**VISVESVARAYA TECHNOLOGICAL UNIVERSITY**



**“JnanaSangama”, Belgaum -590014, Karnataka.**

**LAB RECORD**

Bio Inspired Systems (23CS5BSBIS)

***Submitted by***

**PRATIK JANA(1BM22CS356)**

***in partial fulfillment for the award of the degree of***

**BACHELOR OF ENGINEERING**

***in***

**COMPUTER SCIENCE AND ENGINEERING**

**B.M.S. COLLEGE OF ENGINEERING**

**(Autonomous Institution under VTU)**

**BENGALURU-560019**

**Sep-2024 to Jan-2025**

**B.M.S. College of Engineering,**

**Bull Temple Road, Bangalore 560019**

(Affiliated To Visvesvaraya Technological University, Belgaum)

**Department of Computer Science and Engineering**

****

**CERTIFICATE**

This is to certify that the Lab work entitled “Bio Inspired Systems (23CS5BSBIS)” carried out by **PRATIK JANA (1BM22CS356),** who is bonafide student of **B.M.S. College of Engineering.** It is in partial fulfillment for the award of **Bachelor of Engineering in Computer Science and Engineering** of the Visvesvaraya Technological University, Belgaum. The Lab report has been approved as it satisfies the academic requirements of the above mentioned subject and the work prescribed for the said degree.

Dr. Kavitha Sooda Professor & HOD

Department of CSE, BMSCE

Prof. Saritha A N Assistant Professor

Department of CSE, BMSCE

**Index**

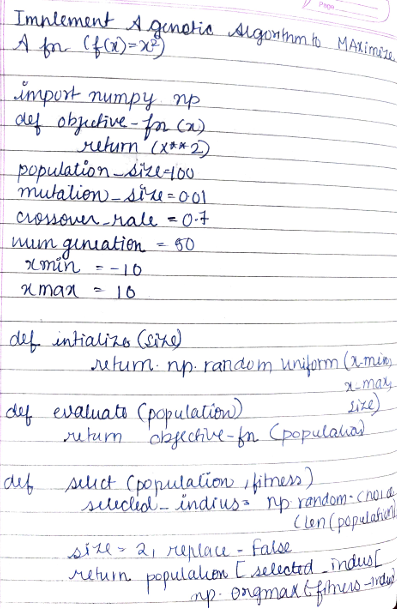
|  |  |  |  |
| --- | --- | --- | --- |
| **Sl.**  **No.** | **Date** | **Experiment Title** | **Page No.** |
| 1 | 24/10/24 | Genetic Algorithm for Optimization Problems | 1-4 |
| 2 | 7/11/24 | Particle Swarm Optimization for Function Optimization | 5-8 |
| 3 | 14/11/24 | Ant Colony Optimization for the Traveling Salesman Problem | 9-14 |
| 4 | 21/11/24 | Cuckoo Search (CS) | 15-18 |
| 5 | 28/11/24 | Grey Wolf Optimizer (GWO) | 19-23 |
| 6 | 19/12/24 | Parallel Cellular Algorithms and Programs | 24-27 |
| 7 | 19/12/24 | Optimization via Gene Expression Algorithms | 28-32 |

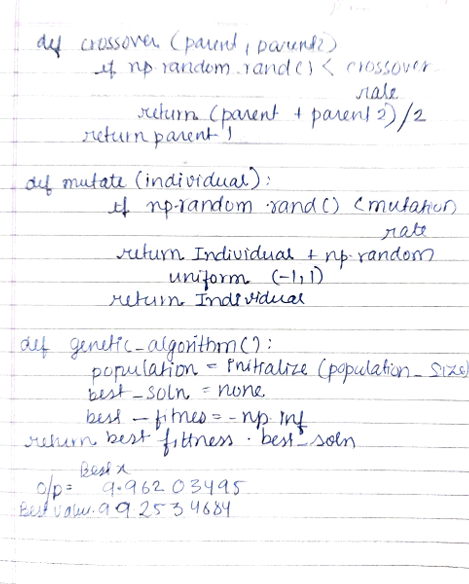
Github Link: https://github.com/AnkitSB19/BISLAB\_1BM22CS353

Algorithm:

# Program 1

## Genetic Algorithm for Optimization Problems





Code:

import numpy as np

# Define the objective function def objective\_function(x):

return x\*\*2

# Initialize parameters population\_size = 100

mutation\_rate = 0.01

crossover\_rate = 0.7

num\_generations = 50

x\_min = -10

x\_max = 10

# Create initial population

def initialize\_population(size):

return np.random.uniform(x\_min, x\_max, size)

# Evaluate fitness

def evaluate\_fitness(population):

return objective\_function(population)

# Selection (Tournament Selection) def select(population, fitness):

selected\_indices = np.random.choice(len(poTpulation), size=2, replace=False) return population[selected\_indices[np.argmax(fitness[selected\_indices])]]

# Crossover

def crossover(parent1, parent2):

if np.random.rand() < crossover\_rate:

return (parent1 + parent2) / 2 # Simple averaging return parent1

# Mutation

def mutate(individual):

if np.random.rand() < mutation\_rate:

return individual + np.random.uniform(-1, 1) # Random mutation return individual

# Genetic Algorithm def genetic\_algorithm():

population = initialize\_population(population\_size) best\_solution = None

best\_fitness = -np.inf

for generation in range(num\_generations):

fitness = evaluate\_fitness(population) # Track the best solution

current\_best\_index = np.argmax(fitness)

if fitness[current\_best\_index] > best\_fitness: best\_fitness = fitness[current\_best\_index] best\_solution = population[current\_best\_index]

# Create a new population new\_population = []

for \_ in range(population\_size): parent1 = select(population, fitness) parent2 = select(population, fitness)

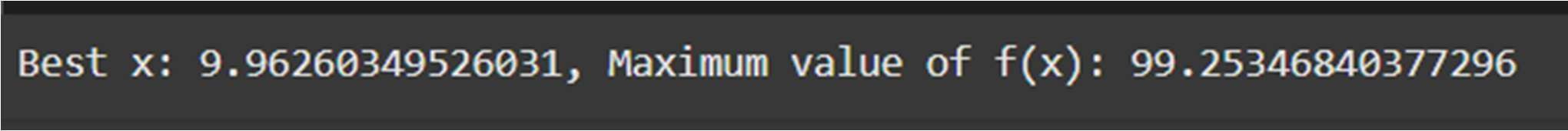
offspring = crossover(parent1, parent2) offspring = mutate(offspring) new\_population.append(offspring)

population = np.array(new\_population) return best\_solution, best\_fitness

# Run the Genetic Algorithm

best\_x, best\_value = genetic\_algorithm()

