VISVESVARAYA TECHNOLOGICAL UNIVERSITY

"JnanaSangama", Belgaum -590014, Karnataka.

LAB RECORD

Bio Inspired Systems (23CS5BSBIS)

Submitted by

Ameena Yasmeen(1BM23CS027)

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
Aug-2025 to Dec-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 **Ameena Yasmeen(1BM23CS027),** 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.

Lab faculty In charge Name Assistant Professor Department of CSE, BMSCE	Dr.Namratha M Professor & HOD Department of CSE, BMSCE

Index

Github Link:

https://github.com/Ameena1BM23CS27/BIS-Lab

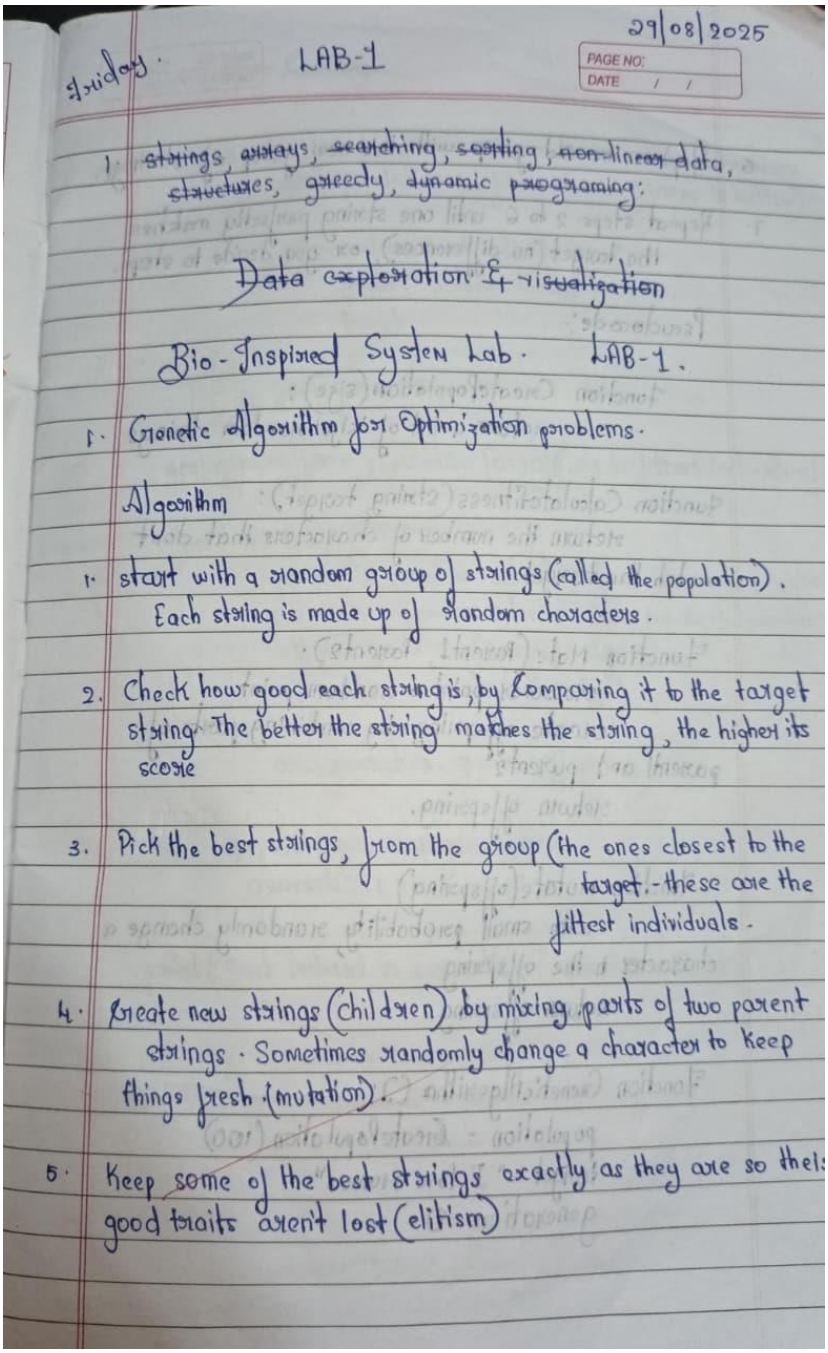
Program 1

Genetic Algorithm for Optimization Problems

Problem Statement

Genetic Algorithms (GA) are inspired by the process of natural selection and genetics, where the fittest individuals are selected for reproduction to produce the next generation. GAs are widely used for solving optimization and search problems.
Implement a Genetic Algorithm using Python to solve a basic optimization problem, such as finding the maximum value of a mathematical function.

Algorithm:



Friday. LAB-1 29/08/2025

PAGE NO:
DATE / /

1. ~~strings, arrays, searching, sorting, non-linear data, structures, greedy, dynamic programming:~~

~~Data exploration & visualization~~

Bio-Inspired System Lab. LAB-1.

1. Genetic Algorithm for Optimization problems.

Algorithm

1. start with a random group of strings (called the population). Each string is made up of random characters.

2. Check how good each string is, by comparing it to the target string. The better the string matches the string, the higher its score

3. Pick the best strings, from the group (the ones closest to the target. - these are the fittest individuals.

4. Create new strings (children) by mixing parts of two parent strings. Sometimes randomly change a character to keep things fresh (mutation).

5. Keep some of the best strings exactly as they are so their good traits aren't lost (elitism)

6. Replace the old group with this new group of strings.
7. Repeat steps 2 to 6 until one string perfectly matches the target (no differences), or you decide to stop.

Pseudocode:-

```
function CreatePopulation(size):
    return a list of 'size' random strings

function CalculateFitness(string, target):
    return the number of characters that don't match the target

function Mate(Parent1, Parent2):
    crossover_point = random point
    create offspring by combining parts of parent1 and parent2
    return offspring.

function mutate(offspring)
    with small probability, randomly change a character in the offspring
    return offspring

function GeneticAlgorithm():
    population = CreatePopulation(100)
    target = "I love momos"
    generations = 0
```

Competitive Coding.

PAGE NO.
DATE / /

```
while not solution_found and generations < max_generations:
        sort population by fitness(lower fitness is better)

        if fitness of the best individual == 0;
            print("Solution found!")
            break

        new_generation = []

    new_generation = population[0:10][# top 10 fittest individual]

    for i in range(90):
        parent1 = Random selection from top 50% population
        parent2 = Random selection from top 50% population
        offspring = Mate(parent1, parent2)
        offspring = mutate(offspring)
        newgeneration.append(offspring)

    population = new_generation
    generation += 1

print("Best Individ in population:", population[0])
```

Code:

```
import random
import math

# 1. Define the problem — function to optimize
def fitness_function(x):
    # Example function: f(x) = x * sin(10πx) + 1.0 (maximize this)
    return x * math.sin(10 * math.pi * x) + 1.0

# 2. Initialize Parameters
POP_SIZE = 20          # Number of individuals in population
GENES = 16             # Number of bits to represent a chromosome
MUTATION_RATE = 0.01   # Probability of mutation
CROSSOVER_RATE = 0.7   # Probability of crossover
GENERATIONS = 50       # Number of generations
X_MIN, X_MAX = 0, 1    # Range of x values

# Helper functions
def decode(chromosome):
    """Convert binary chromosome to a real value in range [X_MIN,
X_MAX]"""
    decimal_value = int(chromosome, 2)
    return X_MIN + (decimal_value / (2**GENES - 1)) * (X_MAX - X_MIN)

def create_individual():
    """Generate a random binary string (chromosome)"""
    return ''.join(random.choice('01') for _ in range(GENES))

def crossover(parent1, parent2):
    """Single-point crossover"""
    if random.random() < CROSSOVER_RATE:
        point = random.randint(1, GENES - 1)
        return parent1[:point] + parent2[point:], parent2[:point] +
parent1[point:]
    return parent1, parent2

def mutate(chromosome):
    """Random bit flip mutation"""
    chromosome = list(chromosome)
    for i in range(GENES):
        if random.random() < MUTATION_RATE:
            chromosome[i] = '1' if chromosome[i] == '0' else '0'
    return ''.join(chromosome)

# 3. Create initial population
population = [create_individual() for _ in range(POP_SIZE)]

best_solution = None
```

```
best_fitness = float('-inf')

# 4–8. Main GA loop
for generation in range(GENERATIONS):
    # Evaluate fitness for each individual
    fitness_scores = []
    for individual in population:
        x = decode(individual)
        fitness = fitness_function(x)
        fitness_scores.append(fitness)

        if fitness > best_fitness:
            best_fitness = fitness
            best_solution = individual

    # Print progress
    print(f"Generation {generation + 1} | Best Fitness:
{best_fitness:.5f}")

    # 5. Selection (Roulette Wheel Selection)
    total_fitness = sum(fitness_scores)
    probabilities = [f / total_fitness for f in fitness_scores]

    def select_parent():
        r = random.random()
        cumulative = 0
        for i, prob in enumerate(probabilities):
            cumulative += prob
            if r <= cumulative:
                return population[i]

    # 6. Crossover and 7. Mutation
    new_population = []
    while len(new_population) < POP_SIZE:
        parent1 = select_parent()
        parent2 = select_parent()
        offspring1, offspring2 = crossover(parent1, parent2)
        new_population.append(mutate(offspring1))
        if len(new_population) < POP_SIZE:
            new_population.append(mutate(offspring2))

    population = new_population

# 9. Output the best solution
best_x = decode(best_solution)
print("\n Best Solution Found:")
print(f"x = {best_x:.5f}")
print(f"Fitness = {best_fitness:.5f}")
```

