VISVESVARAYA TECHNOLOGICAL UNIVERSITY

"JnanaSangama", Belgaum - 590014, Karnataka.

LAB RECORD

Bio Inspired Systems (23CS5BSBIS)

Submitted by

Ruqaiyya Mahreen (1BM23CS351)

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 **Ruqaiyya Mahreen (1BM23CS351),** who is a 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 -Prameetha Pai Assistant Professor Department of CSE, BMSCE	Dr. Kavitha Sooda Professor & HOD Department of CSE, BMSCE

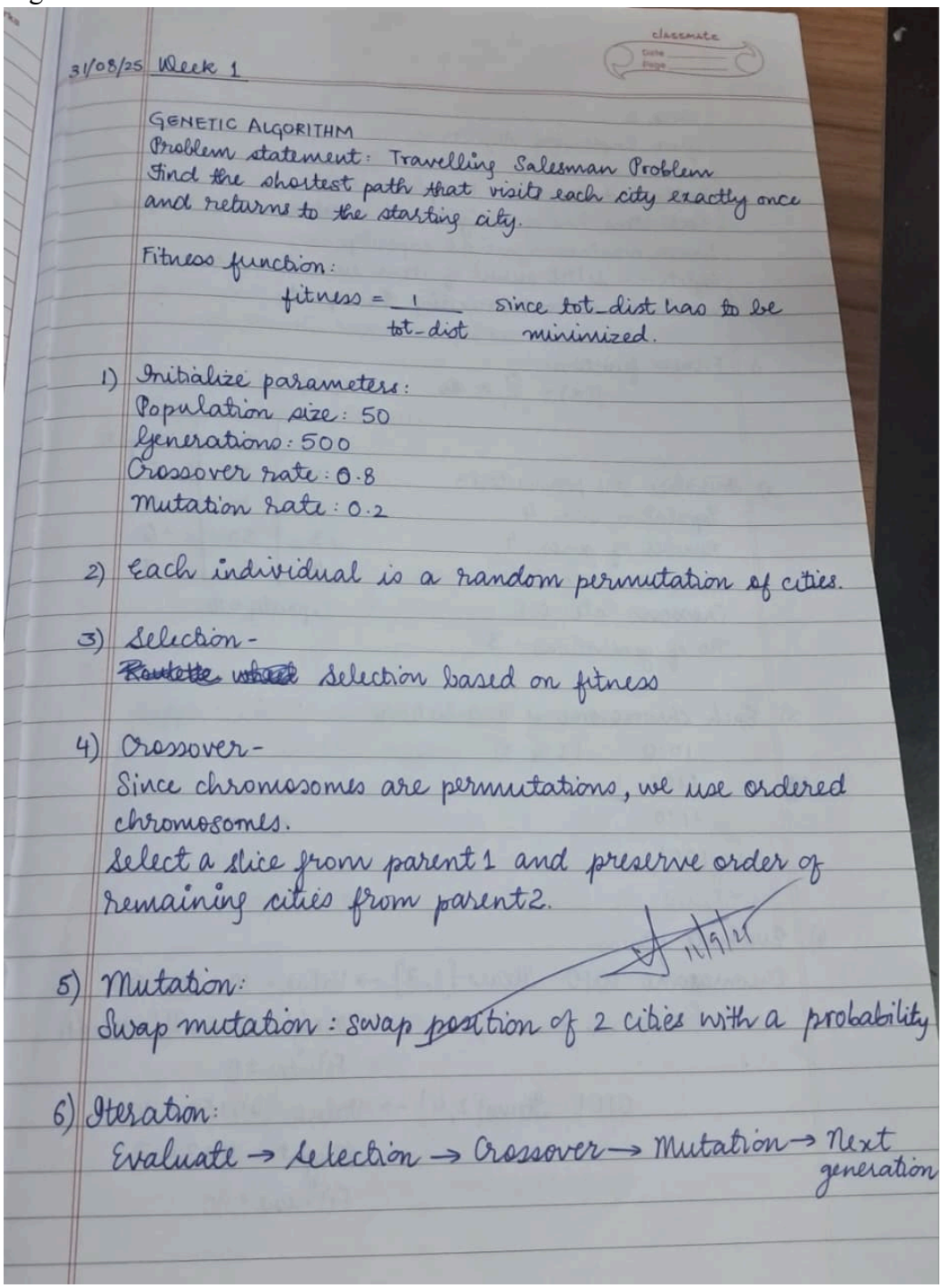
Index

Github Link: https://github.com/Ruqaiyya1BM23CS351/BISLab

Program 1

Implement a Genetic Algorithm to find the shortest possible route that visits a set of cities exactly once and returns to the starting city, optimizing the path using selection, crossover, and mutation operations.