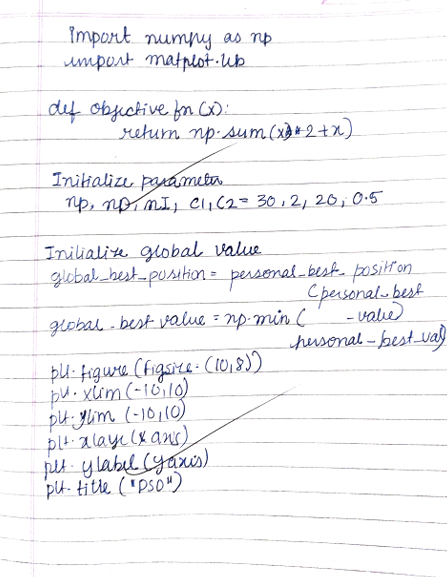
print(f"Best x: {best\_x}, Maximum value of f(x): {best\_value}")

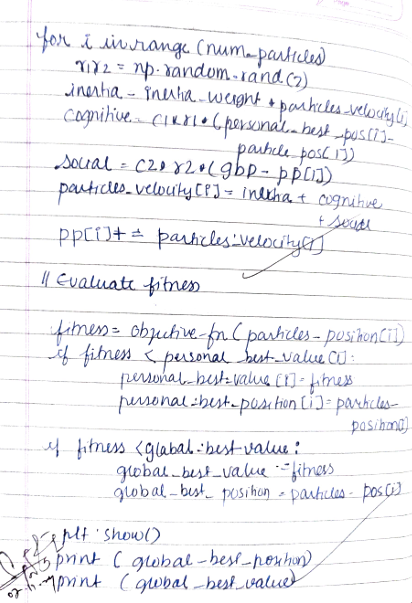


Algorithm:

# Program 2

## Particle Swarm Optimization for Function Optimization





Code:

import numpy as np

import matplotlib.pyplot as plt

# Objective function: f(x) = x^2 + 4x + 4 def objective\_function(x):

return x\*\*2 + 4\*x + 4

# PSO parameters

num\_particles = 30 # Number of particles

dimensions = 1 # Problem dimensionality (1D for this example) iterations = 100 # Number of iterations

w = 0.5 # Inertia weight

c1 = 1.5 # Cognitive coefficient

c2 = 1.5 # Social coefficient

# Initialize the particles

positions = np.random.uniform(-10, 10, size=(num\_particles, dimensions)) # Random positions velocities = np.random.uniform(-1, 1, size=(num\_particles, dimensions)) # Random velocities personal\_best\_positions = np.copy(positions) # Personal best positions

personal\_best\_scores = np.array([objective\_function(p) for p in positions]) # Personal best scores

# Global best (initially the best personal position)

global\_best\_position = personal\_best\_positions[np.argmin(personal\_best\_scores)] global\_best\_score = np.min(personal\_best\_scores)

# PSO Optimization loop

for iteration in range(iterations): for i in range(num\_particles):

# Update velocity

r1 = np.random.rand() r2 = np.random.rand()

velocities[i] = w \* velocities[i] + c1 \* r1 \* (personal\_best\_positions[i] - positions[i]) + c2 \* r2 \* (global\_best\_position - positions[i])

# Update position

positions[i] = positions[i] + velocities[i]

# Evaluate the objective function

current\_score = objective\_function(positions[i])

# Update personal best

if current\_score < personal\_best\_scores[i]: personal\_best\_scores[i] = current\_score personal\_best\_positions[i] = positions[i]

# Update global best

if current\_score < global\_best\_score: global\_best\_score = current\_score

global\_best\_position = positions[i]

# Optionally print the global best score during the iterations if iteration % 10 == 0:

print(f"Iteration {iteration}: Global Best Score = {global\_best\_score}")

# Final result

print(f"Final Global Best Position: {global\_best\_position}") print(f"Final Global Best Score: {global\_best\_score}")

# Plotting the results for visualization x = np.linspace(-10, 10, 400)

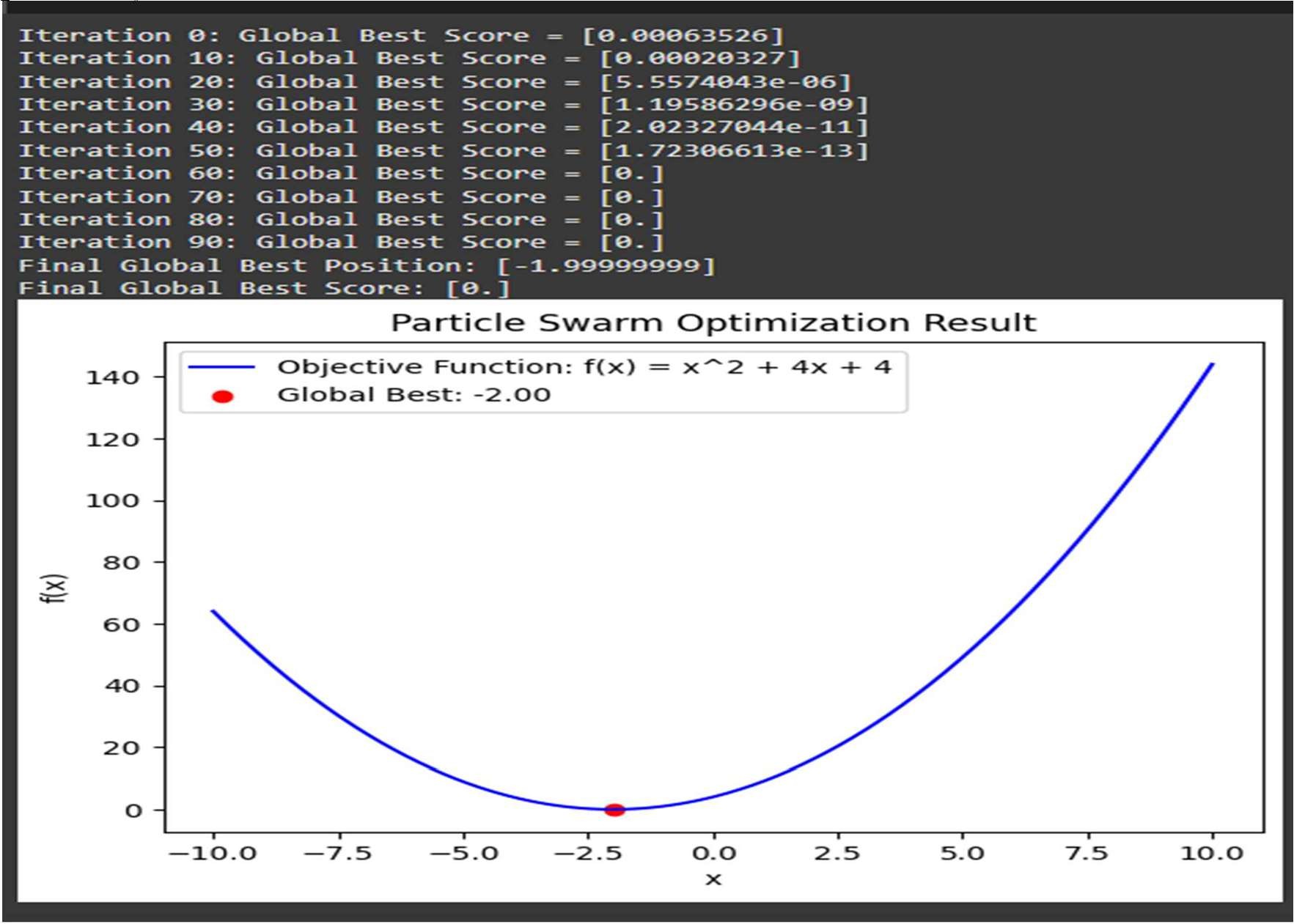
y = objective\_function(x)