Output:

```
OUTPUT:

Generation 1 | Best Fitness: 1.62960
Generation 2 | Best Fitness: 1.62960
Generation 3 | Best Fitness: 1.84872
Generation 4 | Best Fitness: 1.84913
Generation 5 | Best Fitness: 1.85005
Generation 6 | Best Fitness: 1.85005
Generation 7 | Best Fitness: 1.85005
Generation 8 | Best Fitness: 1.85005
Generation 9 | Best Fitness: 1.85005
Generation 10 | Best Fitness: 1.85005
Generation 11 | Best Fitness: 1.85059
Generation 12 | Best Fitness: 1.85059
Generation 13 | Best Fitness: 1.85059
Generation 14 | Best Fitness: 1.85059
Generation 15 | Best Fitness: 1.85059
Generation 16 | Best Fitness: 1.85059
Generation 17 | Best Fitness: 1.85059
Generation 18 | Best Fitness: 1.85059
Generation 19 | Best Fitness: 1.85059
Generation 20 | Best Fitness: 1.85059
Generation 21 | Best Fitness: 1.85059
Generation 22 | Best Fitness: 1.85059
Generation 23 | Best Fitness: 1.85059
Generation 24 | Best Fitness: 1.85059
Generation 25 | Best Fitness: 1.85059
Generation 26 | Best Fitness: 1.85059
Generation 27 | Best Fitness: 1.85059
Generation 28 | Best Fitness: 1.85059
Generation 29 | Best Fitness: 1.85059
Generation 30 | Best Fitness: 1.85059
Generation 31 | Best Fitness: 1.85059
Generation 32 | Best Fitness: 1.85059
Generation 33 | Best Fitness: 1.85059
Generation 34 | Best Fitness: 1.85059
Generation 35 | Best Fitness: 1.85059
Generation 36 | Best Fitness: 1.85059
Generation 37 | Best Fitness: 1.85059
Generation 38 | Best Fitness: 1.85059
Generation 39 | Best Fitness: 1.85059
Generation 40 | Best Fitness: 1.85059
Generation 41 | Best Fitness: 1.85059
Generation 42 | Best Fitness: 1.85059
Generation 43 | Best Fitness: 1.85059
Generation 44 | Best Fitness: 1.85059
Generation 45 | Best Fitness: 1.85059
Generation 46 | Best Fitness: 1.85059
Generation 47 | Best Fitness: 1.85059
Generation 48 | Best Fitness: 1.85059
Generation 49 | Best Fitness: 1.85059
Generation 50 | Best Fitness: 1.85059

Best Solution Found:
x = 0.85122
Fitness = 1.85059
```

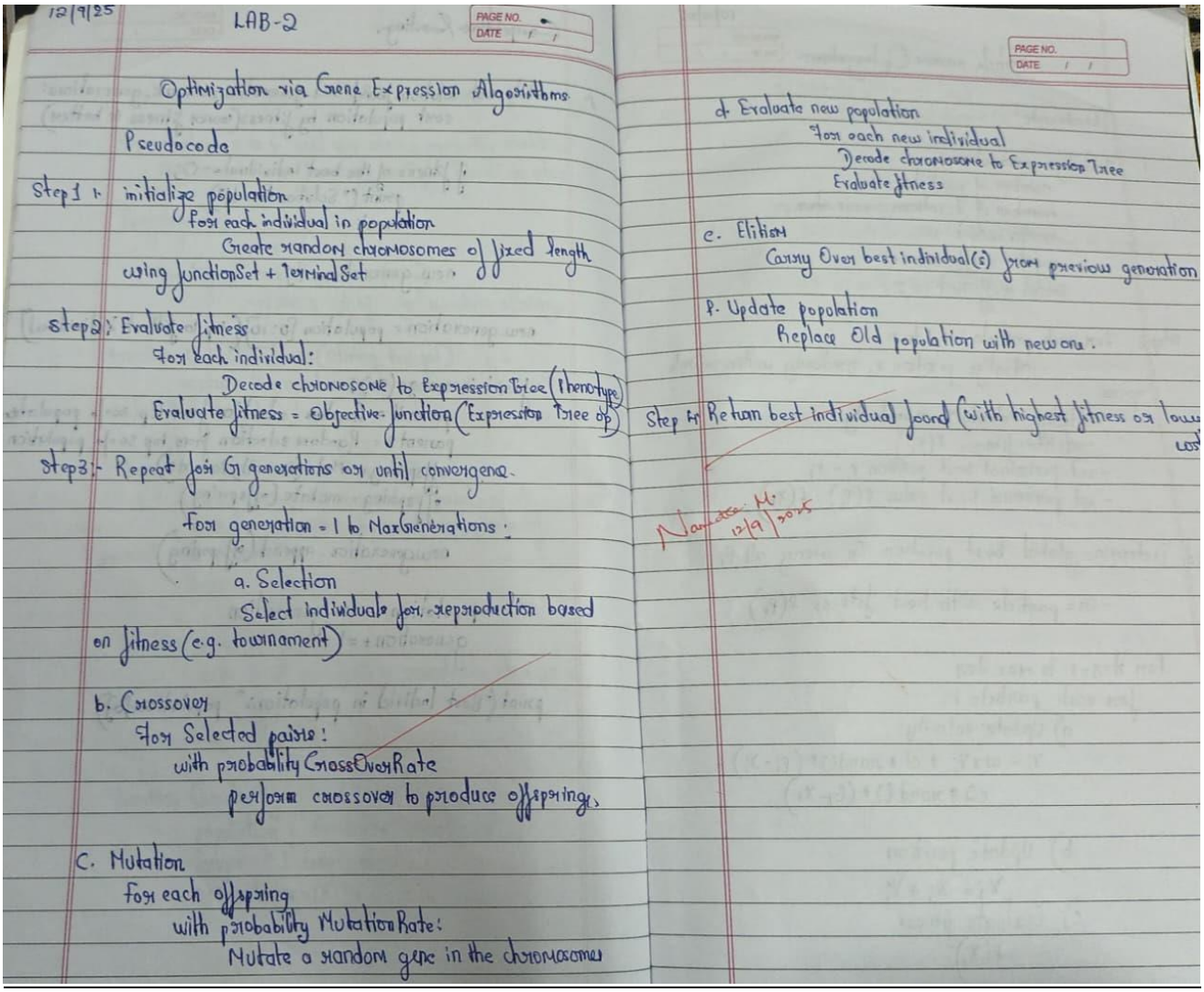
Program 2

Optimization via Gene Expression Algorithms:

Problem Statement

Gene Expression Algorithms (GEA) are inspired by the biological process of gene expression in living organisms. This process involves the translation of genetic information encoded in DNA into functional proteins. In GEA, solutions to optimization problems are encoded in a manner similar to genetic sequences. The algorithm evolves these solutions through selection, crossover, mutation, and gene expression to find optimal or near-optimal solutions. GEA is effective for solving complex optimization problems in various domains, including engineering, data analysis, and machine learning.

Algorithm:



12/9/25 LAB-2

Optimization via Gene Expression Algorithms

Pseudocode

Step 1 :- initialize population
for each individual in population
Create random chromosomes of fixed length
using functionSet + terminalSet

Step 2:- Evaluate fitness
for each individual:
Decode chromosome to expressionTree (phenotype)
Evaluate fitness = objective function (expression tree of)

Step 3:- Repeat for G generations or until convergence.
for generation = 1 to MaxGenerations:
a. Selection
Select individuals for reproduction based on fitness (e.g. tournament)

b. Crossover
for selected pairs:
with probability CrossOverRate
perform crossover to produce offsprings

c. Mutation
for each offspring
with probability MutationRate:
Mutate a random gene in the chromosomes

d. Evaluate new population
for each new individual
Decode chromosome to expressionTree
Evaluate fitness

e. Elition
Carry over best individual(s) from previous generation

f. Update population
Replace old population with new one.

