'] = fuzz.trimf(temperature.universe, [0, 0, 50])
temperature['medium'] = fuzz.trimf(temperature.universe, [0, 50, 100])
temperature['high'] = fuzz.trimf(temperature.universe, [50, 100, 100])

# Define membership functions for fan speed
fan_speed['low'] = fuzz.trimf(fan_speed.universe, [0, 0, 50])
fan_speed['medium'] = fuzz.trimf(fan_speed.universe, [0, 50, 100])
fan_speed['high'] = fuzz.trimf(fan_speed.universe, [50, 100, 100])

# Define fuzzy rules
rule1 = ctrl.Rule(temperature['low'], fan_speed['low'])
```

```
rule2 = ctrl.Rule(temperature['medium'], fan_speed['medium'])
rule3 = ctrl.Rule(temperature['high'], fan_speed['high'])

# Create control system and add rules
fan_ctrl = ctrl.ControlSystem([rule1, rule2, rule3])
fan_speed_ctrl = ctrl.ControlSystemSimulation(fan_ctrl)

# Input the temperature value
temperature_value = 75

# Pass the input to the control system
fan_speed_ctrl.input['temperature'] = temperature_value

# Compute the result
fan_speed_ctrl.compute()


# Print the output
print("Fan Speed:", fan_speed_ctrl.output['fan_speed'])
# Plot membership functions and output
temperature.view()
fan_speed.view()
fan_speed.view(sim=fan_speed_ctrl)
```

Output:







Result: Thus the above program for fuzzy control interface system executed successfully with desired output.

EXP 2	Programming exercise on classification with a discrete perceptron	DATE:

AIM: Understand the concept of classification with discrete perceptron using python programming language.

Algorithm:
Step 1: Initialize weights W and bias b to small random values
Step 2: Define learning rate
Step 3: Define the number of training epochs
Step 4: Define the training data (features and labels
Step 5: Define the perceptron training algorithm
Step 6: The perceptron is now trained, and you can use it to make predictions

PROGRAM:

```
import numpy as np

class DiscretePerceptron:
    def___init_(self, input_size):
        self.weights = np.zeros(input_size)
        self.bias = 0

    def predict(self, inputs):
        activation = np.dot(self.weights, inputs) + self.bias
        return 1 if activation > 0 else 0

    def train(self, inputs, target, learning_rate=0.1, epochs=100):
        for _ in range(epochs):
            for x, y in zip(inputs, target):
                prediction = self.predict(x)
                error = y - prediction
                self.weights += learning_rate * error * x
                self.bias += learning_rate * error

def main():
    # Generate some example data points for two classes
    class_0 = np.array([[2, 3], [3, 2], [1, 1]])
    class_1 = np.array([[5, 7], [6, 8], [7, 6]])

    # Combine the data points and create labels (0 for class 0, 1 for class 1)
    inputs = np.vstack((class_0, class_1))
    targets = np.array([0, 0, 0, 1, 1, 1])

    # Create a discrete perceptron with input size 2
    perceptron = DiscretePerceptron(input_size=2)
```

```
    # Train the perceptron
    perceptron.train(inputs, targets)

    # Test the trained perceptron with new data
    test_data = np.array([[4, 5], [2, 2]])
    for data in test_data:
        prediction = perceptron.predict(data)
        if prediction == 0:
            print(f"Data {data} belongs to class 0")
        else:
            print(f"Data {data} belongs to class 1")

if __name__ == "__main__":
    main()
```

Output:

Data [4 5] belongs to class 1
Data [2 2] belongs to class 0

Result: Thus the above program classification with discrete perceptron executed successfully with desired output.

EXP 3	Implementation of XOR with backpropagation algorithm	DATE:

AIM: Understand the concept of XOR with backpropagation algorithm using python programing language.

Algorithm:

1. Initialize the neural network with random weights and biases.
2. Define the training data for XOR
3. Set hyperparameters:
   Learning rate (alpha)
   Number of epochs (iterations)
   Number of hidden layers and neurons per layer
   Activation function (e.g., sigmoid)
4. Repeat for each epoch:
   a. Initialize the total error for this epoch to 0.
   b. For each training example in the dataset:
      i. Forward propagation:
         - ✓ Compute the weighted sum of inputs and biases for each neuron in the hidden layer(s) and output layer.
         - ✓ Apply the activation function to each neuron's output.
      ii. Compute the error between the predicted output and the actual output for the current training example.
      iii. Update the total error for this epoch with the squared error from step ii.
      iv. Backpropagation:
         - ✓ Compute the gradient of the error with respect to the output layer neurons.
         - ✓ Backpropagate the gradients through the hidden layers.
         - ✓ Update the weights and biases using the gradients and the learning rate.
   c. Calculate the average error for this epoch by dividing the total error by the number of training examples.
   d. Check if the average error is below a predefined threshold or if the desired accuracy is reached.
   - If yes, exit the training loop.
5. Once training is complete, you can use the trained neural network to predict XOR values for new inputs.
6. End.

<u>PROGRAM:</u>