Algorithm:



31/08/25 Week 1

GENETIC ALGORITHM

Problem statement: Travelling Salesman Problem
Find the shortest path that visits each city exactly once and returns to the starting city.

Fitness function:

fitness = $\frac{1}{\text{tot\_dist}}$ since tot_dist has to be minimized.

1) Initialize parameters:
Population size: 50
Generations: 500
Crossover rate: 0.8
Mutation rate: 0.2

2) Each individual is a random permutation of cities.

3) Selection -
~~Roulette wheel~~ Selection based on fitness

4) Crossover -
Since chromosomes are permutations, we use ordered chromosomes.
Select a slice from parent 1 and preserve order of remaining cities from parent 2.

11/9/25

5) Mutation:
Swap mutation : swap position of 2 cities with a probability

6) Iteration:
Evaluate → Selection → Crossover → Mutation → Next generation

Code:

```
import java.util.*;
```

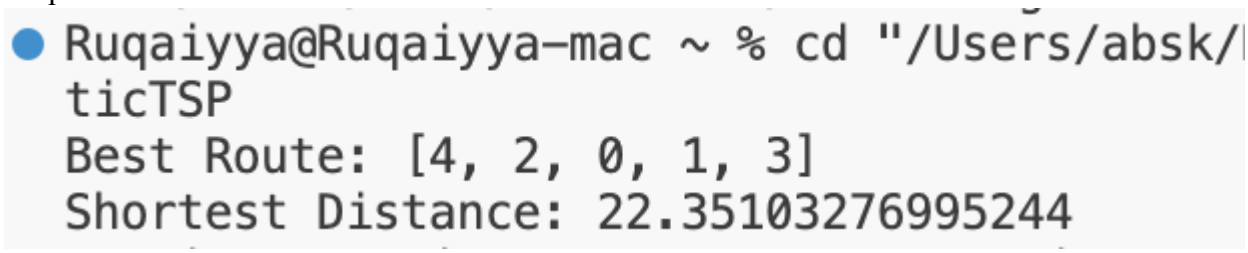
```
public class GeneticTSP {

    // Parameters
    static final int POP_SIZE = 50;
    static final int GENS = 500;
    static final double CROSS_RATE = 0.8;
    static final double MUT_RATE = 0.2;

    static final int NUM_CITIES = 5;
    static final double[][] cities = {
        {0, 0}, {1, 5}, {5, 2}, {6, 6}, {8, 3}
    };

    static Random rand = new Random();

    // Individual representation
    static class Individual {
        int[] route;
        double fitness;

        Individual(int[] route) {
            this.route = route.clone();
            this.fitness = evaluate(route);
        }
    }

    // Euclidean distance
    static double distance(double[] c1, double[] c2) {
        return Math.sqrt(Math.pow(c1[0]-c2[0], 2) + Math.pow(c1[1]-c2[1], 2));
    }

    // Total distance of a route
    static double totalDistance(int[] route) {
        double dist = 0;
        for(int i = 0; i < NUM_CITIES; i++) {
            dist += distance(cities[route[i]], cities[route[(i+1)%NUM_CITIES]]);
        }
        return dist;
    }

    // Fitness function
    static double evaluate(int[] route) {
        return 1.0 / totalDistance(route);
```

```
}

// Initialize population
static List<Individual> initPopulation() {
    List<Individual> population = new ArrayList<>();
    int[] base = new int[NUM_CITIES];
    for(int i=0;i<NUM_CITIES;i++) base[i]=i;
    for(int i=0;i<POP_SIZE;i++) {
        int[] route = base.clone();
        shuffle(route);
        population.add(new Individual(route));
    }
    return population;
}

// Shuffle array
static void shuffle(int[] array) {
    for(int i=array.length-1;i>0;i--) {
        int j = rand.nextInt(i+1);
        int temp = array[i];
        array[i] = array[j];
        array[j] = temp;
    }
}

// Tournament selection
static Individual select(List<Individual> population) {
    int tournamentSize = 5;
    Individual best = null;
    for(int i=0;i<tournamentSize;i++) {
        Individual ind = population.get(rand.nextInt(POP_SIZE));
        if(best==null || ind.fitness > best.fitness) best = ind;
    }
    return best;
}

// Ordered Crossover (OX)
static Individual[] crossover(Individual p1, Individual p2) {
    if(rand.nextDouble() > CROSS_RATE) {
        return new Individual[]{ new Individual(p1.route), new Individual(p2.route) };
    }
    int start = rand.nextInt(NUM_CITIES);
    int end = rand.nextInt(NUM_CITIES);
    if(start > end) { int tmp=start; start=end; end=tmp; }
```

```
    int[] c1 = new int[NUM_CITIES];
    Arrays.fill(c1, -1);
    int[] c2 = new int[NUM_CITIES];
    Arrays.fill(c2, -1);

    // Copy slice
    for(int i=start;i<end;i++) { c1[i]=p1.route[i]; c2[i]=p2.route[i]; }

    // Fill remaining
    fillRemaining(c1, p2.route, end);
    fillRemaining(c2, p1.route, end);

    return new Individual[]{ new Individual(c1), new Individual(c2) };
}

static void fillRemaining(int[] child, int[] parent, int start) {
    int idx = start;
    for(int i=0;i<NUM_CITIES;i++) {
        int city = parent[i];
        if(!contains(child, city)) {
            while(child[idx%NUM_CITIES]!=-1) idx++;
            child[idx%NUM_CITIES]=city;
        }
    }
}

static boolean contains(int[] arr, int val) {
    for(int v: arr) if(v==val) return true;
    return false;
}

// Swap Mutation
static void mutate(Individual ind) {
    for(int i=0;i<NUM_CITIES;i++) {
        if(rand.nextDouble()<MUT_RATE) {
            int j = rand.nextInt(NUM_CITIES);
            int temp = ind.route[i];
            ind.route[i] = ind.route[j];
            ind.route[j] = temp;
        }
    }
    ind.fitness = evaluate(ind.route);
}
```

```
    public static void main(String[] args) {
        List<Individual> population = initPopulation();
        Individual best = null;

        for(int gen=0;gen<GENS;gen++) {
            List<Individual> newPop = new ArrayList<>();

            // Track best
            for(Individual ind: population) {
                if(best==null || ind.fitness>best.fitness) best=ind;
            }

            // Generate new population
            while(newPop.size()<POP_SIZE) {
                Individual p1 = select(population);
                Individual p2 = select(population);
                Individual[] offspring = crossover(p1, p2);
                mutate(offspring[0]);
                mutate(offspring[1]);
                newPop.add(offspring[0]);
                if(newPop.size()<POP_SIZE) newPop.add(offspring[1]);
            }
            population = newPop;
        }

        // Output best
        System.out.println("Best Route: " + Arrays.toString(best.route));
        System.out.println("Shortest Distance: " + totalDistance(best.route));
    }
}
```