plt.plot(x, y, label="Objective Function: f(x) = x^2 + 4x + 4", color='blue') plt.scatter(global\_best\_position, global\_best\_score, color='red', label=f"Global Best:

{global\_best\_position[0]:.2f}") plt.legend()

plt.title("Particle Swarm Optimization Result") plt.xlabel("x")

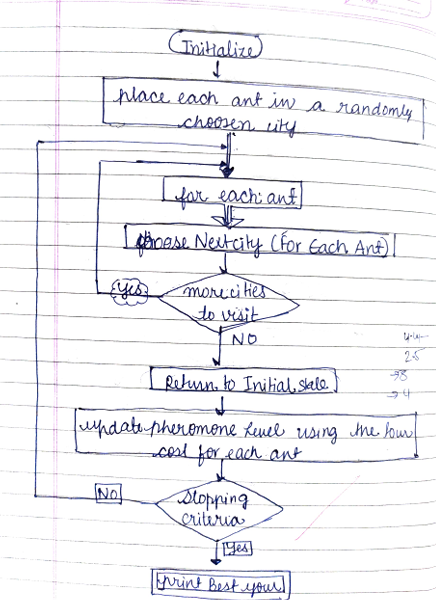
plt.ylabel("f(x)") plt.show()

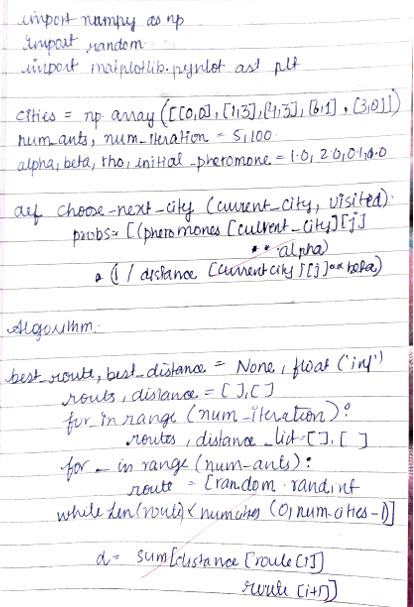


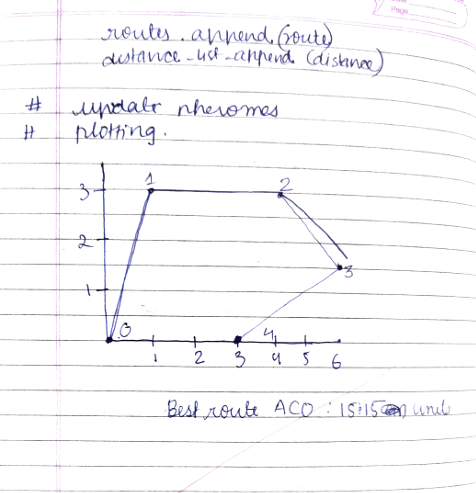
# Program 3

## Ant Colony Optimization for the Traveling Salesman Problem

Algorithm:







Code:

import random import math

import numpy as np

# Calculate the Euclidean distance between two cities def distance(city1, city2):

return math.sqrt((city1[0] - city2[0])\*\*2 + (city1[1] - city2[1])\*\*2)

# Ant Colony Optimization for TSP class AntColony:

def init (self, cities, num\_ants, alpha, beta, rho, iterations): self.cities = cities

self.num\_cities = len(cities) self.num\_ants = num\_ants

self.alpha = alpha # Influence of pheromone self.beta = beta # Influence of distance self.rho = rho # Pheromone evaporation rate self.iterations = iterations

self.pheromone = np.ones((self.num\_cities, self.num\_cities)) # Initial pheromone self.distances = np.zeros((self.num\_cities, self.num\_cities))

# Calculate the distance matrix for all pairs of cities for i in range(self.num\_cities):

for j in range(i + 1, self.num\_cities):

self.distances[i][j] = distance(self.cities[i], self.cities[j]) self.distances[j][i] = self.distances[i][j]

def probability(self, ant, city, visited):

"""Calculates the probability of moving to a next city.""" pheromone = self.pheromone[city]

heuristic = np.array([1.0 / self.distances[city][i] if i not in visited else 0 for i in range(self.num\_cities)])

pheromone\_heuristic = pheromone \*\* self.alpha \* heuristic \*\* self.beta pheromone\_heuristic[visited] = 0 # Ensure no city is visited twice

return pheromone\_heuristic / pheromone\_heuristic.sum() def run(self):

best\_distance = float('inf') best\_tour = None

# Iterate for a number of iterations for \_ in range(self.iterations):

all\_tours = [] all\_distances = []

# Each ant constructs a solution for ant in range(self.num\_ants):

visited = [0] # Start from city 0 tour = [0]

total\_distance = 0

# Construct the solution by visiting all cities while len(visited) < self.num\_cities:

city = visited[-1]

prob = self.probability(ant, city, visited)

next\_city = np.random.choice(range(self.num\_cities), p=prob) visited.append(next\_city)

tour.append(next\_city)

total\_distance += self.distances[city][next\_city]

# Add the return to the starting city

total\_distance += self.distances[visited[-1]][visited[0]]