```
import numpy as np

# Define sigmoid activation function and its derivative
def sigmoid(x):
    return 1 / (1 + np.exp(-x))

def sigmoid_derivative(x):
```

```
    return x * (1 - x)

# XOR input and target data
input_data = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])
target_data = np.array([[0], [1], [1], [0]])

# Neural network architecture
input_size = 2
hidden_size = 2
output_size = 1
learning_rate = 0.1
epochs = 10000

# Initialize weights randomly with mean 0
hidden_weights = np.random.uniform(size=(input_size, hidden_size))
output_weights = np.random.uniform(size=(hidden_size, output_size))

# Training loop
for _ in range(epochs):
    # Forward propagation
    hidden_layer_activation = np.dot(input_data, hidden_weights)
    hidden_layer_output = sigmoid(hidden_layer_activation)

    output_layer_activation = np.dot(hidden_layer_output, output_weights)
    predicted_output = sigmoid(output_layer_activation)

    # Calculate error
    error = target_data - predicted_output

    # Backpropagation
    output_delta = error * sigmoid_derivative(predicted_output)

    hidden_layer_error = output_delta.dot(output_weights.T)
    hidden_layer_delta = hidden_layer_error * sigmoid_derivative(hidden_layer_output)

    # Update weights
    output_weights += hidden_layer_output.T.dot(output_delta) * learning_rate
    hidden_weights += input_data.T.dot(hidden_layer_delta) * learning_rate

# Test the trained network
test_data = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])
for data in test_data:
    hidden_layer_activation = np.dot(data, hidden_weights)
    hidden_layer_output = sigmoid(hidden_layer_activation)

    output_layer_activation = np.dot(hidden_layer_output, output_weights)
    predicted_output = sigmoid(output_layer_activation)