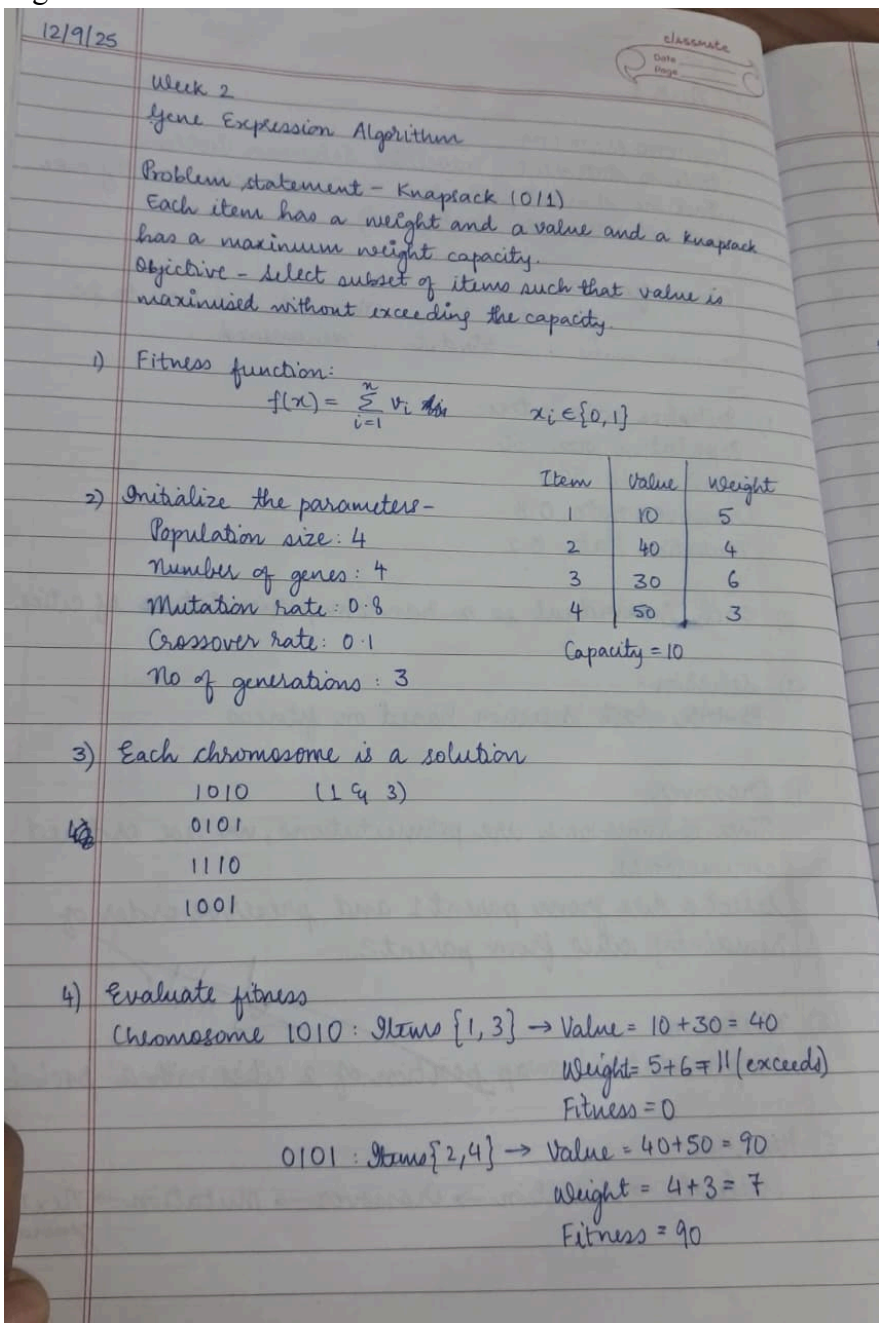
Output:



Program 2

Implement a Genetic Algorithm to solve the 0/1 Knapsack Problem — selecting a combination of items to maximize total value without exceeding the knapsack's weight capacity, using operations like selection, crossover, and mutation.

Algorithm:



12/9/25

Week 2

Gene Expression Algorithm

Problem statement - Knapsack (0/1)
Each item has a weight and a value and a knapsack has a maximum weight capacity.
Objective - Select subset of items such that value is maximised without exceeding the capacity.

1) Fitness function:

$$f(x) = \sum_{i=1}^{n} v_i \quad x_i \in \{0,1\}$$

2) Initialize the parameters -
Population size: 4
Number of genes: 4
Mutation rate: 0.8
Crossover rate: 0.1
No of generations: 3

Item	Value	Weight
1	10	5
2	40	4
3	30	6
4	50	3

Capacity = 10

3) Each chromosome is a solution
1010 (1 & 3)
0101
1110
1001

4) Evaluate fitness
Chromosome 1010 : Items {1, 3} → Value = 10+30 = 40
Weight = 5+6 = 11 (exceeds)
Fitness = 0

0101 : Items {2,4} → Value = 40+50 = 90
Weight = 4+3 = 7
Fitness = 90

1110 : Value = 10+40+30 = 80
Weight = 5+4+6 = 15
Fitness = 80

1001 : Value = 60 Weight = 8 Fitness = 60

Best in generation 0 = 0101 (fitness 90)

5) Selection
Choose parents based on fitness (0101 and 1001)