# Track the best tour and distance all\_tours.append(tour) all\_distances.append(total\_distance)

if total\_distance < best\_distance: best\_distance = total\_distance best\_tour = tour

# Update pheromone trails

self.pheromone \*= (1 - self.rho) # Evaporate pheromone for ant in range(self.num\_ants):

for i in range(self.num\_cities - 1): city1 = all\_tours[ant][i]

city2 = all\_tours[ant][i + 1]

self.pheromone[city1][city2] += 1.0 / all\_distances[ant] # Pheromone reinforcement self.pheromone[city2][city1] += 1.0 / all\_distances[ant]

return best\_tour, best\_distance # Example usage

if name == " main ":

# Define cities as a list of (x, y) coordinates

cities = [(0, 0), (1, 3), (4, 3), (6, 1), (6, 5), (2, 7), (3, 4), (5, 2)]

# Set the ACO parameters num\_ants = 10

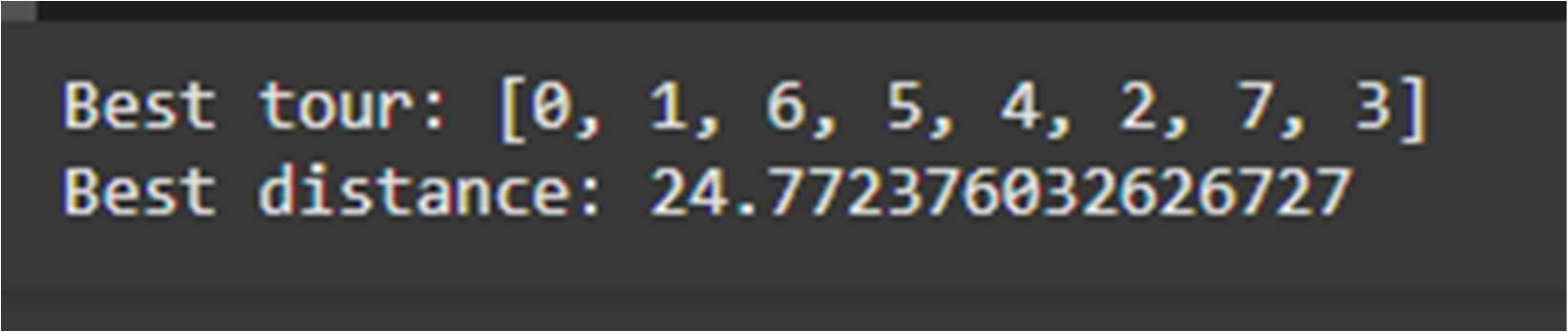
alpha = 1.0 # Pheromone importance beta = 2.0 # Heuristic importance rho = 0.1 # Pheromone evaporation

iterations = 100

# Create and run the ant colony optimizer

aco = AntColony(cities, num\_ants, alpha, beta, rho, iterations) best\_tour, best\_distance = aco.run()

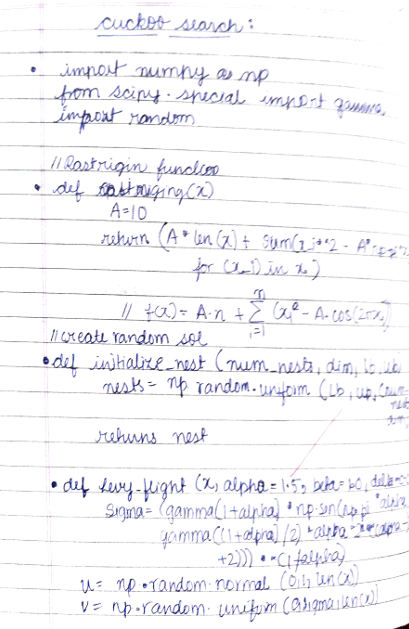
# Output the best solution found print("Best tour:", best\_tour) print("Best distance:", best\_distance)



Algorithm:

# Program 4

## Cuckoo Search (CS)





Code:

import numpy as np import math

# Sphere Function: f(x) = sum(x\_i^2) def sphere\_function(x):

return np.sum(x\*\*2)

# Lévy Flight function

def levy\_flight(Lambda, d):

# Lévy flight step size based on power-law distribution

sigma\_u = (math.gamma(1 + Lambda) \* np.sin(np.pi \* Lambda / 2) /

(math.gamma((1 + Lambda) / 2) \* Lambda \* 2\*\*((Lambda - 1) / 2)))\*\*(1 / Lambda) u = np.random.normal(0, sigma\_u, d)

v = np.random.normal(0, 1, d) step = u / np.abs(v)\*\*(1 / Lambda) return step

# Initialize the Cuckoo Search Algorithm

def cuckoo\_search(func, n\_nests, n\_dim, max\_iter, pa=0.25, alpha=0.01, lambda\_levy=1.5): # Initialize nests randomly

nests = np.random.uniform(-5, 5, (n\_nests, n\_dim)) # Bound the values between -5 and 5 for the Sphere function

fitness = np.apply\_along\_axis(func, 1, nests) # Calculate fitness of each nest

# Keep track of the best solution found so far best\_nest = nests[np.argmin(fitness)] best\_fitness = np.min(fitness)

for iteration in range(max\_iter):

# Generate new nests via Lévy flights

new\_nests = nests + alpha \* levy\_flight(lambda\_levy, n\_dim) # Ensure new nests are within bounds

new\_nests = np.clip(new\_nests, -5, 5)

# Evaluate new nests' fitness

new\_fitness = np.apply\_along\_axis(func, 1, new\_nests)

# Replace worst nests with new ones based on probability of discovery for i in range(n\_nests):

if np.random.rand() < pa: # Discovery probability nests[i] = new\_nests[i]

fitness[i] = new\_fitness[i]

# Update the best solution if we find a better one if np.min(fitness) < best\_fitness:

best\_fitness = np.min(fitness)

best\_nest = nests[np.argmin(fitness)]

# Output the current iteration's best solution return best\_nest, best\_fitness

# Set algorithm parameters

n\_nests = 50 # Number of nests (solutions)

n\_dim = 10 # Dimensionality of the problem (number of variables) max\_iter = 100 # Number of iterations

pa = 0.25 # Probability of discovery (abandoning the worst nests) alpha = 0.01 # Scaling factor for the Lévy flight