    print(f"Input: {data} Predicted Output: {predicted_output[0]}")
```

Output:

Input: [0 0] Predicted Output: 0.287381655624125
Input: [0 1] Predicted Output: 0.6696713061093961
Input: [1 0] Predicted Output: 0.6697648563700653
Input: [1 1] Predicted Output: 0.42466198065447125

Result: Thus the above program classification with discrete perception executed successfully with desired output.

EXP 4	Implementation of self-organizing maps for a specific application.	DATE:

AIM: Understand the concept of self-organizing maps for a specific application using python programming language.

Algorithm:

1. Initialize the SOM:
   - Define the size and shape of the SOM grid (e.g., rows and columns).
   - Initialize the weight vectors for each neuron in the grid with random values.
   - Define the learning rate and initial neighborhood radius.

2. Define the training dataset:
   - Input data for the SOM, often a set of high-dimensional vectors.

3. Train the SOM:
   - For each epoch (iteration):
     a. Randomly select a data point from the training dataset.
     b. Find the Best Matching Unit (BMU), the neuron with the weight vector closest to the input.
     c. Update the BMU's weights and the weights of its neighbors:
       - Calculate the influence on the neighboring neurons based on the neighborhood radius.
       - Update the weights using the learning rate and the difference between the input and the BMU's weight.
     d. Decrease the learning rate and neighborhood radius over time (e.g., using a decay function).

4. Repeat the training process until convergence (or a predetermined number of epochs).

5. Map new or unseen data:
   - Given a new input vector, find the BMU based on the current SOM weights.
   - Use the BMU's location to make decisions or predictions based on your specific application.

6. Visualization (optional):
   - Visualize the trained SOM grid to understand the data distribution and clustering.

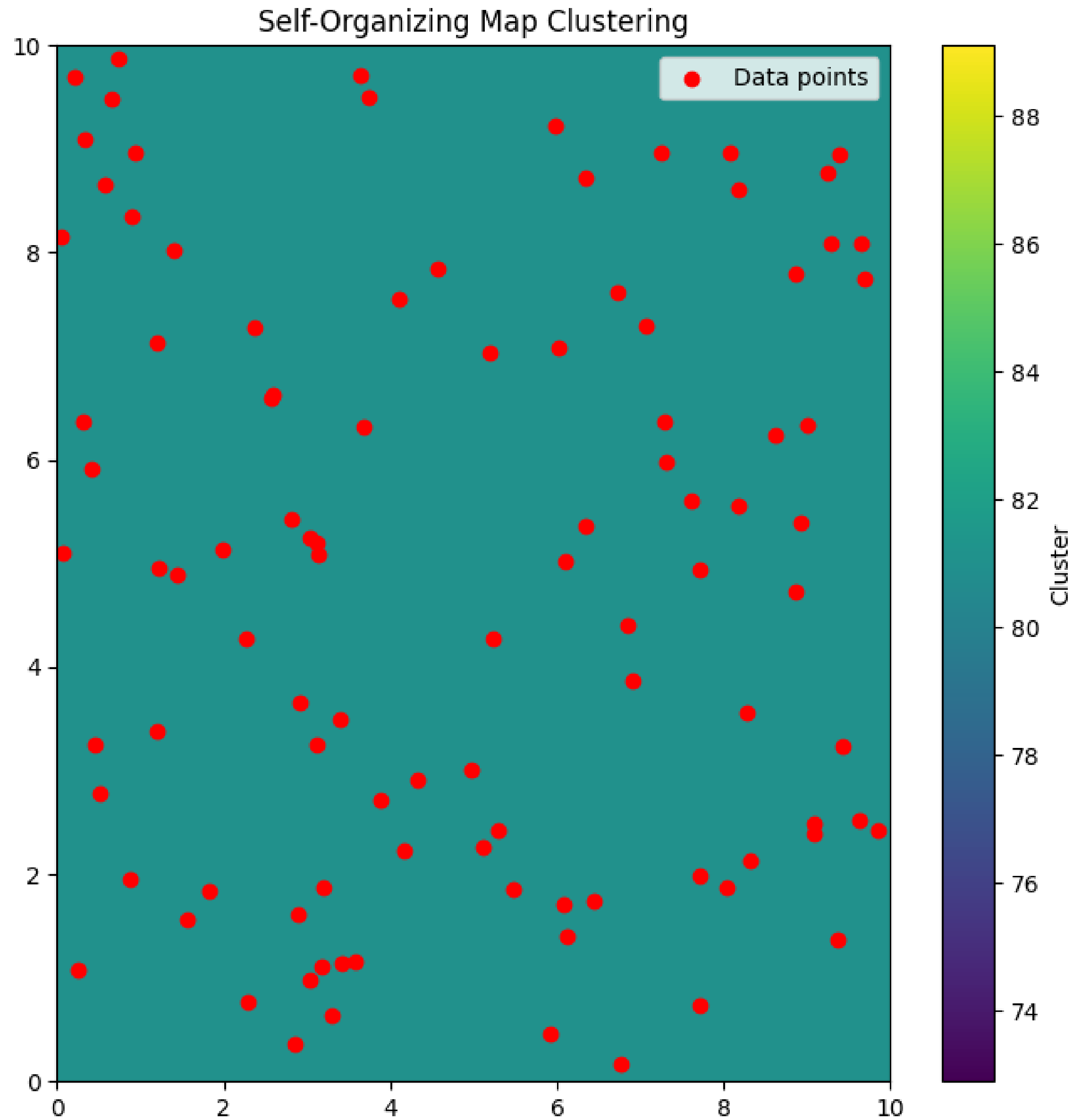
7. End of the SOM implementation.

PROGRAM:

```
import numpy as np
import matplotlib.pyplot as plt

# Generate some sample data (replace this with your own dataset)
np.random.seed(42)
data = np.random.rand(100, 2)
```

```
# SOM parameters
grid_size = (10, 10) # Grid size of the SOM
input_dim = 2 # Dimensionality of the input data
learning_rate = 0.2
num_epochs = 1000