Step 4:- Return best individual found (with highest fitness or lowest cost)

Namitha M 12/9/2025

Code:

```
import random
import math
def objective_function(x):
"""Example mathematical function to maximize."""
return x * math.sin(10 * math.pi * x) + 1.0
POP_SIZE = 20
NUM_GENES = 16
MUTATION_RATE = 0.05
CROSSOVER_RATE = 0.7
GENERATIONS = 50
X_MIN, X_MAX = 0, 1
def create_gene():
# Population size
# Number of genes per chromosome
# Mutation probability
# Crossover probability
# Number of generations
# Search range
"""Each gene is a binary value (0 or 1)."""
return random.choice(['0', '1'])
def create_chromosome():
"""A chromosome is a list of genes."""
return [create_gene() for _ in range(NUM_GENES)]

def create_population():
    """Generate initial population."""
    return [create_chromosome() for _ in range(POP_SIZE)]
def decode(chromosome):
    """Translate genes into a real number (phenotype)."""
    value = int(".join(chromosome), 2)
    return X_MIN + (value / (2**NUM_GENES - 1)) * (X_MAX - X_MIN)

def fitness(chromosome):
    """Compute the fitness value of an individual."""
    x = decode(chromosome)
    return objective_function(x)
def select_parent(population, fitnesses):
    total_fit = sum(fitnesses)
    pick = random.uniform(0, total_fit)
    current = 0
    for i, f in enumerate(fitnesses):
        current += f
        if current > pick:
            return population[i]
def crossover(parent1, parent2):
    if random.random() < CROSSOVER_RATE:
        point = random.randint(1, NUM_GENES - 1)
        child1 = parent1[:point] + parent2[point:]
        child2 = parent2[:point] + parent1[point:]
        return child1, child2
    return parent1[:], parent2[:]
```

```
def mutate(chromosome):
    for i in range(NUM_GENES):
        if random.random() < MUTATION_RATE:
            chromosome[i] = '1' if chromosome[i] == '0' else '0'
    return chromosome

# 8. Gene Expression (Translate Genotype → Phenotype)
def express(chromosome):
    """Expression phase maps genetic code to actual function value."""
    x = decode(chromosome)
    return x, objective_function(x)

# 9. Iterate through Generations
population = create_population()
best_solution = None
best_fitness = float('-inf')

for gen in range(GENERATIONS):
    fitnesses = [fitness(individual) for individual in population]

    #  for i, f in enumerate(fitnesses):
        if f > best_fitness:
            best_fitness = f
            best_solution = population[i]

    print(f"Generation {gen + 1} | Best Fitness: {best_fitness:.5f}")

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

    population = new_population[:POP_SIZE]
# 10. Output the Best Solution
best_x, best_y = express(best_solution)
print("\n Best Solution Found:")
print(f"x = {best_x:.5f}")
print(f"Fitness = {best_y:.5f}")
```

Output:

```
Generation 1 | Best Fitness: 1.62960
Generation 2 | Best Fitness: 1.62960
Generation 3 | Best Fitness: 1.84872
Generation 4 | Best Fitness: 1.84913
Generation 5 | Best Fitness: 1.85005
Generation 6 | Best Fitness: 1.85005
Generation 7 | Best Fitness: 1.85005
Generation 8 | Best Fitness: 1.85005
Generation 9 | Best Fitness: 1.85005
Generation 10 | Best Fitness: 1.85005
Generation 11 | Best Fitness: 1.85059
Generation 12 | Best Fitness: 1.85059
Generation 13 | Best Fitness: 1.85059
Generation 14 | Best Fitness: 1.85059
Generation 15 | Best Fitness: 1.85059
Generation 16 | Best Fitness: 1.85059
Generation 17 | Best Fitness: 1.85059
Generation 18 | Best Fitness: 1.85059
Generation 19 | Best Fitness: 1.85059
Generation 20 | Best Fitness: 1.85059
Generation 21 | Best Fitness: 1.85059
Generation 22 | Best Fitness: 1.85059
Generation 23 | Best Fitness: 1.85059
Generation 24 | Best Fitness: 1.85059
Generation 25 | Best Fitness: 1.85059
Generation 26 | Best Fitness: 1.85059
Generation 27 | Best Fitness: 1.85059
Generation 28 | Best Fitness: 1.85059
Generation 29 | Best Fitness: 1.85059
Generation 30 | Best Fitness: 1.85059
Generation 31 | Best Fitness: 1.85059
Generation 32 | Best Fitness: 1.85059
Generation 33 | Best Fitness: 1.85059
Generation 34 | Best Fitness: 1.85059
Generation 35 | Best Fitness: 1.85059
Generation 36 | Best Fitness: 1.85059
Generation 37 | Best Fitness: 1.85059
Generation 38 | Best Fitness: 1.85059
Generation 39 | Best Fitness: 1.85059
Generation 40 | Best Fitness: 1.85059
Generation 41 | Best Fitness: 1.85059
Generation 42 | Best Fitness: 1.85059
Generation 43 | Best Fitness: 1.85059
Generation 44 | Best Fitness: 1.85059
Generation 45 | Best Fitness: 1.85059
Generation 46 | Best Fitness: 1.85059
Generation 47 | Best Fitness: 1.85059
Generation 48 | Best Fitness: 1.85059
Generation 49 | Best Fitness: 1.85059
Generation 50 | Best Fitness: 1.85059

🧠 Best Solution Found:
x = 0.85122
Fitness = 1.85059
```

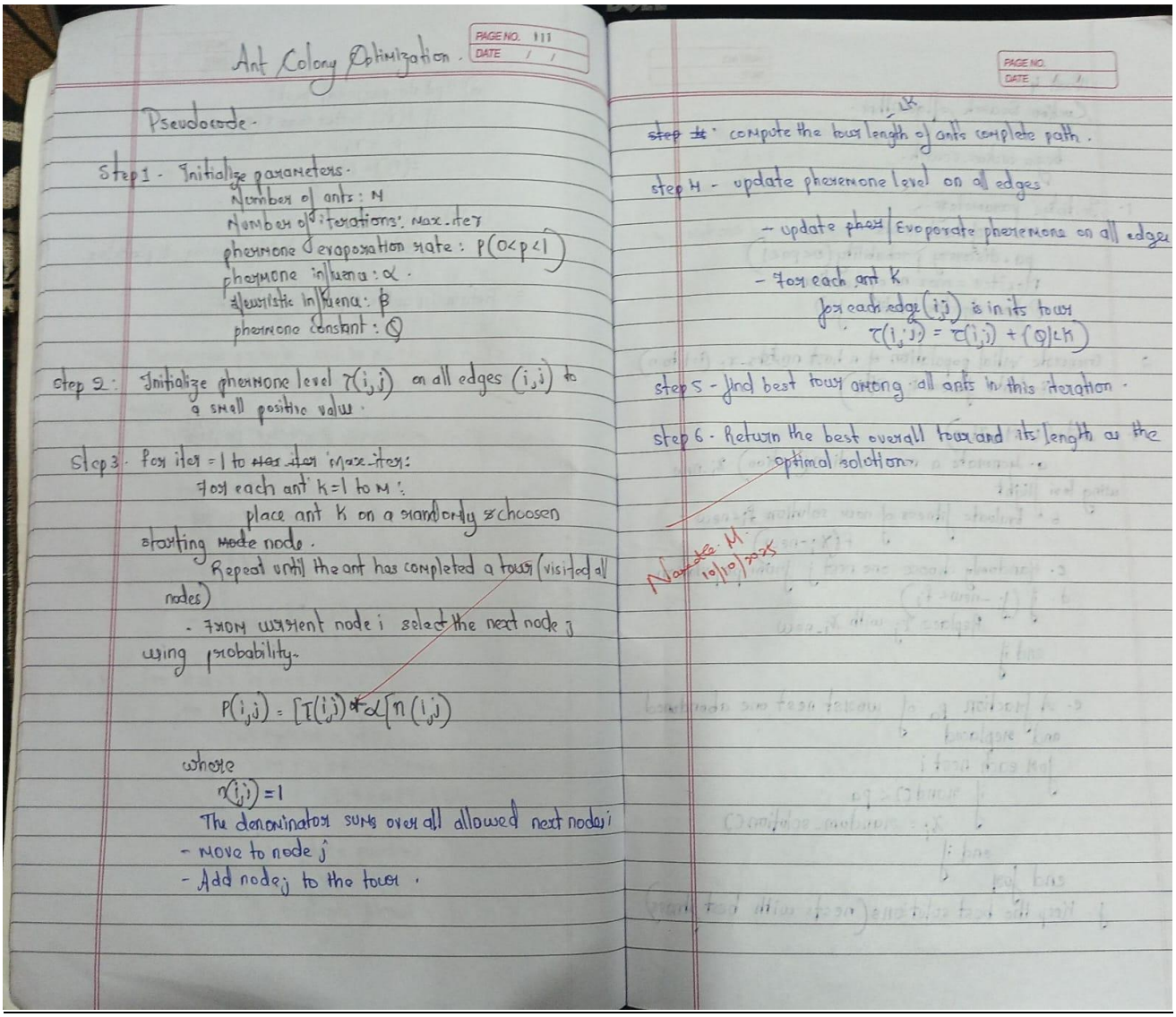
Program 3

Ant Colony Optimization for the Traveling Salesman Problem

Problem Statement

The foraging behavior of ants has inspired the development of optimization algorithms that can solve complex problems such as the Traveling Salesman Problem (TSP). Ant Colony Optimization (ACO) simulates the way ants find the shortest path between food sources and their nest. Implement the ACO algorithm using Python to solve the TSP, where the objective is to find the shortest possible route that visits a list of cities and returns to the origin city.