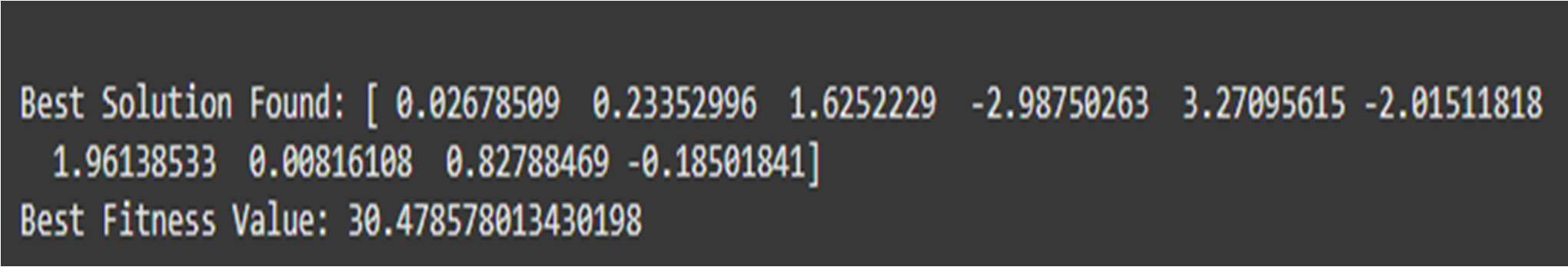
lambda\_levy = 1.5 # Exponent for Lévy flight distribution

# Run the Cuckoo Search Algorithm to minimize the Sphere Function

best\_solution, best\_value = cuckoo\_search(sphere\_function, n\_nests, n\_dim, max\_iter)

# Output the best solution found

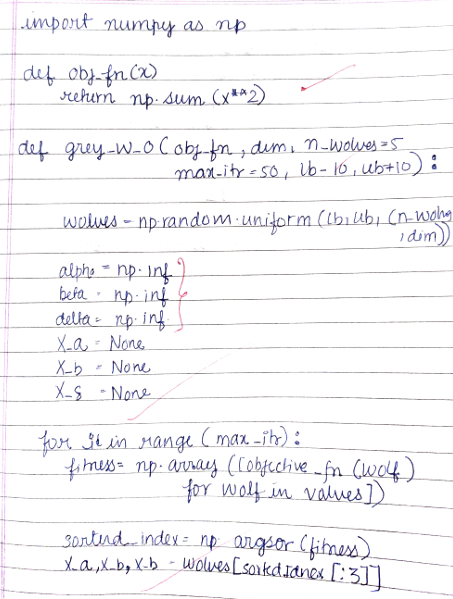
print("\nBest Solution Found:", best\_solution) print("Best Fitness Value:", best\_value)

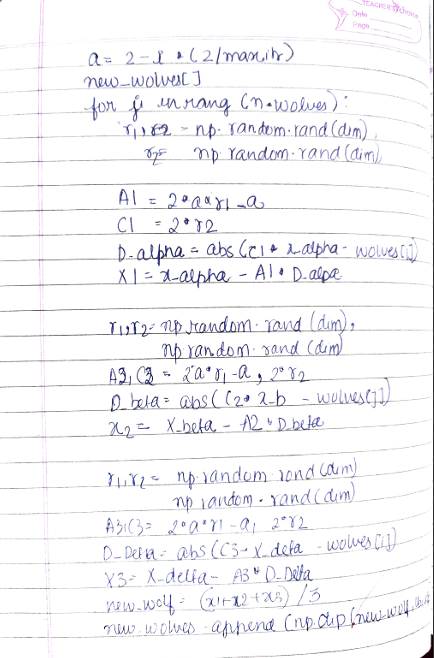


Algorithm:

# Program 5

## Grey Wolf Optimizer (GWO)





Code:

import numpy as np

# Define the objective function (Sphere function: sum(x^2)) def sphere(x):

return np.sum(x\*\*2)

# Grey Wolf Optimizer (GWO) class GWO:

def init (self, obj\_func, dim, pop\_size, max\_iter, lb, ub): self.obj\_func = obj\_func # Objective function to minimize self.dim = dim # Number of dimensions

self.pop\_size = pop\_size # Number of wolves in the population self.max\_iter = max\_iter # Maximum number of iterations self.lb = lb # Lower bound of search space

self.ub = ub # Upper bound of search space

# Initialize the wolves' positions randomly within bounds

self.position = np.random.uniform(self.lb, self.ub, (self.pop\_size, self.dim))

self.fitness = np.array([self.obj\_func(ind) for ind in self.position]) # Initial fitness of all wolves

# Initialize the alpha, beta, and delta wolves' positions and fitness self.alpha\_pos = np.zeros(self.dim)

self.beta\_pos = np.zeros(self.dim) self.delta\_pos = np.zeros(self.dim)

self.alpha\_score = float('inf') # Best score (we minimize, so start with infinity) self.beta\_score = float('inf')

self.delta\_score = float('inf')

def update\_position(self, alpha, beta, delta, a, A, C, position):

# Update the position of a single wolf based on the positions of alpha, beta, delta r1 = np.random.random(self.dim)

r2 = np.random.random(self.dim)

# Update position using the equation D\_alpha = abs(C[0] \* r1 - position - alpha) D\_beta = abs(C[1] \* r1 - position - beta) D\_delta = abs(C[2] \* r1 - position - delta)

X1 = alpha - A[0] \* D\_alpha X2 = beta - A[1] \* D\_beta X3 = delta - A[2] \* D\_delta

# New position is the average of the three components new\_position = (X1 + X2 + X3) / 3

return new\_position

def optimize(self):

for t in range(self.max\_iter):

# Update parameters A and C based on the iteration

a = 2 - t \* (2 / self.max\_iter) # Declining over iterations A = np.random.uniform(-a, a, 3)

C = np.random.uniform(0, 2, 3)

# Evaluate fitness and update the alpha, beta, delta wolves for i in range(self.pop\_size):

fitness = self.obj\_func(self.position[i])

if fitness < self.alpha\_score: self.alpha\_score = fitness self.alpha\_pos = self.position[i]

elif fitness < self.beta\_score: self.beta\_score = fitness self.beta\_pos = self.position[i]

elif fitness < self.delta\_score: self.delta\_score = fitness self.delta\_pos = self.position[i]

# Update the position of each wolf in the population for i in range(self.pop\_size):

# Update the position of wolf i

self.position[i] = self.update\_position(self.alpha\_pos, self.beta\_pos, self.delta\_pos, a, A, C, self.position[i])

# Ensure the new position stays within the bounds self.position[i] = np.clip(self.position[i], self.lb, self.ub)

# Optionally print the progress

print(f"Iteration {t+1}/{self.max\_iter} - Best Fitness: {self.alpha\_score}") return self.alpha\_pos, self.alpha\_score

