ANNA UNIVERSITY REGIONAL CAMPUS COIMBATORE

LABORATORY RECORD

2024-2025

NAME	
REG.NO	
BRANCH	**B.E COMPUTER SCIENCE AND ENGINEERING**
SUBJECT CODE	**CCS364**
SUBJECT TITLE	**SOFT COMPUTING (COMPONENT LAB)**

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

ANNA UNIVERSITY REGIONAL CAMPUS

COIMBATORE - 641046

ANNA UNIVERSITY

REGIONAL CAMPUS, COIMBATORE - 641046

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

BONAFIDE CERTIFICATE

Certified that this is the bonafide record of Practical done in **CCS364 – SOFT COMPUTING (COMPONENT LABORATORY)** by …………………………....

Reg. No ……………………… in Fourth Year/Seventh Semester during 2024-2025.

STAFF IN-CHARGE **HEAD OF THE DEPARTMENT**

University Register Number: ………………………………………………...…………

Submitted for the University Practical Examination Held on …………………..……........

INTERNAL EXAMINER **EXTERNAL EXAMINER**

INDEX

Ex No: 01	IMPLEMENTATION OF FUZZY CONTROL/ INFERENCE SYSTEM
Date:	

AIM:

To understand the concept of fuzzy control/ inference system using python programming language.

PROCEDURE:

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..

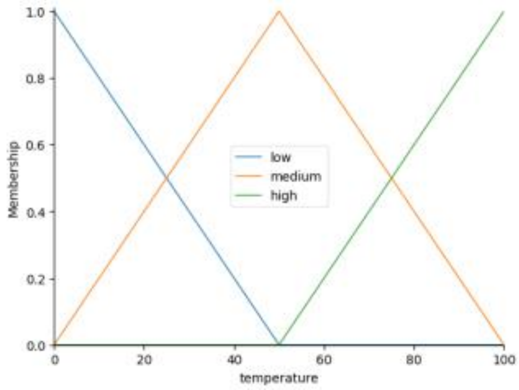
PROGRAM:

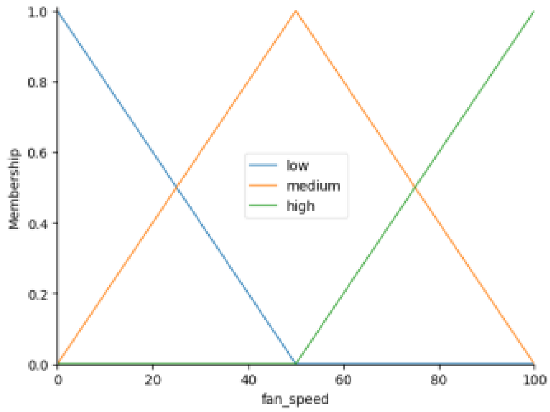
```
pip install scikit-fuzzy

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
```

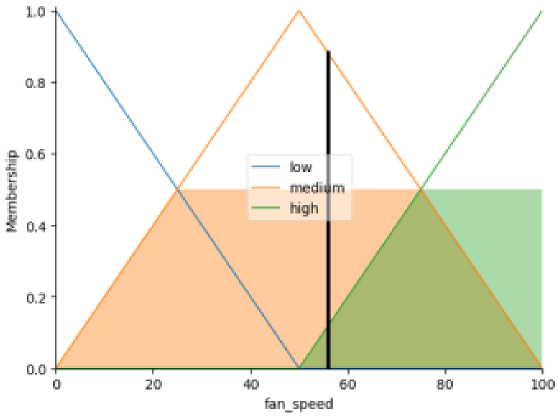
```
temperature['low'] = fuzz.trimf(temperature.universe, [0, 0, 50])
temperature['medium'] = fuzz.trimf(temperature.universe, [0, 50, 100])
temperature['high'] = fuzz.trimf(temperature.universe, [50, 100, 100])
# Define membership functions for fan speed
fan_speed['low'] = fuzz.trimf(fan_speed.universe, [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 and print the result
fan_speed_ctrl.compute()
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 is executed successfully with the desired output.

Ex No: 02	IMPLEMENTING THE CLASSIFICATION WITH A DISCRETE PERCEPTRON
Date:	

AIM:

To understand the concept of classification with discrete perceptron using python programming language.

PROCEDURE:

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:
```

```python
            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 for classification with discrete perceptron is executed successfully with the desired output.

Ex No: 03	IMPLEMENTATION OF XOR WITH BACKPROPAGATION ALGORITHM
Date:	

AIM:

To understand the concept of XOR with backpropagation algorithm using python programing language.

PROCEDURE:

Step 1:Initialize Neural Network:

Step 2:Randomly initialize weights and biases.

Step 3:Define Training Data as XOR input and target data.

Step 4:Set Hyperparameters for Learning rate, number of epochs, number of hidden layers and neurons, activation function.

Step 5:Training Loop for each epoch

- Forward Propagation: Compute activations for hidden and output layers.
- Calculate Error: Compute error between predicted and actual outputs.
- Backpropagation: Compute gradients and update weights and biases.

Step 6:Use the trained network to predict XOR values for new inputs.

PROGRAM:

```
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, hidden_size, output_size = 2, 2, 1
learning_rate, epochs = 0.1, 10000
# Initialize weights randomly
```

```python
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_output = sigmoid(np.dot(input_data, hidden_weights))
    predicted_output = sigmoid(np.dot(hidden_output, output_weights))
    # Calculate error and backpropagation
    error = target_data - predicted_output
    output_delta = error * sigmoid_derivative(predicted_output)
    hidden_delta = output_delta.dot(output_weights.T)  *
sigmoid_derivative(hidden_output)
    # Update weights
    output_weights += hidden_output.T.dot(output_delta) * learning_rate
    hidden_weights += input_data.T.dot(hidden_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_output = sigmoid(np.dot(data, hidden_weights))
    predicted_output = sigmoid(np.dot(hidden_output, output_weights))
    print(f"Input: {data} Predicted Output: {predicted_output[0]}")
```

OUTPUT:

Input: [0 0] Predicted Output: 0.24923330111068986

Input: [0 1] Predicted Output: 0.698513884181489

Input: [1 0] Predicted Output: 0.6984375039219314

Input: [1 1] Predicted Output: 0.39007274766633737

RESULT:

Thus the above program for the classification with the discrete perception is executed successfully with the desired output.

Ex No: 04	IMPLEMENTATION OF SELF ORGANIZING MAPS
Date:	

AIM:

To understand the concept of self-organizing maps for a specific application using python programming language.

PROCEDURE:

Step 1:Initialize SOM

- Define grid size and shape.
- Initialize neuron weights randomly.
- Set learning rate and neighborhood radius.

Step 2:Define Training Dataset

- Use high-dimensional input vectors.

Step 3:Train SOM

For each epoch:

- Select a random data point.
- Find the Best Matching Unit (BMU).
- Update BMU and neighbor weights.
- Decrease learning rate and neighborhood radius.

Step 4:Repeat Training

- Continue until convergence or for a set number of epochs.

Step 5:Map New Data

- Find BMU for new input vectors.
- Use BMU location for decisions or predictions.

Step 6:Visualization (Optional)

- Visualize SOM grid to understand data clustering.

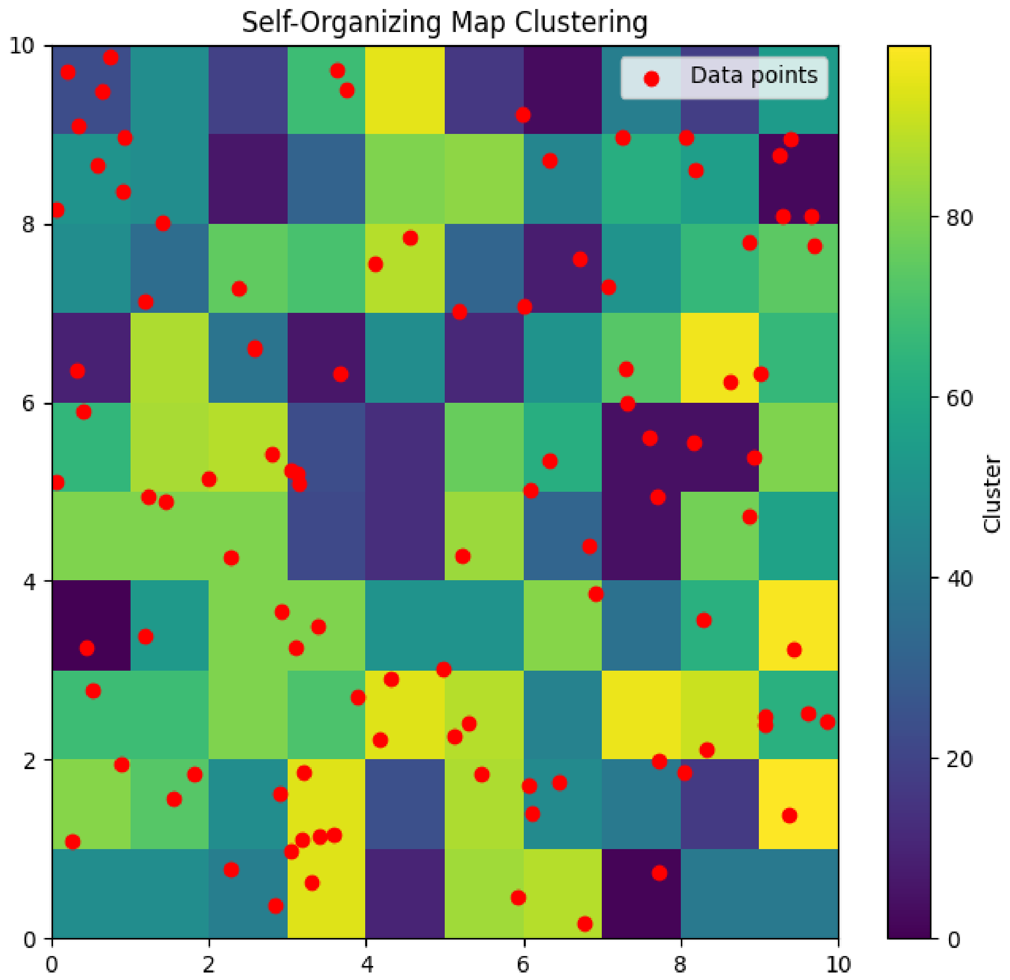
Step 7:End.

PROGRAM:

```
import numpy as np
import matplotlib.pyplot as plt
from minisom import MiniSom
# Generate sample data (replace this with your own dataset)
np.random.seed(42)
data = np.random.rand(100, 2)
# SOM parameters
grid_size = (10, 10)
input_dim = 2
learning_rate = 0.5
num_epochs = 1000
# Initialize and train the SOM
som = MiniSom(grid_size[0], grid_size[1], input_dim, sigma=1.0,
learning_rate=learning_rate)
som.random_weights_init(data)
som.train_random(data, num_epochs)
# 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 - som.get_weights()[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 the self-organizing map is executed successfully with the desired output.

Ex No: 05	IMPLEMENTING A MAXIMIZING FUNCTION USING GENETIC ALGORITHM
Date:	

AIM:

To understand the concept of maximizing function using the Genetic algorithm using python programming.

PROCEDURE:

Step 1:Initialize the population with random solutions.

Step 2:Define the fitness function to evaluate how good each solution is.

Step 3:Set the maximum number of generations.

Step 4:Set the mutation rate (probability of changing a gene in an individual).

Step 5:Set the crossover rate (probability of two individuals mating).

Step 6:Repeat for each generation:

- Evaluate the fitness of each individual in the population using the fitness function.
- Select the best individuals based on their fitness to become parents.
- Create a new generation by crossover (mixing) the genes of the parents.
- Apply mutation to some individuals in the new generation.
- Replace the old population with the new generation.

Step 7:Repeat for the specified number of generations.

Step 8:Find and return the individual with the highest fitness as the best solution.