# Initialize the SOM
weight_matrix = np.random.rand(grid_size[0], grid_size[1], input_dim)

# Training loop
for epoch in range(num_epochs):
    for input_vector in data:
        # Find the Best Matching Unit (BMU)
        distances = np.linalg.norm(weight_matrix - input_vector, axis=-1)
        bmu_coords = np.unravel_index(np.argmin(distances), distances.shape)

        # Update the BMU and its neighbors
        for i in range(grid_size[0]):
            for j in range(grid_size[1]):
                distance_to_bmu = np.linalg.norm(np.array([i, j]) - np.array(bmu_coords))
                influence = np.exp(-distance_to_bmu**2 / (2 * (epoch + 1)**2))  # Adjusting the
influence based on the current epoch
                weight_matrix[i, j] += influence * learning_rate * (input_vector - weight_matrix[i, j])

# Create a map of cluster assignments
cluster_map = np.zeros((grid_size[0], grid_size[1]), dtype=int)
for i in range(grid_size[0]):
    for j in range(grid_size[1]):
        distances = np.linalg.norm(data - weight_matrix[i, j], axis=-1)
        cluster_map[i, j] = np.argmin(distances)

# Visualize the results
plt.figure(figsize=(8, 8))
plt.pcolormesh(cluster_map, cmap='viridis')
plt.colorbar(label='Cluster')
plt.scatter(data[:, 0] * grid_size[0], data[:, 1] * grid_size[1], color='red', label='Data points')
plt.legend()
plt.title('Self-Organizing Map Clustering')
plt.show()
```

Output:



Result: Thus the above program for self-organizing map executed successfully with desired output.

EXP 5	Programming exercises on maximizing a function using Genetic algorithm.	DATE:

AIM: Understand the concept of maximizing function using Genetic algorithm using python programming.

Algorithm:

1. Initialize the population with random solutions.
2. Define the fitness function to evaluate how good each solution is.
3. Set the maximum number of generations.
4. Set the mutation rate (probability of changing a gene in an individual).
5. Set the crossover rate (probability of two individuals mating).
6. Repeat for each generation:
   a. Evaluate the fitness of each individual in the population using the fitness function.
   b. Select the best individuals based on their fitness to become parents.
   c. Create a new generation by crossover (mixing) the genes of the parents.
   d. Apply mutation to some individuals in the new generation.
   e. Replace the old population with the new generation.
7. Repeat for the specified number of generations.
8. Find and return the individual with the highest fitness as the best solution.

PROGRAM:

```
import random

# Define the fitness function (our objective function to maximize)
def fitness_function(x):
    return -x**2 + 6*x + 9

# Initialize the population
def initialize_population(pop_size, lower_bound, upper_bound):
    return [random.uniform(lower_bound, upper_bound) for _ in range(pop_size)]

# Select parents based on their fitness
def select_parents(population):
    total_fitness = sum(fitness_function(individual) for individual in population)
    roulette_wheel = [fitness_function(individual) / total_fitness for individual in population]
    parent1 = random.choices(population, weights=roulette_wheel)[0]
    parent2 = random.choices(population, weights=roulette_wheel)[0]
    return parent1, parent2

# Perform crossover to create a new generation
def crossover(parent1, parent2, crossover_prob=0.7):
    if random.random() < crossover_prob:
        crossover_point = random.randint(1, 1)  # Corrected this line
        child1 = (parent1 + parent2) / 2
        child2 = (parent1 + parent2) / 2
        return child1, child2
    else:
```

```
        return parent1, parent2

# Perform mutation in the population
def mutate(individual, mutation_prob=0.01):
    if random.random() < mutation_prob:
        individual += random.uniform(-1, 1)
    return individual

# Genetic Algorithm
def genetic_algorithm(generations, pop_size, lower_bound, upper_bound):
    population = initialize_population(pop_size, lower_bound, upper_bound)

    for gen in range(generations):
        new_population = []

        while len(new_population)  < pop_size:
            parent1, parent2 = select_parents(population)
            child1, child2 = crossover(parent1, parent2)
            child1 = mutate(child1)
            child2 = mutate(child2)
            new_population.extend([child1, child2])

        population = new_population

        best_individual = max(population, key=fitness_function)
        print(f"Generation {gen+1}: Best individual - {best_individual}, Fitness -
{fitness_function(best_individual)}")

    return max(population, key=fitness_function)

if __name__== "_main_":
    generations = 50
    pop_size = 100
    lower_bound = -10
    upper_bound = 10

    best_solution = genetic_algorithm(generations, pop_size, lower_bound, upper_bound)
    print(f"Best solution found: {best_solution}, Fitness: {fitness_function(best_solution)}")
```

Output:

```
Generation 1: Best individual - 1.338221851975824, Fitness - 15.23849338674934
Generation 2: Best individual - -4.617497504442627, Fitness - -40.02626823018966
Generation 3: Best individual - -6.0365409961964005, Fitness - -63.65907317593823
Generation 4: Best individual - -6.086873542143298, Fitness - -64.57127077090388
Generation 5: Best individual - -7.73134856380424, Fitness - -97.16184199786333
Generation 6: Best individual - -8.010012301451464, Fitness - -103.22037087811256
Generation 7: Best individual - -8.128709772289175, Fitness - -105.84818119584459
```

Generation 8: Best individual - -8.128709772289175, Fitness - -105.84818119584459
Generation 9: Best individual - -8.106182932958884, Fitness - -105.3472993403472
Generation 10: Best individual - -8.4253359573576, Fitness - -112.53830173848851
Generation 11: Best individual - -8.339831707745882, Fitness - -110.59178315999888
Generation 12: Best individual - -8.511740621618573, Fitness - -114.52017213942318
Generation 13: Best individual - -8.562519336850656, Fitness - -115.69185341504533
Generation 14: Best individual - -8.475270784635734, Fitness - -113.68183958071442
Generation 15: Best individual - -7.799825444300792, Fitness - -98.63622962736679
Generation 16: Best individual - -7.799825444300792, Fitness - -98.63622962736679
Generation 17: Best individual - -8.23641044465958, Fitness - -108.25691968085488
Generation 18: Best individual - -8.469625574063269, Fitness - -113.55231080920618
Generation 19: Best individual - -8.005479094012971, Fitness - -103.12057008875657
Generation 20: Best individual - -8.331329932705664, Fitness - -110.39903804383135
Generation 21: Best individual - -8.483646271761524, Fitness - -113.87413169494235
Generation 22: Best individual - -8.512424180517046, Fitness - -114.53591051215358
Generation 23: Best individual - -8.512424180517046, Fitness - -114.53591051215358
Generation 24: Best individual - -8.536034194890835, Fitness - -115.08008494569063
Generation 25: Best individual - -8.58733120697589, Fitness - -116.26624450015733
Generation 26: Best individual - -7.932088839364656, Fitness - -101.51056639176127
Generation 27: Best individual - -7.932088839364656, Fitness - -101.51056639176127
Generation 28: Best individual - -8.633557704228673, Fitness - -117.33966485761832
Generation 29: Best individual - -8.500839675355433, Fitness - -114.26931323822967
Generation 30: Best individual - -8.663730407782117, Fitness - -118.04260702542118

Result: Thus the above program maximizing function using genetic algorithm executed successfully with desired output.

EXP 6	Implementation of two input sine function.	DATE:

AIM: Understand the concept of implementation of two input sine function using Genetic algorithm.

Algorithm:

Genetic Algorithm for Two-Input Sine Function Optimization

1. Define the fitness function
2. Initialize the population
3. Define functions for genetic operations
4. Implement the main genetic algorithm loop
5. Print the final best solution found by the genetic algorithm.

PROGRAM

```
import random
import math

# Define the fitness function (sine function with two inputs)
def fitness_function(x, y):
    return math.sin(x) + math.sin(y)

# Initialize the population
def initialize_population(pop_size, lower_bound, upper_bound):
    return [(random.uniform(lower_bound, upper_bound), random.uniform(lower_bound,
upper_bound)) for _ in range(pop_size)]

# Select parents based on their fitness
def select_parents(population):
    total_fitness = sum(fitness_function(x, y) for x, y in population)
    roulette_wheel = [fitness_function(x, y) / total_fitness for x, y in population]
    parent1 = random.choices(population, weights=roulette_wheel)[0]
    parent2 = random.choices(population, weights=roulette_wheel)[0]
    return parent1, parent2

# Perform crossover to create a new generation
def crossover(parent1, parent2, crossover_prob=0.7):
    if random.random() < crossover_prob:
        crossover_point = random.randint(0, 1)
        child1 = (parent1[0], parent2[1])
        child2 = (parent2[0], parent1[1])
        return child1, child2
    else:
        return parent1, parent2
```