Algorithm:



Code:

```
import numpy as np
import random
import math

# 1. Define the Problem — Cities and Coordinates
cities = {
   'A': (0, 0),
   'B': (1, 5),
   'C': (5, 2),
   'D': (6, 6),
   'E': (8, 3)
}

city_names = list(cities.keys())
num_cities = len(city_names)

# Distance Matrix
dist_matrix = np.zeros((num_cities, num_cities))
for i in range(num_cities):
   for j in range(num_cities):
      xi, yi = cities[city_names[i]]
      xj, yj = cities[city_names[j]]
      dist_matrix[i][j] = math.sqrt((xi - xj)**2 + (yi - yj)**2)

# 2. Initialize Parameters
num_ants = 10
num_iterations = 50
alpha = 1.0      # Pheromone importance
beta = 5.0       # Heuristic importance (visibility)
rho = 0.5        # Pheromone evaporation rate
Q = 100          # Pheromone deposit factor
initial_pheromone = 1.0

# Initialize pheromone trails
pheromone = np.full((num_cities, num_cities), initial_pheromone)

def route_length(route):
   length = 0
   for i in range(len(route) - 1):
      length += dist_matrix[route[i]][route[i + 1]]
   length += dist_matrix[route[-1]][route[0]]  # Return to start
   return length
best_route = None
best_length = float('inf')

for iteration in range(num_iterations):
   all_routes = []
   all_lengths = []

   for ant in range(num_ants):
      visited = [random.randint(0, num_cities - 1)]
```

```
        while len(visited) < num_cities:
            current = visited[-1]
            probabilities = []
            for j in range(num_cities):
                if j not in visited:
                    tau = pheromone[current][j] ** alpha
                    eta = (1.0 / dist_matrix[current][j]) ** beta
                    probabilities.append(tau * eta)
                else:
                    probabilities.append(0)
            probabilities = np.array(probabilities)
            probabilities /= probabilities.sum()

            next_city = np.random.choice(range(num_cities),
p=probabilities)
            visited.append(next_city)

        length = route_length(visited)
        all_routes.append(visited)
        all_lengths.append(length)
        if length < best_length:
            best_length = length
            best_route = visited

    # 4. Update Pheromones
    pheromone *= (1 - rho)  # Evaporation
    for i, route in enumerate(all_routes):
        for j in range(num_cities - 1):
            a, b = route[j], route[j + 1]
            pheromone[a][b] += Q / all_lengths[i]
            pheromone[b][a] = pheromone[a][b]
        # Add pheromone for return to start
        a, b = route[-1], route[0]
        pheromone[a][b] += Q / all_lengths[i]
        pheromone[b][a] = pheromone[a][b]

    print(f"Iteration {iteration + 1} | Best Length:
{best_length:.4f}")

# 6. Output the Best Solution
best_city_names = [city_names[i] for i in best_route]
print("\n🐜 Best Route Found:")
print(" → ".join(best_city_names + [best_city_names[0]]))
print(f"Total Distance = {best_length:.4f}")
```

Output:

```
Iteration 1 | Best Length: 22.3510
Iteration 2 | Best Length: 22.3510
Iteration 3 | Best Length: 22.3510
Iteration 4 | Best Length: 22.3510
Iteration 5 | Best Length: 22.3510
Iteration 6 | Best Length: 22.3510
Iteration 7 | Best Length: 22.3510
Iteration 8 | Best Length: 22.3510
Iteration 9 | Best Length: 22.3510
Iteration 10 | Best Length: 22.3510
Iteration 11 | Best Length: 22.3510
Iteration 12 | Best Length: 22.3510
Iteration 13 | Best Length: 22.3510
Iteration 14 | Best Length: 22.3510
Iteration 15 | Best Length: 22.3510
Iteration 16 | Best Length: 22.3510
Iteration 17 | Best Length: 22.3510
Iteration 18 | Best Length: 22.3510
Iteration 19 | Best Length: 22.3510
Iteration 20 | Best Length: 22.3510
Iteration 21 | Best Length: 22.3510
Iteration 22 | Best Length: 22.3510
Iteration 23 | Best Length: 22.3510
Iteration 24 | Best Length: 22.3510
Iteration 25 | Best Length: 22.3510
Iteration 26 | Best Length: 22.3510
Iteration 27 | Best Length: 22.3510
Iteration 28 | Best Length: 22.3510
Iteration 29 | Best Length: 22.3510
Iteration 30 | Best Length: 22.3510
Iteration 31 | Best Length: 22.3510
Iteration 32 | Best Length: 22.3510
Iteration 33 | Best Length: 22.3510
Iteration 34 | Best Length: 22.3510
Iteration 35 | Best Length: 22.3510
Iteration 36 | Best Length: 22.3510
Iteration 37 | Best Length: 22.3510
Iteration 38 | Best Length: 22.3510
Iteration 39 | Best Length: 22.3510
Iteration 40 | Best Length: 22.3510
Iteration 41 | Best Length: 22.3510
Iteration 42 | Best Length: 22.3510
Iteration 43 | Best Length: 22.3510
Iteration 44 | Best Length: 22.3510
Iteration 45 | Best Length: 22.3510
Iteration 46 | Best Length: 22.3510
Iteration 47 | Best Length: 22.3510
Iteration 48 | Best Length: 22.3510
Iteration 49 | Best Length: 22.3510
Iteration 50 | Best Length: 22.3510

🐎 Best Route Found:
D → B → A → C → E → D
Total Distance = 22.3510
```