PROGRAM:

```
import random

# Define the fitness function

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]
    return random.choices(population, weights=roulette_wheel, k=2)
# Perform crossover to create a new generation
def crossover(parent1, parent2, crossover_prob=0.7):
    if random.random() < crossover_prob:
        return (parent1 + parent2) / 2, (parent1 + parent2) / 2
    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)
            new_population.extend([mutate(child1), mutate(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 best_individual

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 - 1.1664779815441046, Fitness - 14.63819700783742

Generation 2: Best individual - 0.11971547614047484, Fitness - 9.703961061615308

Generation 3: Best individual - -7.137435971911696, Fitness - -84.76760808460924

Generation 4: Best individual - -7.137435971911696, Fitness - -84.76760808460924

………

Generation 49: Best individual - -8.135847575292075, Fitness - -106.0071012201384

Generation 50: Best individual - -8.530432465548909, Fitness - -114.95087284258429

Best solution found: -8.530432465548909, Fitness: -114.95087284258429

RESULT:

Thus the above program for maximizing function using the genetic algorithm is executed successfully with the desired output.

Ex No: 06	IMPLEMENTATION OF TWO INPUT SINE FUNCTION
Date:	

AIM:

To understand the concept of implementation of two input sine function using the Genetic algorithm.

PROCEDURE:

Step 1:Define the fitness function

Step 2:Initialize the population

Step 3:Define functions for genetic operations

Step 4:Implement the main genetic algorithm loop

Step 5: Print the final best solution found by the genetic algorithm.

PROGRAM:

```
import random
import math
# Define the fitness function
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]
    return random.choices(population, weights=roulette_wheel, k=2)
# Perform crossover to create a new generation
def crossover(parent1, parent2, crossover_prob=0.7):
```

```
    if random.random() < crossover_prob:
        return (parent1[0], parent2[1]), (parent2[0], parent1[1])
    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)
            new_population.extend([mutate(child1), mutate(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 best_individual
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 - (-4.150368698953198, -4.741662144060966), Fitness - 1.845751808619636

Generation 2: Best individual - (-4.150368698953198, 1.9167108227131777), Fitness - 1.786946016736844

Generation 3: Best individual - (-4.150368698953198, 1.9167108227131777), Fitness - 1.786946016736844

Generation 4: Best individual - (-4.150368698953198, 1.3789433634441872), Fitness - 1.827832837944686

…………

Generation 50: Best individual - (-4.297944867170654, 1.9167108227131777), Fitness - 1.856106086704245

Best solution found: (-4.297944867170654, 1.9167108227131777), Fitness: 1.856106086704245

RESULT:

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

Ex No: 07	IMPLEMENTATION OF THREE INPUT NONLINEAR FUNCTION
Date:	

AIM:

To understand the concept of implementation of three input nonlinear function using the Genetic algorithm.

PROCEDURE:

Step 1:Define the fitness function.

Step 2:Initialize the population.

Step 3:Define functions for genetic operations.

Step 4:Implement the main genetic algorithm loop.

Step 5:Print the final best solution found by the genetic algorithm.

PROGRAM:

```
import random
import math
# Define the fitness function
def fitness_function(x, y, z):
    return math.sin(x) + math.cos(y) + math.tan(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]
    return random.choices(population, weights=roulette_wheel, k=2)
# Perform crossover to create a new generation
def crossover(parent1, parent2, crossover_prob=0.7):
    if random.random() < crossover_prob:
        return (parent1[0], parent2[1], parent2[2]), (parent2[0], parent1[1], parent1[2])
    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)
                new_population.extend([mutate(child1), mutate(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 best_individual
    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 - (1.8617081428285154, -0.4948635733095559, 4.692862349285036), Fitness - 53.04361824176664

Generation 2: Best individual - (1.8617081428285154, -0.4948635733095559, 4.719947363777022), Fitness - -130.46288792066935

Generation 3: Best individual - (1.7992768790063087, -0.4948635733095559, 4.712995685573096), Fitness - -1646.3927415827002

Generation 4: Best individual - (1.8617081428285154, -0.4948635733095559, 4.6643034471231495), Fitness - 22.6182608243739

……….

Generation 50: Best individual - (1.8617081428285154, -0.5856648230135555, 4.787923047528845), Fitness - -11.422544245738507

Best solution found: (1.8617081428285154, -0.5856648230135555, 4.787923047528845), Fitness: -11.422544245738507

RESULT:

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