```
# Perform mutation in the population
def mutate(individual, mutation_prob=0.01):
    x, y = individual
    if random.random() < mutation_prob:
        x += random.uniform(-0.1, 0.1)
    if random.random() < mutation_prob:
        y += random.uniform(-0.1, 0.1)
    return x, y

# Genetic Algorithm
def genetic_algorithm(generations, pop_size, lower_bound, upper_bound):
    population = initialize_population(pop_size, lower_bound, upper_bound)

    for gen in range(generations):
        new_population = []

        while len(new_population)  < pop_size:
            parent1, parent2 = select_parents(population)
            child1, child2 = crossover(parent1, parent2)
            child1 = mutate(child1)
            child2 = mutate(child2)
            new_population.extend([child1, child2])

        population = new_population

        best_individual = max(population, key=lambda ind: fitness_function(*ind))
        print(f"Generation {gen+1}: Best individual - {best_individual}, Fitness -
{fitness_function(*best_individual)}")

    return max(population, key=lambda ind: fitness_function(*ind))

if __name__ == "_main_":
    generations = 50
    pop_size = 100
    lower_bound = -2 * math.pi
    upper_bound = 2 * math.pi

    best_solution = genetic_algorithm(generations, pop_size, lower_bound, upper_bound)
    print(f"Best solution found: {best_solution}, Fitness: {fitness_function(*best_solution)}")
```

Output:

Generation 1: Best individual - (-5.806639394411164, 2.957052015269947), Fitness - 0.6422076600091893
Generation 2: Best individual - (-3.7004701839702663, 4.4413546380285975), Fitness - - 0.43325964387284566

Generation 3: Best individual - (-3.7004701839702663, 5.464316418988149), Fitness - - 0.20013884834113005
Generation 4: Best individual - (5.481791654037208, 3.3095163097626763), Fitness - - 0.8854619344317294
Generation 5: Best individual - (4.897491323013819, 3.3095163097626763), Fitness - - 1.150052992911647
Generation 6: Best individual - (4.976671184995054, 3.3095163097626763), Fitness - - 1.132415822508853б
Generation 7: Best individual - (3.9420165382340246, 3.3095163097626763), Fitness - - 0.8847869227205696
Generation 8: Best individual - (4.198534144176835, 5.481189847293816), Fitness - - 1.5896010966615468
Generation 9: Best individual - (4.198534144176835, 5.481189847293816), Fitness - - 1.5896010966615468
Generation 10: Best individual - (4.198534144176835, 5.481189847293816), Fitness - - 1.5896010966615468
Generation 11: Best individual · (4.34542752972704, 5.481189847293816), Fitness - - 1.6521667383260996
Generation 12: Best individual - (-1.2170450032547304, 5.481189847293816), Fitness - - 1.6568246976897136
Generation 13: Best individual - (-1.2185577577082327, 5.481189847293816), Fitness - - 1.6573476714317006
Generation 14: Best individual - (-1.2170450032547304, 5.481189847293816), Fitness - - 1.6568246976897136
Generation 15: Best individual - (-1.2170450032547304, 5.481189847293816), Fitness - - 1.6568246976897136
Generation 16: Best individual - (-1.2170450032547304, 5.481189847293816), Fitness - - 1.6568246976897136
Generation 17: Best individual - (-1.2170450032547304, 5.481189847293816), Fitness - - 1.6568246976897136
Generation 18: Best individual - (-1.2170450032547304, 5.481189847293816), Fitness - - 1.6568246976897136
Generation 19: Best individual - (-1.2185577577082327, 5.481189847293816), Fitness - - 1.6573476714317006
Generation 20: Best individual - (4.266603727856264, 5.481189847293816), Fitness - - 1.621017281069609
Generation 21: Best individual - (-1.2170450032547304, 5.481189847293816), Fitness - - 1.6568246976897136
Generation 22: Best individual - (-1.2170450032547304, 5.481189847293816), Fitness - - 1.6568246976897136
Generation 23: Best individual - (4.976671184995054, 5.481189847293816), Fitness - - 1.6840251615701645
Generation 24: Best individual - (4.897491323013819, 5.481189847293816), Fitness - - 1.7016623319729578
Generation 25: Best individual - (-1.2185577577082327, 5.481189847293816), Fitness - - 1.6573476714317006
Generation 26: Best individual - (-1.2185577577082327, 5.481189847293816), Fitness - - 1.6573476714317006
Generation 27: Best individual - (-1.2185577577082327, 5.481189847293816), Fitness - - 1.6573476714317006

Generation 28: Best individual - (-1.2170450032547304, 4.380981364013678), Fitness - -1.8836650637984946
Generation 29: Best individual - (-1.2170450032547304, 4.380981364013678), Fitness - -1.8836650637984946
Generation 30: Best individual - (-1.2170450032547304, 4.380981364013678), Fitness - -1.8836650637984946

Result: Thus the above program implementation of two input sine function using genetic algorithm executed successfully.

EXP 7	Implementation of three input nonlinear function	DATE:

AIM

Algorithm

Genetic Algorithm for Three-Input Nonlinear Function Optimization
1. Define the fitness function.
2. Initialize the population.
3. Define functions for genetic operations.
4. Implement the main genetic algorithm loop.
5. Print the final best solution found by the genetic algorithm.

PROGRAM