# Set problem-specific parameters

dim = 10 # Number of dimensions (variables in the function) pop\_size = 50 # Number of wolves in the population

max\_iter = 100 # Number of iterations

lb = -5.12 # Lower bound of search space ub = 5.12 # Upper bound of search space

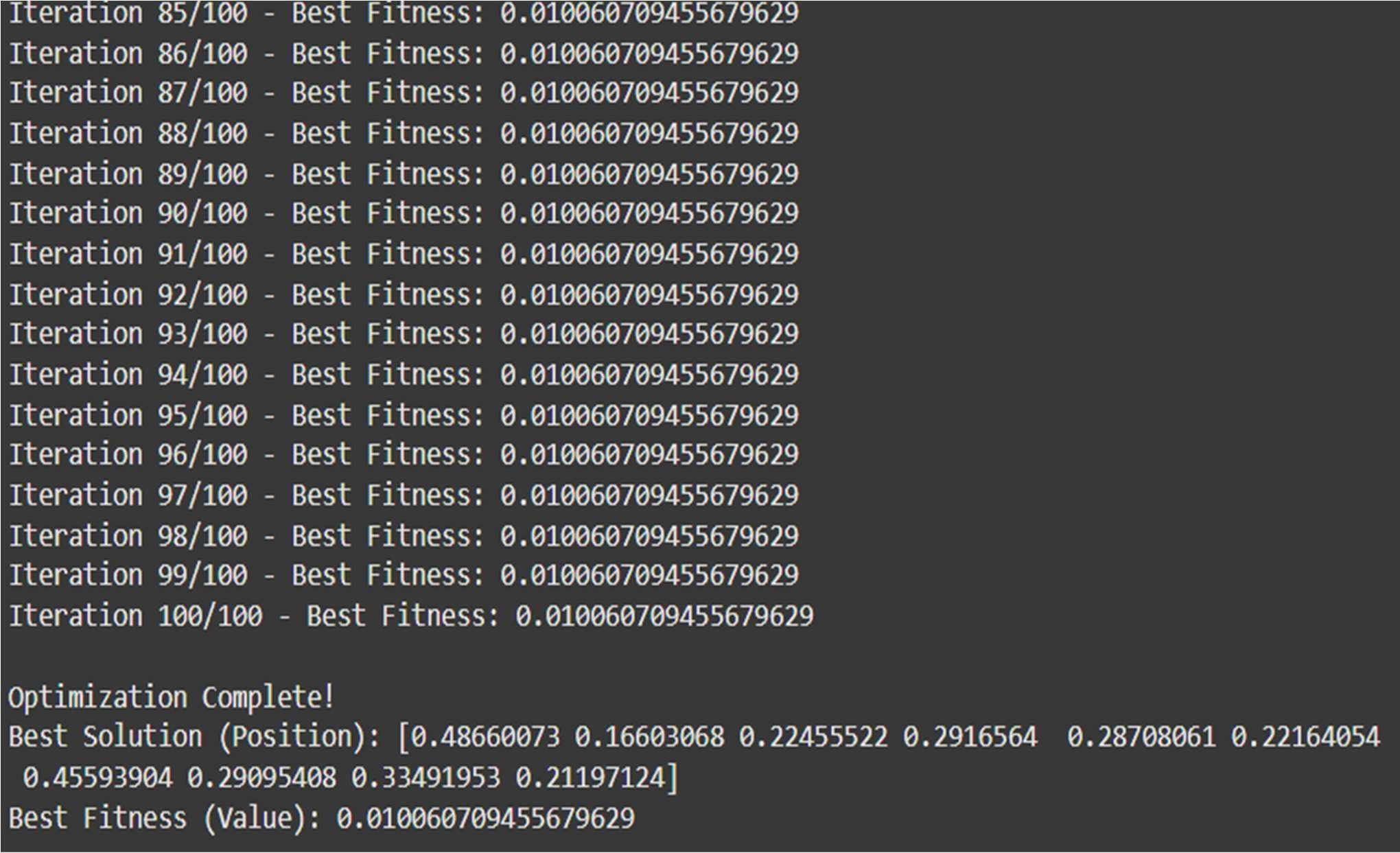
# Create an instance of the GWO class

gwo = GWO(obj\_func=sphere, dim=dim, pop\_size=pop\_size, max\_iter=max\_iter, lb=lb, ub=ub)

# Run the optimization

best\_pos, best\_score = gwo.optimize() print("\nOptimization Complete!")

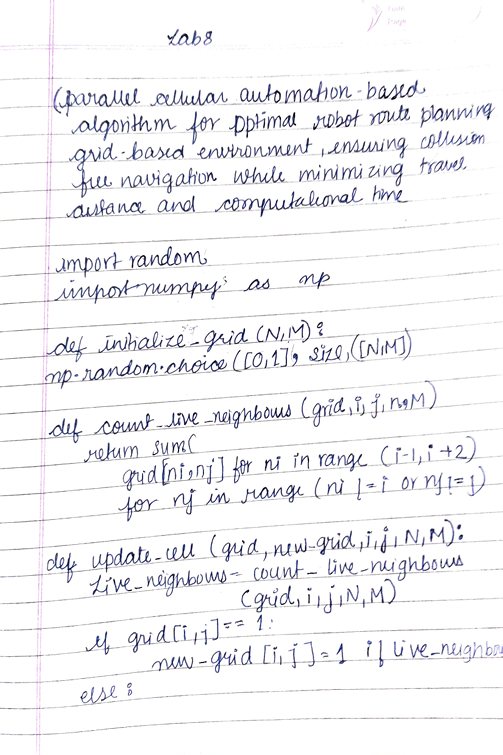
print("Best Solution (Position):", best\_pos) print("Best Fitness (Value):", best\_score)

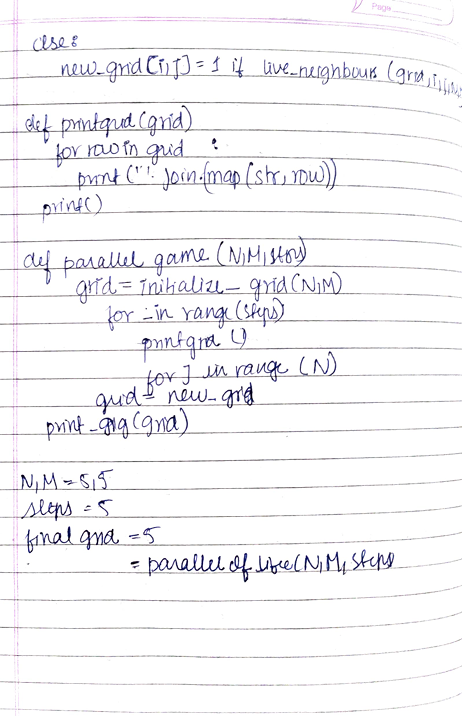


Algorithm:

# Program 6

## Parallel Cellular Algorithms and Programs





Code:

import numpy as np import random

# Objective function: f(x) = x^2 + 4x + 4 def objective\_function(x):

return x\*\*2 + 4\*x + 4

# Parameters

grid\_size = 10 # Number of cells in the grid (1D grid here for simplicity) num\_iterations = 100 # Number of iterations

neighborhood\_radius = 1 # Neighborhood range (cell's neighbors) mutation\_rate = 0.1 # Probability of mutation

# Initialize the grid: random values within a range (-10, 10) def initialize\_grid(grid\_size):

return np.random.uniform(-10, 10, grid\_size)

# Fitness evaluation for each cell def evaluate\_fitness(grid):

return np.array([objective\_function(cell) for cell in grid])

# Update the state of each cell based on its neighbors def update\_states(grid, fitness, neighborhood\_radius):

new\_grid = np.copy(grid) for i in range(grid\_size):

# Get the neighbors (with wraparound at boundaries) left = (i - neighborhood\_radius) % grid\_size

right = (i + neighborhood\_radius) % grid\_size

# Ensure that the indices are valid if left <= right:

neighbors = grid[left:right+1] fitness\_neighbors = fitness[left:right+1]

else:

# Handle wraparound correctly

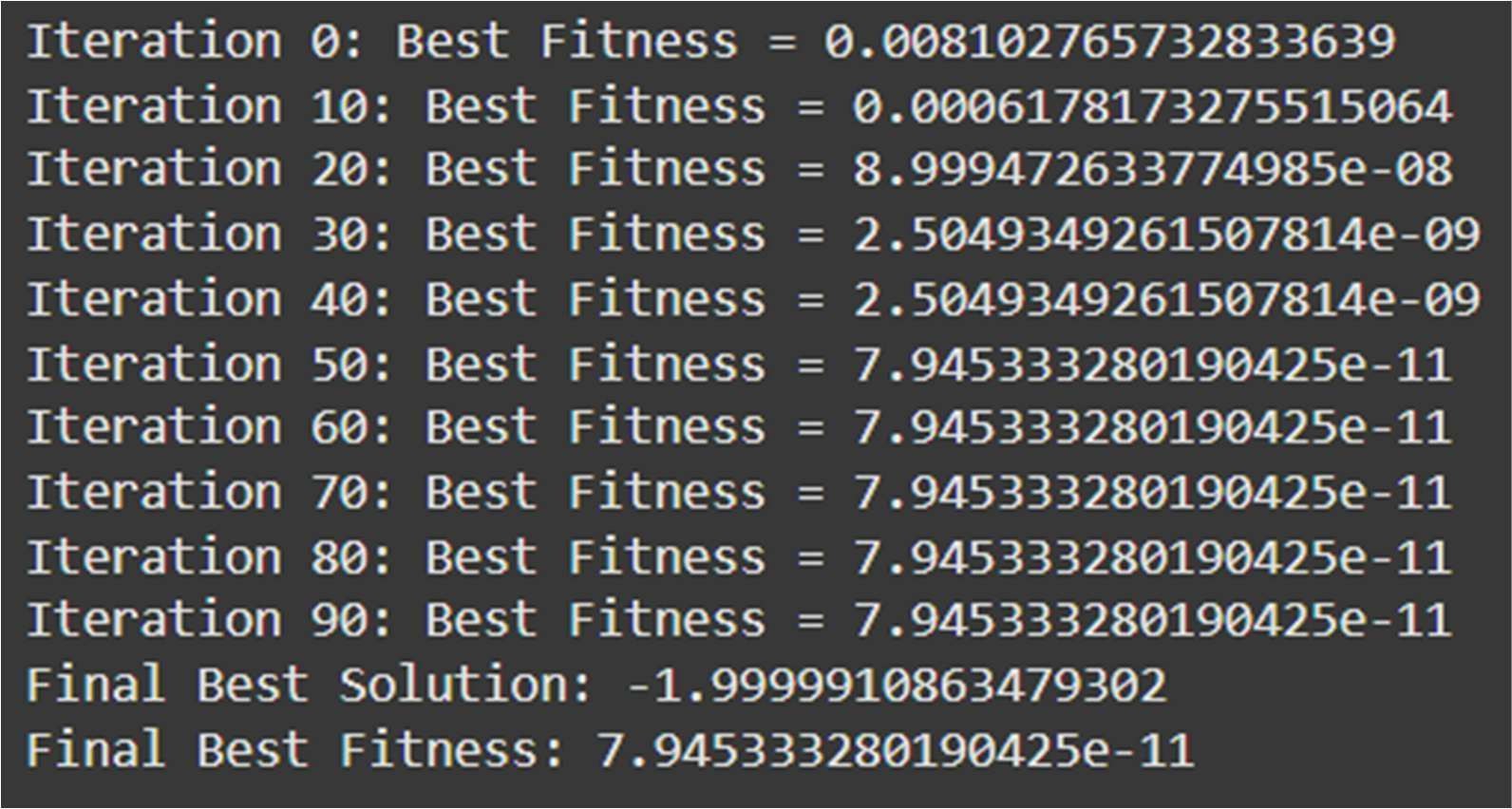
neighbors = np.concatenate([grid[left:], grid[:right+1]]) fitness\_neighbors = np.concatenate([fitness[left:], fitness[:right+1]])

# Update rule: take the average of neighbors if their fitness is better best\_neighbor = neighbors[np.argmin(fitness\_neighbors)]

# Update rule: Apply smaller mutation

new\_grid[i] = best\_neighbor + random.uniform(-mutation\_rate / 10, mutation\_rate / 10) #

Reduced mutation impact return new\_grid

# Main Cellular Algorithm Function def parallel\_cellular\_algorithm():

# Initialize grid

grid = initialize\_grid(grid\_size)

best\_solution = None best\_fitness = float('inf')

# Iterate through generations

for iteration in range(num\_iterations): fitness = evaluate\_fitness(grid)

# Track the best solution current\_best\_index = np.argmin(fitness)

if fitness[current\_best\_index] < best\_fitness: best\_fitness = fitness[current\_best\_index] best\_solution = grid[current\_best\_index]

# Update states based on neighbors

grid = update\_states(grid, fitness, neighborhood\_radius)

# Output the best solution at regular intervals if iteration % 10 == 0:

print(f"Iteration {iteration}: Best Fitness = {best\_fitness}") return best\_solution, best\_fitness