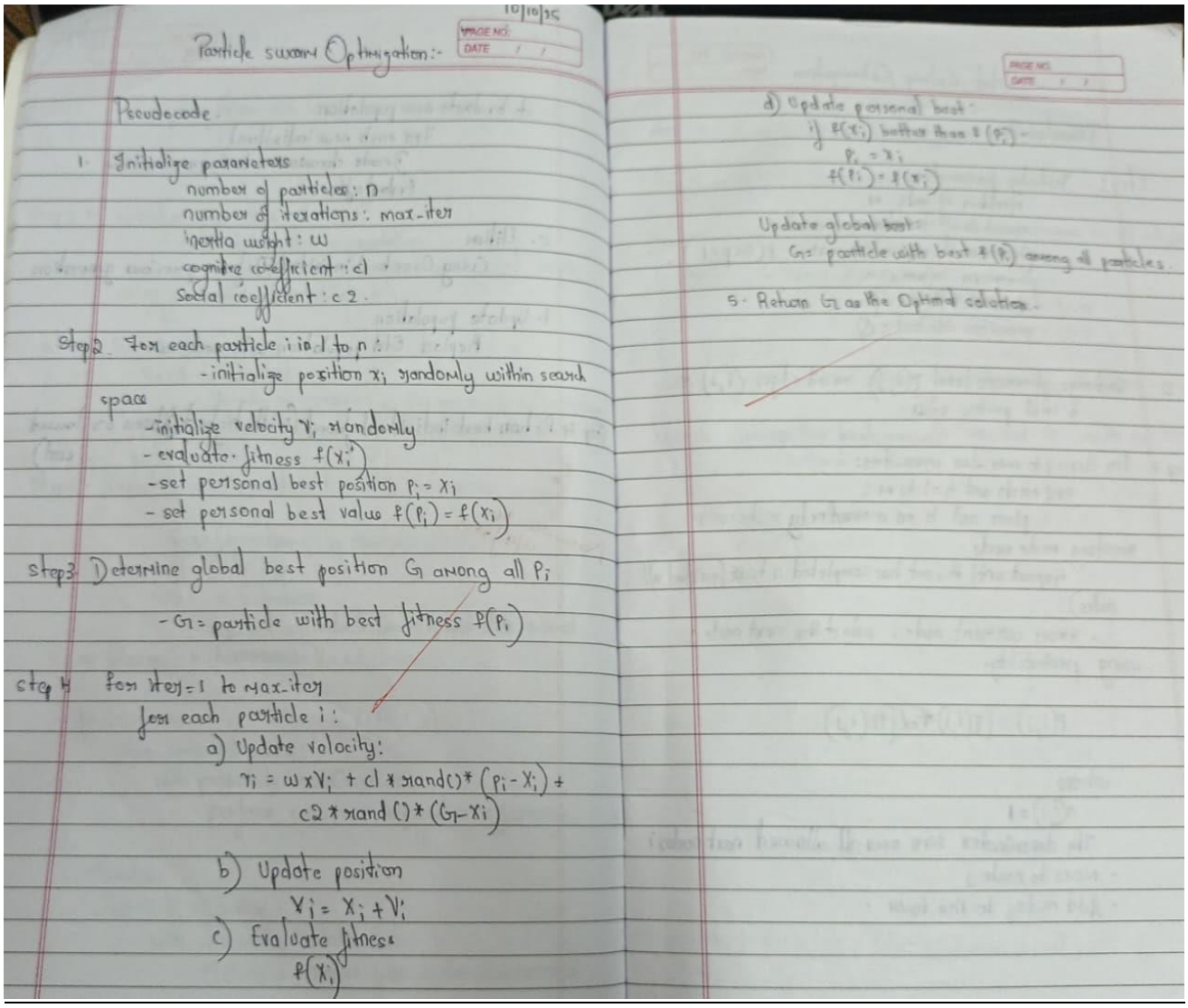
Program 4

Particle Swarm Optimization for Function Optimization

Problem Statement

Particle Swarm Optimization (PSO) is inspired by the social behavior of birds flocking or fish schooling. PSO is used to find optimal solutions by iteratively improving a candidate solution with regard to a given measure of quality. Implement the PSO algorithm using Python to optimize a mathematical function..

Algorithm:



Code:

```
import random
import math
# 1. Define the Problem
def objective_function(x):
"""Example mathematical function to maximize."""
return x * math.sin(10 * math.pi * x) + 1.0
# 2. Initialize Parameters
NUM_PARTICLES = 30
# Number of particles in the swarm
MAX_ITER = 50
W = 0.7
C1 = 1.5
C2 = 1.5
X_MIN, X_MAX = 0, 1
# Number of iterations (generations)
# Inertia weight
# Cognitive (particle's own experience)
# Social (swarm experience)
# Search space boundaries
V_MIN, V_MAX = -0.1, 0.1  # Velocity limits
# 3. Initialize Particles
particles = []  # Each particle: position, velocity, personal_best_pos,
personal_best_val
for _ in range(NUM_PARTICLES):
x = random.uniform(X_MIN, X_MAX)
v = random.uniform(V_MIN, V_MAX)
fitness = objective_function(x)
particles.append({
"position": x,
        "velocity": v,
        "best_position": x,
        "best_value": fitness
    })

# Initialize global best
global_best_position = max(particles, key=lambda p:
p["best_value"])["best_position"]
global_best_value = objective_function(global_best_position)

# 4–6. Main PSO Loop
for iteration in range(MAX_ITER):
    for particle in particles:
        x = particle["position"]
        v = particle["velocity"]
        fitness = objective_function(x)

        # Update personal best
        if fitness > particle["best_value"]:
            particle["best_value"] = fitness
            particle["best_position"] = x
```

```
        # Update global best
        if fitness > global_best_value:
            global_best_value = fitness
            global_best_position = x

    # Update velocities and positions
    for particle in particles:
        r1, r2 = random.random(), random.random()
        cognitive = C1 * r1 * (particle["best_position"] -
particle["position"])
        social = C2 * r2 * (global_best_position -
particle["position"])
        new_velocity = W * particle["velocity"] + cognitive + social
        new_velocity = max(min(new_velocity, V_MAX), V_MIN)  # Clamp
velocity

        new_position = particle["position"] + new_velocity
        new_position = max(min(new_position, X_MAX), X_MIN)  # Keep
within bounds

        particle["velocity"] = new_velocity
        particle["position"] = new_position

    # Print progress
    print(f"Iteration {iteration + 1} | Best Fitness:
{global_best_value:.5f}")

# 7. Output Best Solution
print("\n🐦 Best Solution Found:")
print(f"x = {global_best_position:.5f}")
print(f"Fitness = {global_best_value:.5f}")
```

Output:

```
Iteration 28 | Best Fitness: 1.85060
Iteration 29 | Best Fitness: 1.85060
Iteration 30 | Best Fitness: 1.85060
Iteration 31 | Best Fitness: 1.85060
Iteration 32 | Best Fitness: 1.85060
Iteration 33 | Best Fitness: 1.85060
Iteration 34 | Best Fitness: 1.85060
Iteration 35 | Best Fitness: 1.85060
Iteration 36 | Best Fitness: 1.85060
Iteration 37 | Best Fitness: 1.85060
Iteration 38 | Best Fitness: 1.85060
Iteration 39 | Best Fitness: 1.85060
Iteration 40 | Best Fitness: 1.85060
Iteration 41 | Best Fitness: 1.85060
Iteration 42 | Best Fitness: 1.85060
Iteration 43 | Best Fitness: 1.85060
Iteration 44 | Best Fitness: 1.85060
Iteration 45 | Best Fitness: 1.85060
Iteration 46 | Best Fitness: 1.85060
Iteration 47 | Best Fitness: 1.85060
Iteration 48 | Best Fitness: 1.85060
Iteration 49 | Best Fitness: 1.85060
Iteration 50 | Best Fitness: 1.85060

🐦 Best Solution Found:
x = 0.85119
Fitness = 1.85060
```

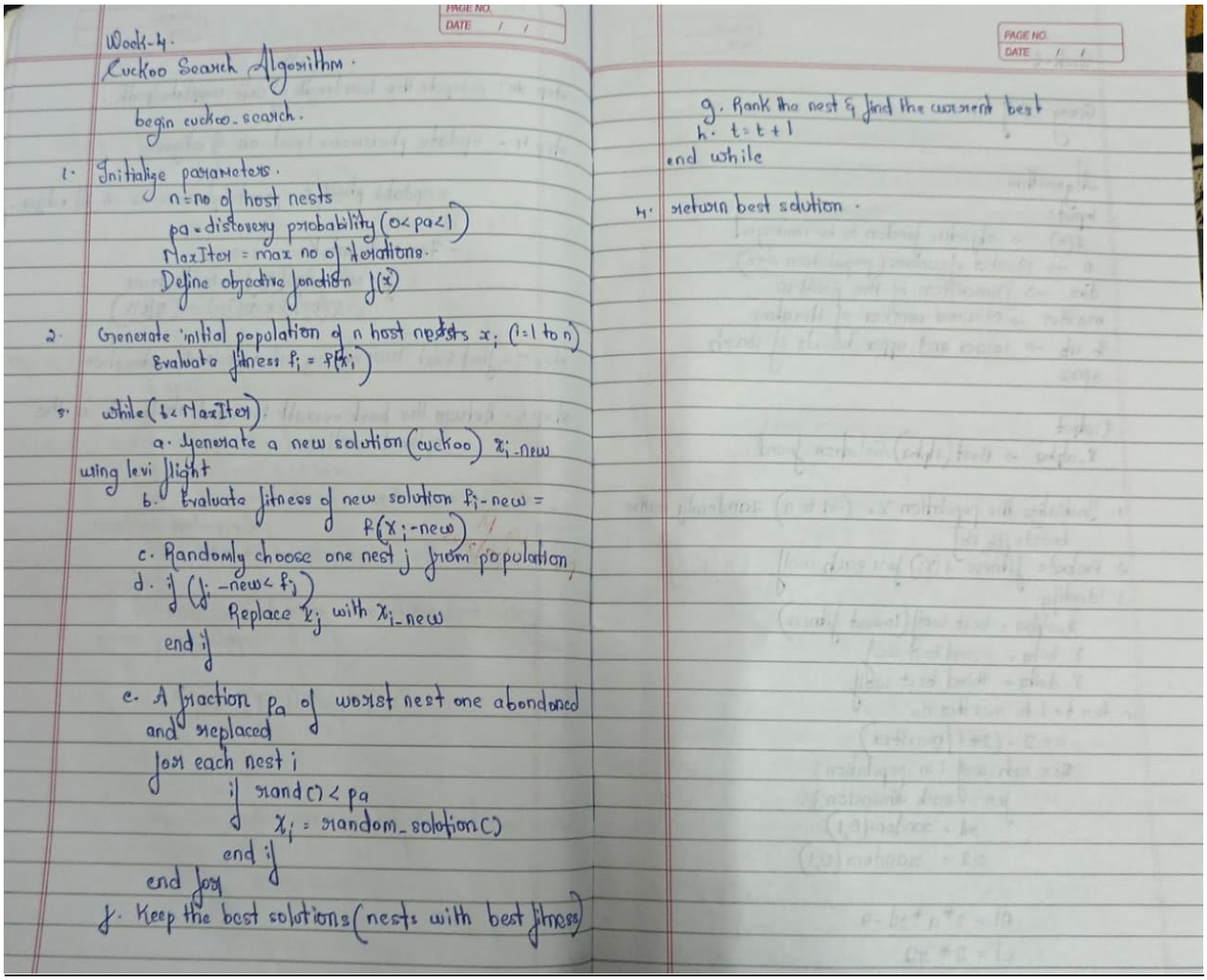
Program 5

Cuckoo Search for the Traveling Salesman Problem

Problem Statement

Cuckoo Search (CS) is a nature-inspired optimization algorithm based on the brood parasitism of some cuckoo species. This behavior involves laying eggs in the nests of other birds, leading to the optimization of survival strategies. CS uses Lévy flights to generate new solutions, promoting global search capabilities and avoiding local minima. The algorithm is widely used for solving continuous optimization problems and has applications in various domains, including engineering design, machine learning, and data mining.