```
import random

# Define the fitness function (three-input nonlinear function)
def fitness_function(x, y, z):
    return -(x**2 + y**2 + z**2) + 10 * (math.cos(2*math.pi*x) + math.cos(2*math.pi*y) +
math.cos(2*math.pi*z))

# Initialize the population
def initialize_population(pop_size, lower_bound, upper_bound):
    return [(random.uniform(lower_bound, upper_bound), random.uniform(lower_bound,
upper_bound), random.uniform(lower_bound, upper_bound)) for _ in range(pop_size)]

# Select parents based on their fitness
def select_parents(population):
    total_fitness = sum(fitness_function(x, y, z) for x, y, z in population)
    roulette_wheel = [fitness_function(x, y, z) / total_fitness for x, y, z in population]
    parent1 = random.choices(population, weights=roulette_wheel)[0]
    parent2 = random.choices(population, weights=roulette_wheel)[0]
    return parent1, parent2

# Perform crossover to create a new generation
def crossover(parent1, parent2, crossover_prob=0.7):
    if random.random() < crossover_prob:
        crossover_point1 = random.uniform(0, 1)
        crossover_point2 = random.uniform(0, 1)
        child1 = (crossover_point1 * parent1[0] + (1 - crossover_point1) * parent2[0],
                  crossover_point1 * parent1[1] + (1 - crossover_point1) * parent2[1],
                  crossover_point1 * parent1[2] + (1 - crossover_point1) * parent2[2])

        child2 = (crossover_point2 * parent1[0] + (1 - crossover_point2) * parent2[0],
                  crossover_point2 * parent1[1] + (1 - crossover_point2) * parent2[1],
```

```
            crossover_point2 * parent1[2] + (1 - crossover_point2) * parent2[2])
        return child1, child2
    else:
        return parent1, parent2

# Perform mutation in the population
def mutate(individual, mutation_prob=0.01):
    x, y, z = individual
    if random.random() < mutation_prob:
        x += random.uniform(-0.1, 0.1)
    if random.random() < mutation_prob:
        y += random.uniform(-0.1, 0.1)
    if random.random() < mutation_prob:
        z += random.uniform(-0.1, 0.1)
    return x, y, z

# Genetic Algorithm
def genetic_algorithm(generations, pop_size, lower_bound, upper_bound):
    population = initialize_population(pop_size, lower_bound, upper_bound)

    for gen in range(generations):
        new_population = []

        while len(new_population)  < pop_size:
            parent1, parent2 = select_parents(population)
            child1, child2 = crossover(parent1, parent2)
            child1 = mutate(child1)
            child2 = mutate(child2)
            new_population.extend([child1, child2])

        population = new_population

        best_individual = max(population, key=lambda ind: fitness_function(*ind))
        print(f"Generation {gen+1}: Best individual - {best_individual}, Fitness -
{fitness_function(*best_individual)}")

    return max(population, key=lambda ind: fitness_function(*ind))