# Run the Parallel Cellular Algorithm

best\_solution, best\_fitness = parallel\_cellular\_algorithm()

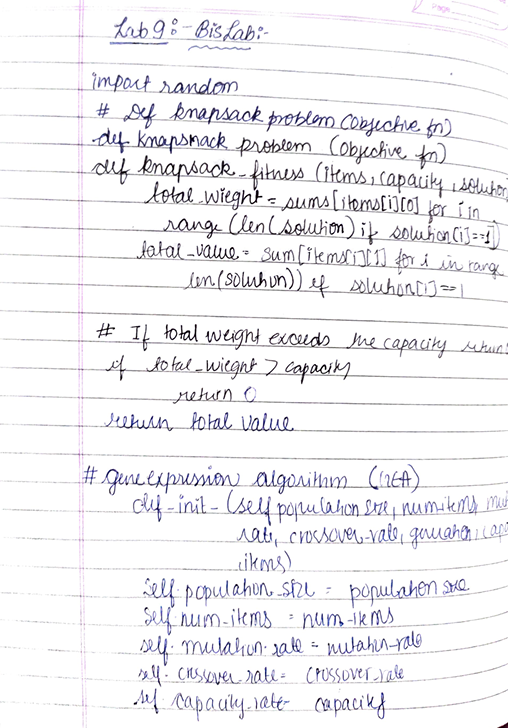
# Output the final best solution

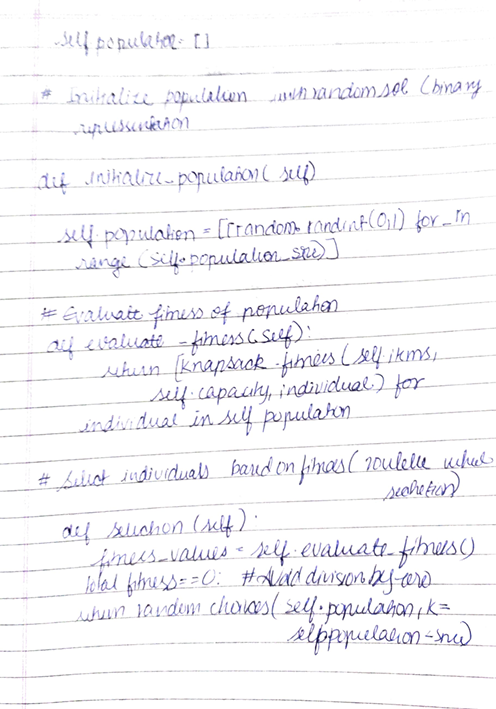
print(f"Final Best Solution: {best\_solution}") print(f"Final Best Fitness: {best\_fitness}")

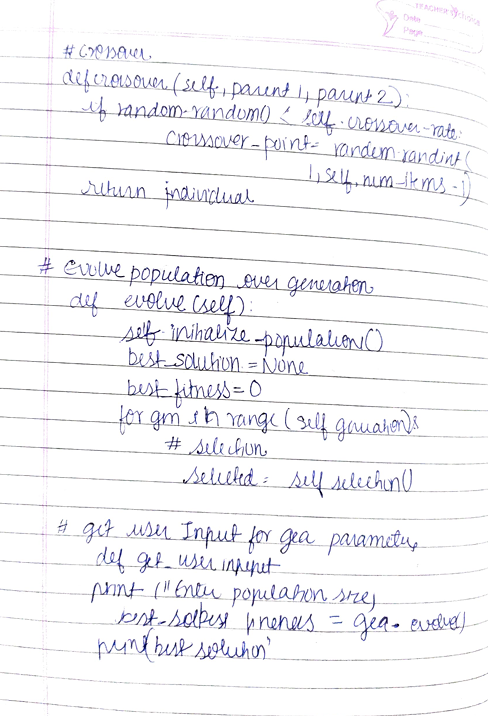
Algorithm:

# Program 7

## Optimization via Gene Expression Algorithms







Code:

import numpy as np import random

# Objective function: f(x) = x^2 + 4x + 4 def objective\_function(x):

return x\*\*2 + 4\*x + 4

# GEA parameters

population\_size = 30 # Number of genetic sequences (solutions)

num\_genes = 1 # Number of genes in each sequence (1D optimization in this case) mutation\_rate = 0.05 # Probability of mutation

crossover\_rate = 0.7 # Probability of crossover num\_generations = 100 # Number of generations

# Initialize Population: Generate random genetic sequences (chromosomes) def initialize\_population(population\_size, num\_genes):

return np.random.uniform(-10, 10, (population\_size, num\_genes))

# Fitness Evaluation: Evaluate fitness of each sequence def evaluate\_fitness(population):

return np.array([objective\_function(individual[0]) for individual in population])

# Selection: Tournament Selection def select(population, fitness):

selected\_indices = np.random.choice(len(population), size=2, replace=False) return population[selected\_indices[np.argmin(fitness[selected\_indices])]]

# Crossover: Single-point crossover def crossover(parent1, parent2):

if random.random() < crossover\_rate:

# Random crossover point (for 1D, just average)

return (parent1 + parent2) / 2 return parent1

# Mutation: Introduce small random changes def mutate(individual):

if random.random() < mutation\_rate:

return individual + np.random.uniform(-1, 1) return individual

# Gene Expression: Translate genetic sequence to a functional solution def gene\_expression(population):

return population

# Main GEA Function: Optimization Loop def gene\_expression\_algorithm():

# Initialize population

population = initialize\_population(population\_size, num\_genes)

best\_solution = None best\_fitness = float('inf')

# Track the best solution through generations for generation in range(num\_generations):

fitness = evaluate\_fitness(population)

# Track the best solution current\_best\_index = np.argmin(fitness)

if fitness[current\_best\_index] < best\_fitness: best\_fitness = fitness[current\_best\_index] best\_solution = population[current\_best\_index]

# Create a new population new\_population = []

for \_ in range(population\_size): parent1 = select(population, fitness) parent2 = select(population, fitness)

offspring = crossover(parent1, parent2) offspring = mutate(offspring) new\_population.append(offspring)

population = np.array(new\_population)

# Gene expression (translation of genetic sequence to solutions) population = gene\_expression(population)

# Output the best solution at regular intervals if generation % 10 == 0:

print(f"Generation {generation}: Best Fitness = {best\_fitness}") return best\_solution, best\_fitness

# Run the GEA

best\_solution, best\_fitness = gene\_expression\_algorithm()

# Output final best solution

print(f"Final Best Solution: {best\_solution}") print(f"Final Best Fitness: {best\_fitness}")