Algorithm:



Week-4.
Cuckoo Search Algorithm.

begin cuckoo_search.

1. Initialize parameters.
   n = no of host nests
   pa = discovery probability (0 < pa < 1)
   MaxIter = max no of iterations.
   Define objective function f(x)
2. Generate initial population of n host nests x_i (i = 1 to n)
   Evaluate fitness $f_i = f(x_i)$
3. while (t < MaxIter)
   a. Generate a new solution (cuckoo) $x_{i\text{-new}}$ using levi flight
   b. Evaluate fitness of new solution $f_{i\text{-new}} = f(x_{i\text{-new}})$
   c. Randomly choose one nest j from population
   d. if ($f_{i\text{-new}} < f_j$)
      Replace x_j with $x_{i\text{-new}}$
      end if
   e. A fraction p_a of worst nest one abondoned and replaced
      for each nest i
         if rand() < pa
            x_i = random_solution()
         end if
      end for
   f. Keep the best solutions (nests with best fitness)
   g. Rank the nest & find the current best
   h. t = t + 1
   end while
4. return best solution.

Code:

```
import numpy as np
import random
import math
# 1. Define the Problem (Objective Function)
def objective_function(x):
"""Example function to maximize."""
return x * math.sin(10 * math.pi * x) + 1.0
# 2. Initialize Parameters
num_nests = 20
# Number of nests (population)
pa = 0.25
abandoned)
max_iter = 50
x_min, x_max = 0, 1
# Discovery probability (fraction of nests
# Number of iterations
# Search space boundaries
# Lévy flight parameters
beta = 1.5  # Lévy exponent
# 3. Initialize Population
nests = np.random.uniform(x_min, x_max, num_nests)
# 4. Evaluate Fitness
fitness = np.array([objective_function(x) for x in nests])
best_nest = nests[np.argmax(fitness)]
best_fitness = max(fitness)

# Helper: Lévy flight step
def levy_flight(Lambda):
    sigma = (math.gamma(1 + Lambda) * math.sin(math.pi * Lambda / 2) /
            (math.gamma((1 + Lambda) / 2) * Lambda * 2 ** ((Lambda -
1) / 2))) ** (1 / Lambda)
    u = np.random.normal(0, sigma)
    v = np.random.normal(0, 1)
    step = u / abs(v) ** (1 / Lambda)
    return step

# 5–7. Main Loop
for iteration in range(max_iter):
    # Generate new solutions via Lévy flights
    new_nests = np.copy(nests)
    for i in range(num_nests):
        step_size = 0.01 * levy_flight(beta)
        new_nests[i] = nests[i] + step_size * np.random.randn()
        new_nests[i] = np.clip(new_nests[i], x_min, x_max)

    # Evaluate fitness of new solutions
    new_fitness = np.array([objective_function(x) for x in new_nests])

    # Replace nests if new solutions are better
    for i in range(num_nests):
        if new_fitness[i] > fitness[i]:
```

```
            fitness[i] = new_fitness[i]
            nests[i] = new_nests[i]

    # Abandon worst nests with probability pa
    abandon = np.random.rand(num_nests) < pa
    nests[abandon] = np.random.uniform(x_min, x_max, np.sum(abandon))
    fitness[abandon] = [objective_function(x) for x in nests[abandon]]

    # Update global best
    current_best = nests[np.argmax(fitness)]
    current_best_fit = max(fitness)
    if current_best_fit > best_fitness:
        best_fitness = current_best_fit
        best_nest = current_best

    print(f"Iteration {iteration + 1} | Best Fitness:
{best_fitness:.5f}")

# 8. Output the Best Solution
print("\n Best Solution Found:")
print(f"x = {best_nest:.5f}")
print(f"Fitness = {best_fitness:.5f}")
```

Output:

```
Iteration 1 | Best Fitness: 1.84947
Iteration 2 | Best Fitness: 1.85056
Iteration 3 | Best Fitness: 1.85056
Iteration 4 | Best Fitness: 1.85059
Iteration 5 | Best Fitness: 1.85059
Iteration 6 | Best Fitness: 1.85059
Iteration 7 | Best Fitness: 1.85059
Iteration 8 | Best Fitness: 1.85059
Iteration 9 | Best Fitness: 1.85059
Iteration 10 | Best Fitness: 1.85059
Iteration 11 | Best Fitness: 1.85059
Iteration 12 | Best Fitness: 1.85059
Iteration 13 | Best Fitness: 1.85059
Iteration 14 | Best Fitness: 1.85059
Iteration 15 | Best Fitness: 1.85059
Iteration 16 | Best Fitness: 1.85059
Iteration 17 | Best Fitness: 1.85059
Iteration 18 | Best Fitness: 1.85059
Iteration 19 | Best Fitness: 1.85059
Iteration 20 | Best Fitness: 1.85059
Iteration 21 | Best Fitness: 1.85059
Iteration 22 | Best Fitness: 1.85059
Iteration 23 | Best Fitness: 1.85059
Iteration 24 | Best Fitness: 1.85059
Iteration 25 | Best Fitness: 1.85059
Iteration 26 | Best Fitness: 1.85059
Iteration 27 | Best Fitness: 1.85059
Iteration 28 | Best Fitness: 1.85059
Iteration 29 | Best Fitness: 1.85059
Iteration 30 | Best Fitness: 1.85059
Iteration 31 | Best Fitness: 1.85059
Iteration 32 | Best Fitness: 1.85059
Iteration 33 | Best Fitness: 1.85059
Iteration 34 | Best Fitness: 1.85059
Iteration 35 | Best Fitness: 1.85059
Iteration 36 | Best Fitness: 1.85059
Iteration 37 | Best Fitness: 1.85059
Iteration 38 | Best Fitness: 1.85059
Iteration 39 | Best Fitness: 1.85059
Iteration 40 | Best Fitness: 1.85059
Iteration 41 | Best Fitness: 1.85059
Iteration 42 | Best Fitness: 1.85059
Iteration 43 | Best Fitness: 1.85059
Iteration 44 | Best Fitness: 1.85059
Iteration 45 | Best Fitness: 1.85059
Iteration 46 | Best Fitness: 1.85059
Iteration 47 | Best Fitness: 1.85059
Iteration 48 | Best Fitness: 1.85059
Iteration 49 | Best Fitness: 1.85059
Iteration 50 | Best Fitness: 1.85059

 Best Solution Found:
x = 0.85113
Fitness = 1.85059
```