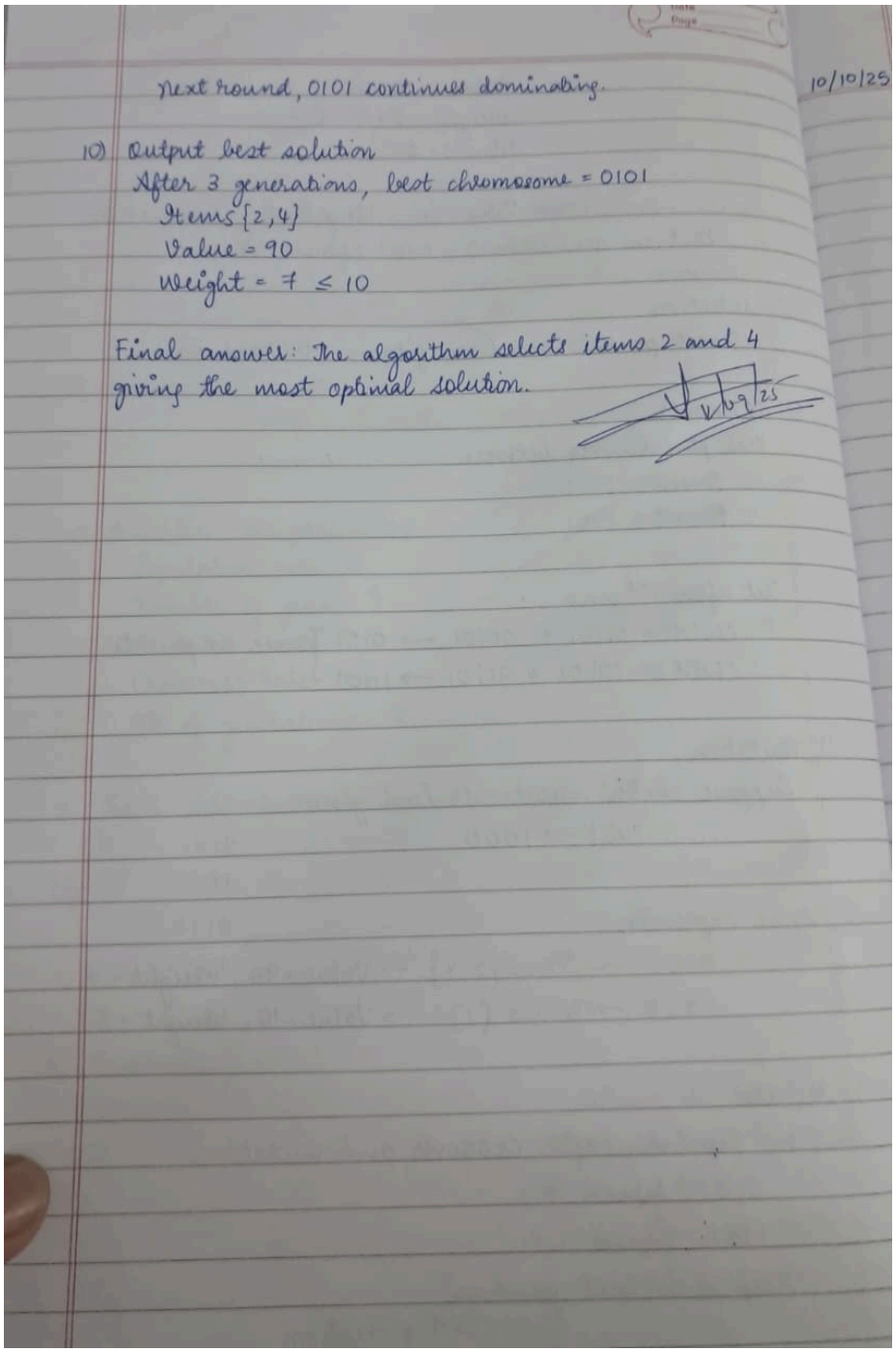
6) Crossover
One pt crossover between :
Parent 1 : 0101
Parent 2 : 1001

Cut after 2nd gene
Child 1 = 01|01 + 00|01 → 0101 } same as parents
Child 2 = 10|01 + 01|01 → 1001 }

7) Mutation
Suppose child2 mutates at last gene
1001 → 1000

8) Gene expression
0101 → Items {2,4} → Value = 90, Weight = 7
1000 → Items {1} → Value = 10, Weight = 5

9) Iterate
New population (after crossover and mutation) :
0101 (fitness 90)
1000 (fitness 10)
Keep elites (best ~~selection~~) from last generation