if__name___== "_main_":
    import math

    generations = 50
    pop_size = 100
    lower_bound = -1
    upper_bound = 1

    best_solution = genetic_algorithm(generations, pop_size, lower_bound, upper_bound)
    print(f"Best solution found: {best_solution}, Fitness: {fitness_function(*best_solution)}")
```

Output:
Generation 1: Best individual - (-0.05856140717606745, 0.031920444393859077, 0.1749430018353162), Fitness - 23.638248996079486
Generation 2: Best individual - (-0.0435664961811546, -0.21954873032302427, -0.16051643562429213), Fitness - 16.78431041370941
Generation 3: Best individual - (-0.08047256311183462, 0.08748607229595336, -0.033675015554337134), Fitness - 27.03730906535697
Generation 4: Best individual - (0.09173450837429278, 0.2701480951847052, -0.04923516012359691), Fitness - 16.563306003309293
Generation 5: Best individual - (0.06931525412773312, 0.05650887471327237, -0.6038978838220976), Fitness - 10.126283111803192
Generation 6: Best individual - (0.09551296321256389, 0.3037721680736508, 0.09828297902264888), Fitness - 12.980004103640377
Generation 7: Best individual - (0.36966788594966404, 0.020338697069605532, 0.0003553226927579256), Fitness - 12.951119752237492
Generation 8: Best individual - (-0.13483009446855374, 0.031089470762199117, -0.5756450454760131), Fitness - 7.188833165750407
Generation 9: Best individual - (-0.063607907585292, -0.04456979821453749, -0.45149742141484683), Fitness - 9.073275131295658
Generation 10: Best individual - (0.011176844816005782, 0.38683718057120575, -0.4586668963552707), Fitness - -7.626399941503013
Generation 11: Best individual - (0.42384530453390673, -0.017961106838255136, -0.4918457018441281), Fitness - -9.349262068882442
Generation 12: Best individual - (0.33991169237820235, 0.31387850909754794, -0.4929268418150241), Fitness - -19.707456018578803
Generation 13: Best individual - (0.4287928945546883, 0.5471800246826355, -0.2740060489612387), Fitness - -20.6405201860691
Generation 14: Best individual - (0.4287928945546883, 0.5471800246826355, -0.2740060489612387), Fitness - -20.6405201860691
Generation 15: Best individual - (0.4287928945546883, 0.5471800246826355, -0.2740060489612387), Fitness - -20.6405201860691
Generation 16: Best individual - (0.4112385766210301, 0.5040715286796309, -0.3178766342306278), Fitness - -23.14240113919352
Generation 17: Best individual - (0.4147812368137274, 0.5041212646314481, -0.33531860658801094), Fitness - -24.24331785854991
Generation 18: Best individual - (0.4147812368137274, 0.5041212646314481, -0.33531860658801094), Fitness - -24.24331785854991
Generation 19: Best individual - (0.4147812368137274, 0.5041212646314481, -0.33531860658801094), Fitness - -24.24331785854991
Generation 20: Best individual - (0.30622472006746704, 0.4950670130302236, -0.44378439110485673), Fitness - -23.373347854338835
Generation 21: Best individual - (0.30622472006746704, 0.4950670130302236, -0.44378439110485673), Fitness - -23.373347854338835
Generation 22: Best individual - (0.3063202575245222, 0.4950822379662878, -0.4437892287140689), Fitness - -23.379191958933983
Generation 23: Best individual - (0.33335312358816604, 0.49763301297219154, -0.43647155864564097), Fitness - -24.763115313088132
Generation 24: Best individual - (0.40994188262594977, 0.4096825578747779, -0.42523970750461587), Fitness - -26.30751096118025

Generation 25: Best individual - (0.39756456561304454, 0.48504626799164063, -0.4088104102096408), Fitness - -26.9186217943733
Generation 26: Best individual - (0.38380825053477785, 0.5006633169359437, -0.4288824223540211), Fitness - -27.051357458861553
Generation 27: Best individual - (0.4057387303749639, 0.5681504161728289, -0.4242177696903832), Fitness - -26.948969587424035
Generation 28: Best individual - (0.40511387532462084, 0.47856788928174143, -0.36781261632617945), Fitness - -25.457363863631336
Generation 29: Best individual - (0.40671661145515176, 0.4783203340919677, -0.3735290334571525), Fitness - -25.777462228224486
Generation 30: Best individual - (0.40696169049260167, 0.4789885169063051, -0.3799677735908236), Fitness - -26.080160960767703

Result: Thus the above program genetic algorithm for three input non-linear function optimization executed successfully.