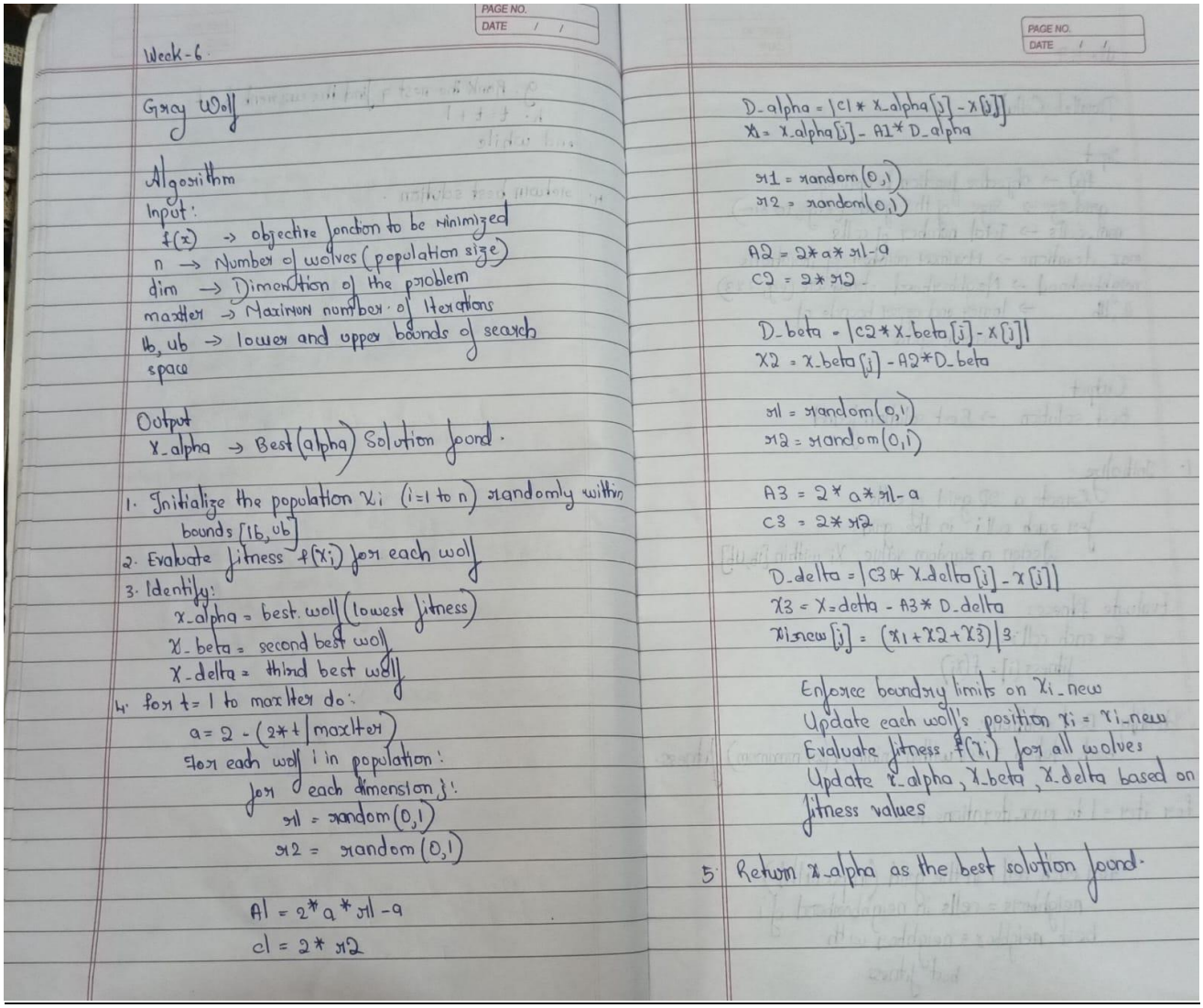
Program 6

Grey Wolf Optimizer for Function Optimization:

Problem Statement

Use Grey Wolf Optimization to find the best solution for a mathematical function by mimicking the hunting behavior of grey wolves.

Algorithm:



Code:

```
import numpy as np

# 1. Define the Objective Function (to be minimized)
def objective_function(x):
    # Example: Sphere function (minimum at x = 0)
    return np.sum(x ** 2)

# 2. Initialize the Grey Wolf Optimizer
```

```
def grey_wolf_optimizer(f, n=20, dim=5, MaxIter=50, lb=-10, ub=10):
    # Initialize the population of wolves randomly within [lb, ub]
    X = np.random.uniform(lb, ub, (n, dim))

    # Evaluate fitness
    fitness = np.array([f(x) for x in X])

    # Identify alpha, beta, delta wolves
    idx = np.argsort(fitness)
    X_alpha, X_beta, X_delta = X[idx[0]], X[idx[1]], X[idx[2]]
    f_alpha, f_beta, f_delta = fitness[idx[0]], fitness[idx[1]], fitness[idx[2]]

    # Main optimization loop
    for t in range(MaxIter):
        a = 2 - 2 * (t / MaxIter)  # linearly decreases from 2 to 0

        for i in range(n):
            for j in range(dim):
                # ---- Alpha influence ----
                r1, r2 = np.random.rand(), np.random.rand()
                A1 = 2 * a * r1 - a
                C1 = 2 * r2
                D_alpha = abs(C1 * X_alpha[j] - X[i][j])
                X1 = X_alpha[j] - A1 * D_alpha

                # ---- Beta influence ----
                r1, r2 = np.random.rand(), np.random.rand()
                A2 = 2 * a * r1 - a
                C2 = 2 * r2
                D_beta = abs(C2 * X_beta[j] - X[i][j])
                X2 = X_beta[j] - A2 * D_beta

                # ---- Delta influence ----
                r1, r2 = np.random.rand(), np.random.rand()
                A3 = 2 * a * r1 - a
                C3 = 2 * r2
                D_delta = abs(C3 * X_delta[j] - X[i][j])
                X3 = X_delta[j] - A3 * D_delta

                # Update wolf position
                X[i][j] = (X1 + X2 + X3) / 3

            # Enforce boundary limits
            X[i] = np.clip(X[i], lb, ub)

        # Recalculate fitness
        fitness = np.array([f(x) for x in X])

        # Update alpha, beta, delta
        idx = np.argsort(fitness)
        if fitness[idx[0]] < f_alpha:
            X_alpha, f_alpha = X[idx[0]], fitness[idx[0]]
        if fitness[idx[1]] < f_beta:
```

```
            X_beta, f_beta = X[idx[1]], fitness[idx[1]]
        if fitness[idx[2]] < f_delta:
            X_delta, f_delta = X[idx[2]], fitness[idx[2]]

        # Display progress
        print(f"Iteration {t+1} | Best Fitness = {f_alpha:.6f}")

    # Return best solution
    return X_alpha, f_alpha


# 3. Run the optimizer
best_position, best_score = grey_wolf_optimizer(objective_function, n=30, dim=5, MaxIter=100, lb=-5, ub=5)

print("\n✓ Best Solution Found:")
print("X_alpha =", best_position)
print("Fitness =", best_score)
```

Output:

```
Iteration 88 | Best Fitness = 0.000000
Iteration 89 | Best Fitness = 0.000000
Iteration 90 | Best Fitness = 0.000000
Iteration 91 | Best Fitness = 0.000000
Iteration 92 | Best Fitness = 0.000000
Iteration 93 | Best Fitness = 0.000000
Iteration 94 | Best Fitness = 0.000000
Iteration 95 | Best Fitness = 0.000000
Iteration 96 | Best Fitness = 0.000000
Iteration 97 | Best Fitness = 0.000000
Iteration 98 | Best Fitness = 0.000000
Iteration 99 | Best Fitness = 0.000000
Iteration 100 | Best Fitness = 0.000000

✅ Best Solution Found:
X_alpha = [ 1.01287500e-12 -1.03916083e-12  1.21093853e-12  1.00657983e-12
 -1.29115530e-12]
Fitness = 6.2524281288562326e-24
```