Code:

```
import random

# ------------------------------
# 1. Define the Problem
# ------------------------------
# Example items
values = [10, 40, 30, 50]   # values of the 4 items
weights = [5, 4, 6, 3]      # weights of the 4 items
capacity = 10               # maximum knapsack weight
num_items = len(values)
```

```
# ------------------------------
# 2. Initialize Parameters
# ------------------------------
POP_SIZE = 10        # population size
GENERATIONS = 30     # number of generations
CROSSOVER_RATE = 0.8
MUTATION_RATE = 0.1

# ------------------------------
# 3. Initialize Population
# ------------------------------
def create_chromosome():
    """Generate a random binary chromosome."""
    return [random.randint(0, 1) for _ in range(num_items)]

def initialize_population():
    """Create an initial population."""
    return [create_chromosome() for _ in range(POP_SIZE)]

# ------------------------------
# 4. Evaluate Fitness
# ------------------------------
def fitness(chromosome):
    total_value = 0
    total_weight = 0
    for gene, v, w in zip(chromosome, values, weights):
        if gene == 1:
            total_value += v
            total_weight += w
    if total_weight > capacity:
        return 0  # penalty for exceeding capacity
    return total_value

# ------------------------------
# 5. Selection
# ------------------------------
def selection(population):
    """Tournament selection."""
    tournament_size = 3
    selected = random.sample(population, tournament_size)
    selected = sorted(selected, key=fitness, reverse=True)
    return selected[0]

# ------------------------------
# 6. Crossover
# ------------------------------
def crossover(parent1, parent2):
    if random.random() < CROSSOVER_RATE:
        point = random.randint(1, num_items - 1)
        child1 = parent1[:point] + parent2[point:]
        child2 = parent2[:point] + parent1[point:]
        return child1, child2
    return parent1[:], parent2[:]

# ------------------------------
# 7. Mutation
```

```
# ----------------------------
def mutate(chromosome):
    for i in range(len(chromosome)):
        if random.random() < MUTATION_RATE:
            chromosome[i] = 1 - chromosome[i]  # flip bit
    return chromosome

# ----------------------------
# 8. Gene Expression (Phenotype Mapping)
# ----------------------------
def decode(chromosome):
    """Translate chromosome into selected items, total value, and total weight."""
    chosen_items = []
    total_value = 0
    total_weight = 0
    for i, gene in enumerate(chromosome):
        if gene == 1:
            chosen_items.append(i + 1)
            total_value += values[i]
            total_weight += weights[i]
    return chosen_items, total_value, total_weight

# ----------------------------
# 9. Iterate Through Generations
# ----------------------------
def gene_expression_algorithm():
    population = initialize_population()
    best_solution = None
    best_fitness = 0

    for generation in range(GENERATIONS):
        new_population = []

        # Evaluate all chromosomes
        population = sorted(population, key=fitness, reverse=True)
        if fitness(population[0]) > best_fitness:
            best_solution = population[0]
            best_fitness = fitness(population[0])

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

        # Elitism: carry the best two
        new_population.extend(population[:2])

        # Create new population
        while len(new_population) < POP_SIZE:
            parent1 = selection(population)
            parent2 = selection(population)
            child1, child2 = crossover(parent1, parent2)
            new_population.append(mutate(child1))
            if len(new_population) < POP_SIZE:
                new_population.append(mutate(child2))

        population = new_population

    return best_solution, best_fitness
```

```
# ------------------------------
# 10. Output Best Solution
# ------------------------------
best_chromosome, best_value = gene_expression_algorithm()
items, total_val, total_wt = decode(best_chromosome)

print("\nBest Chromosome:", best_chromosome)
print("Selected Items:", items)
print("Total Value:", total_val)
print("Total Weight:", total_wt)
```

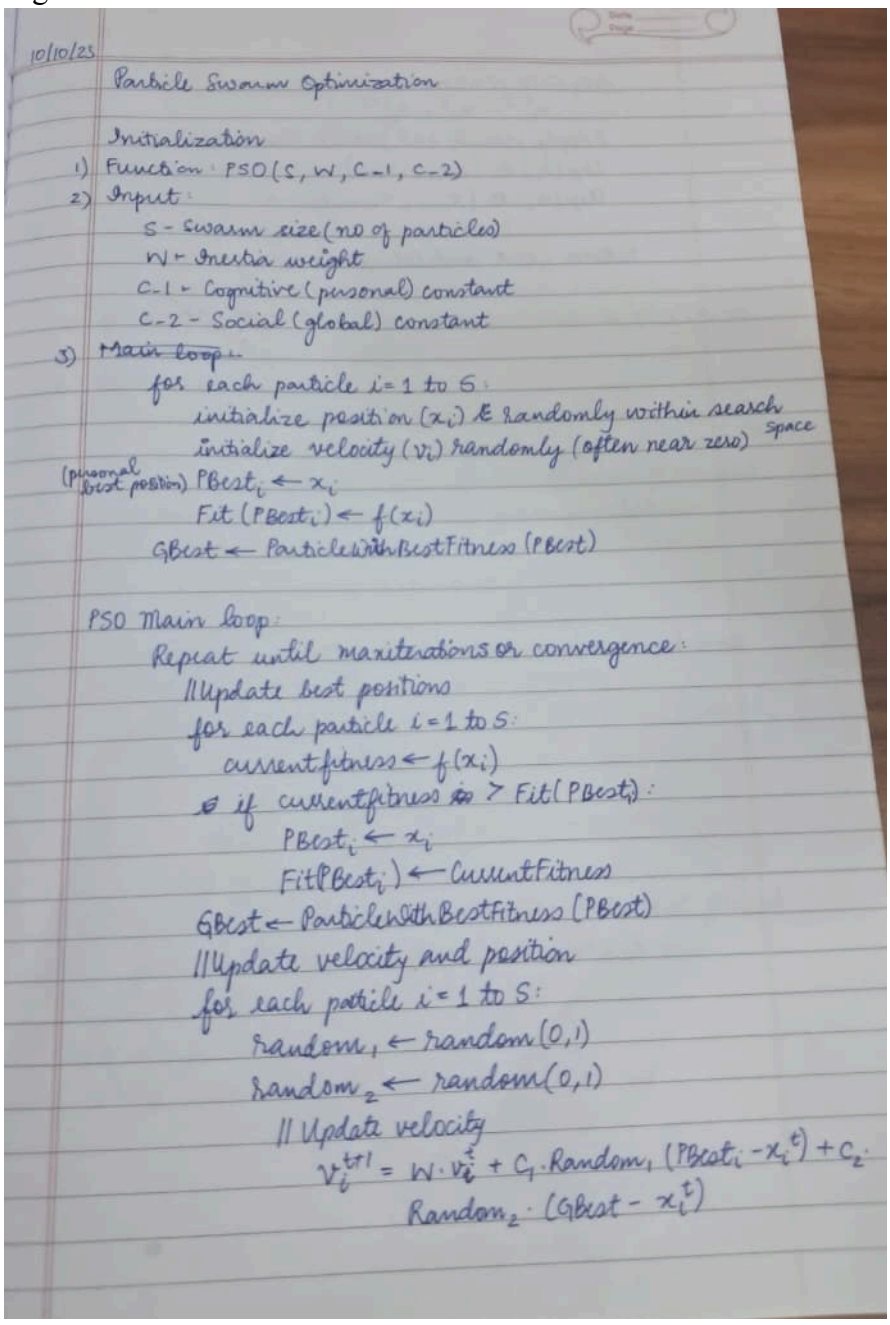
Output:

```
Generation 1: Best Fitness = 70
Generation 2: Best Fitness = 90
Generation 3: Best Fitness = 90
Generation 4: Best Fitness = 90
Generation 5: Best Fitness = 90
Generation 6: Best Fitness = 90
Generation 7: Best Fitness = 90
Generation 8: Best Fitness = 90
Generation 9: Best Fitness = 90
Generation 10: Best Fitness = 90
Generation 11: Best Fitness = 90
Generation 12: Best Fitness = 90
Generation 13: Best Fitness = 90
Generation 14: Best Fitness = 90
Generation 15: Best Fitness = 90
Generation 16: Best Fitness = 90
Generation 17: Best Fitness = 90
Generation 18: Best Fitness = 90
Generation 19: Best Fitness = 90
Generation 20: Best Fitness = 90
Generation 21: Best Fitness = 90
Generation 22: Best Fitness = 90
Generation 23: Best Fitness = 90
Generation 24: Best Fitness = 90
Generation 25: Best Fitness = 90
Generation 26: Best Fitness = 90
Generation 27: Best Fitness = 90
Generation 28: Best Fitness = 90
Generation 29: Best Fitness = 90
Generation 30: Best Fitness = 90

Best Chromosome: [0, 1, 0, 1]
Selected Items: [2, 4]
Total Value: 90
Total Weight: 7
```

Program 3

Implement the Particle Swarm Optimization algorithm to minimize a continuous mathematical function (e.g., the Sphere function) by simulating the collective behavior of particles that explore the search space to find the global optimum.