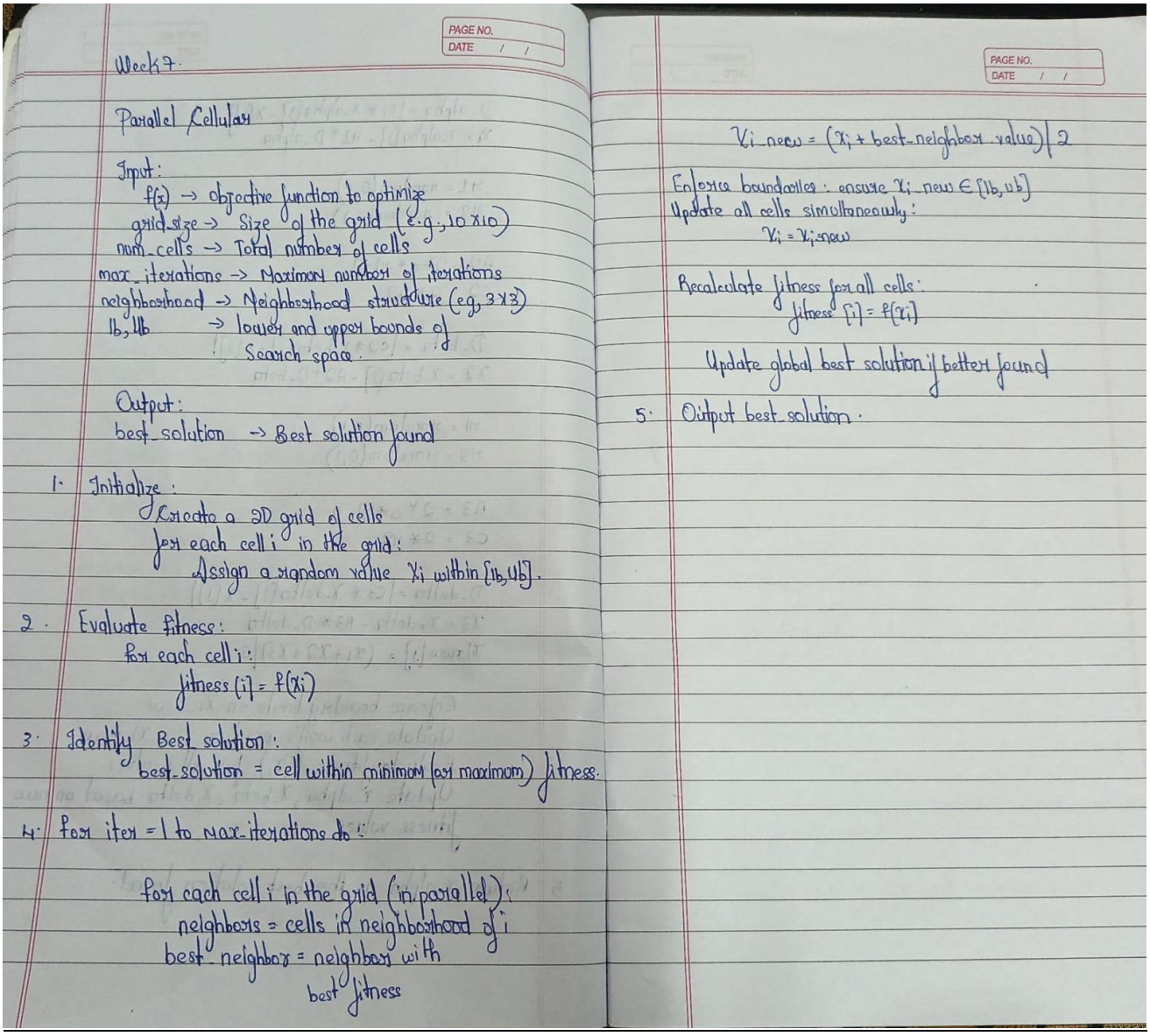
Program 7

Parallel Cellular Algorithm for Function Optimization:

Problem Statement

Use a Parallel Cellular Algorithm to optimize a function efficiently by evolving solutions in a distributed and parallel manner.

Algorithm:



Week 7.

Parallel Cellular

Input:
f(x) → objective function to optimize
grid_size → Size of the grid (e.g., 10 x 10)
num_cells → Total number of cells
max_iterations → Maximum number of iterations
neighborhood → Neighborhood structure (eg, 3x3)
lb, ub → lower and upper bounds of search space.

Output:
best_solution → Best solution found

1. Initialize:
Create a 2D grid of cells
for each cell i in the grid:
Assign a random value x_i within [lb, ub].

2. Evaluate fitness:
for each cell i:
fitness[i] = f(x_i)

3. Identify Best solution:
best_solution = cell within minimum (or maximum) fitness.

4. for iter = 1 to max_iterations do:

for each cell i in the grid (in parallel):
neighbors = cells in neighborhood of i
best_neighbor = neighbor with best fitness

$x_{i\_new} = (x_i + best\_neighbor\_value) / 2$

Enforce boundaries: ensure $x_{i\_new} \in [lb, ub]$
Update all cells simultaneously:
$x_i = x_{i\_new}$

Recalculate fitness for all cells:
fitness[i] = f(x_i)

Update global best solution if better found

5. Output best_solution.

Code:

```
import numpy as np

# 1️⃣ Define the Objective Function (example)
def objective_function(x):
    # Example: Sphere function (minimize)
    return np.sum(x ** 2)

# 2️⃣ Cellular Automata Optimization
def cellular_automata_optimization(f, grid_size=10, max_iterations=50, lb=-10, ub=10, dim=2,
neighborhood=1):
    num_cells = grid_size * grid_size

    # Initialize grid of solutions (each cell = vector of dimension `dim`)
    grid = np.random.uniform(lb, ub, (grid_size, grid_size, dim))

    # Evaluate fitness for each cell
    fitness = np.zeros((grid_size, grid_size))
    for i in range(grid_size):
        for j in range(grid_size):
            fitness[i, j] = f(grid[i, j])

    # Find best initial solution
    best_idx = np.unravel_index(np.argmin(fitness), fitness.shape)
    best_solution = grid[best_idx]
    best_fitness = fitness[best_idx]

    # Main loop
    for iteration in range(max_iterations):
        new_grid = np.copy(grid)

        # For each cell in grid
        for i in range(grid_size):
            for j in range(grid_size):
                # Find neighborhood indices
                i_min = max(0, i - neighborhood)
                i_max = min(grid_size, i + neighborhood + 1)
                j_min = max(0, j - neighborhood)
                j_max = min(grid_size, j + neighborhood + 1)

                # Extract neighbors and their fitness
                neighbors = grid[i_min:i_max, j_min:j_max, :]
                neighbor_fitness = fitness[i_min:i_max, j_min:j_max]

                # Best neighbor
                idx_best = np.unravel_index(np.argmin(neighbor_fitness), neighbor_fitness.shape)
                best_neighbor = neighbors[idx_best]
```

```
            # Update rule: average with best neighbor
            new_grid[i, j] = (grid[i, j] + best_neighbor) / 2.0

            # Boundary enforcement
            new_grid[i, j] = np.clip(new_grid[i, j], lb, ub)

        # Update grid and recompute fitness
        grid = new_grid
        for i in range(grid_size):
            for j in range(grid_size):
                fitness[i, j] = f(grid[i, j])

        # Update global best
        best_idx = np.unravel_index(np.argmin(fitness), fitness.shape)
        if fitness[best_idx] < best_fitness:
            best_solution = grid[best_idx]
            best_fitness = fitness[best_idx]

        print(f"Iteration {iteration+1} | Best Fitness = {best_fitness:.6f}")

    # Return best solution
    return best_solution, best_fitness


# 3️⃣ Run the algorithm
best_sol, best_fit = cellular_automata_optimization(objective_function, grid_size=10,
max_iterations=50, lb=-5, ub=5, dim=2)

print("\n✅ Best Solution Found:")
print("x =", best_sol)
print("Fitness =", best_fit)
```

Output:

```
Iteration 1 | Best Fitness = 0.022848
Iteration 2 | Best Fitness = 0.022848
Iteration 3 | Best Fitness = 0.008335
Iteration 4 | Best Fitness = 0.001871
Iteration 5 | Best Fitness = 0.000076
Iteration 6 | Best Fitness = 0.000076
Iteration 7 | Best Fitness = 0.000076
Iteration 8 | Best Fitness = 0.000004
Iteration 9 | Best Fitness = 0.000004
Iteration 10 | Best Fitness = 0.000000
Iteration 11 | Best Fitness = 0.000000
Iteration 12 | Best Fitness = 0.000000
Iteration 13 | Best Fitness = 0.000000
Iteration 14 | Best Fitness = 0.000000
Iteration 15 | Best Fitness = 0.000000
Iteration 16 | Best Fitness = 0.000000
Iteration 17 | Best Fitness = 0.000000
Iteration 18 | Best Fitness = 0.000000
Iteration 19 | Best Fitness = 0.000000
Iteration 20 | Best Fitness = 0.000000
Iteration 21 | Best Fitness = 0.000000
Iteration 22 | Best Fitness = 0.000000
Iteration 23 | Best Fitness = 0.000000
Iteration 24 | Best Fitness = 0.000000
Iteration 25 | Best Fitness = 0.000000
Iteration 26 | Best Fitness = 0.000000
Iteration 27 | Best Fitness = 0.000000
Iteration 28 | Best Fitness = 0.000000
Iteration 29 | Best Fitness = 0.000000
Iteration 30 | Best Fitness = 0.000000
Iteration 31 | Best Fitness = 0.000000
Iteration 32 | Best Fitness = 0.000000
Iteration 33 | Best Fitness = 0.000000
Iteration 34 | Best Fitness = 0.000000
Iteration 35 | Best Fitness = 0.000000
Iteration 36 | Best Fitness = 0.000000
Iteration 37 | Best Fitness = 0.000000
Iteration 38 | Best Fitness = 0.000000
Iteration 39 | Best Fitness = 0.000000
Iteration 40 | Best Fitness = 0.000000
Iteration 41 | Best Fitness = 0.000000
Iteration 42 | Best Fitness = 0.000000
Iteration 43 | Best Fitness = 0.000000
Iteration 44 | Best Fitness = 0.000000
Iteration 45 | Best Fitness = 0.000000
Iteration 46 | Best Fitness = 0.000000
Iteration 47 | Best Fitness = 0.000000
Iteration 48 | Best Fitness = 0.000000
Iteration 49 | Best Fitness = 0.000000
Iteration 50 | Best Fitness = 0.000000

✅ Best Solution Found:
x = [ 3.4797040e-12 -1.8043457e-12]
Fitness = 1.5364003308026177e-23
```