Algorithm:



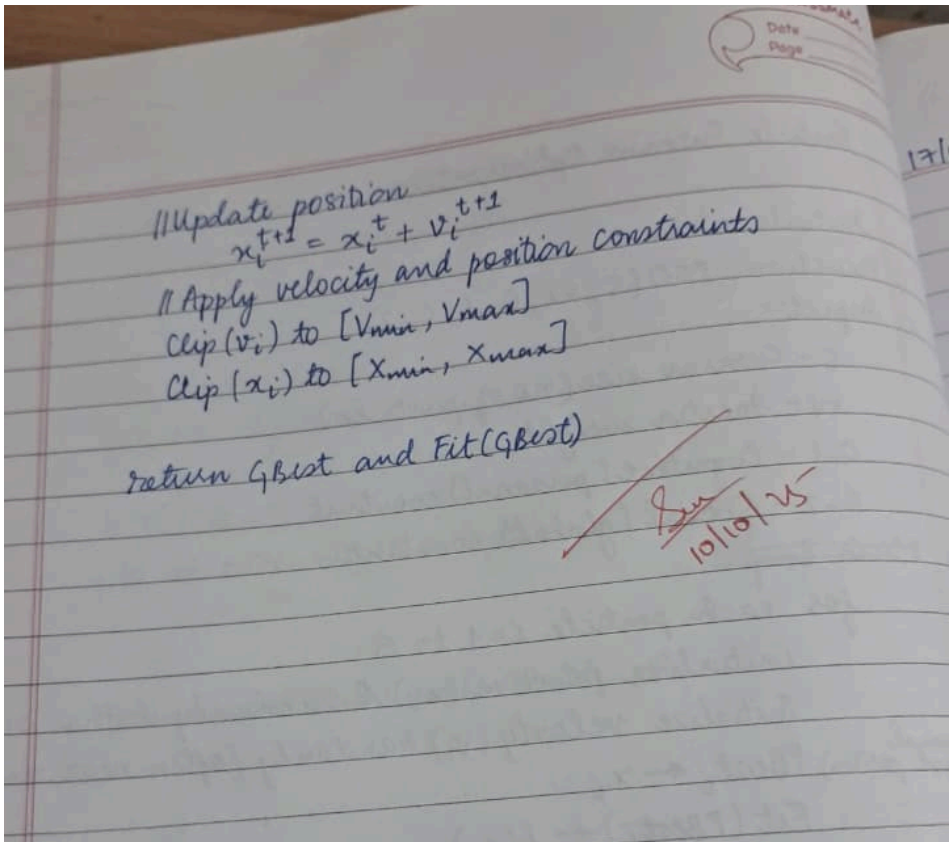
10/10/25

Particle Swarm Optimization

Initialization

1) Function: PSO(S, W, C-1, C-2)
2) Input:
   S - Swarm size (no of particles)
   W - Inertia weight
   C-1 - Cognitive (personal) constant
   C-2 - Social (global) constant
3) Main loop:
   for each particle i = 1 to S:
      initialize position (x_i) & randomly within search space
      initialize velocity (v_i) randomly (often near zero)
   (personal best position) $PBest_i \leftarrow x_i$
      $Fit(PBest_i) \leftarrow f(x_i)$
   $GBest \leftarrow$ ParticleWithBestFitness(PBest)

PSO Main loop:
   Repeat until maxiterations or convergence:
      //Update best positions
      for each particle i = 1 to S:
         currentfitness $\leftarrow f(x_i)$
         if currentfitness > $Fit(PBest_i)$:
            $PBest_i \leftarrow x_i$
            $Fit(PBest_i) \leftarrow$ CurrentFitness
      $GBest \leftarrow$ ParticleWithBestFitness(PBest)
      //Update velocity and position
      for each particle i = 1 to S:
         $random_1 \leftarrow random(0,1)$
         $random_2 \leftarrow random(0,1)$
         // Update velocity
         $v_i^{t+1} = W \cdot v_i^t + C_1 \cdot Random_1 (PBest_i - x_i^t) + C_2 \cdot Random_2 \cdot (GBest - x_i^t)$



Code:

```
import numpy as np
import random
import matplotlib.pyplot as plt
# Set seeds for reproducibility
random.seed(42)
np.random.seed(42)
# --- 1. Objective Function ---
def sphere_function(position):
    """
    The classic Sphere function (f(x) = sum(x^2)), used for minimization.
    The global minimum is f(x)=0 at x=[0, 0, ..., 0].
    """
    # Ensures the input is treated as a NumPy array for vectorized operation
    return np.sum(position**2)
# --- 2. PSO Algorithm Implementation ---
def pso_optimizer(
    objective_func,
    num_particles=30,
    dimensions=2,
    search_range=(-10, 10),
    max_iterations=100,
    w=0.729,    # Inertia Weight (W)
    c1=1.4944, # Cognitive Constant (C1)
    c2=1.4944  # Social Constant (C2)
):
    """
    Particle Swarm Optimization (PSO) algorithm for continuous optimization.
```

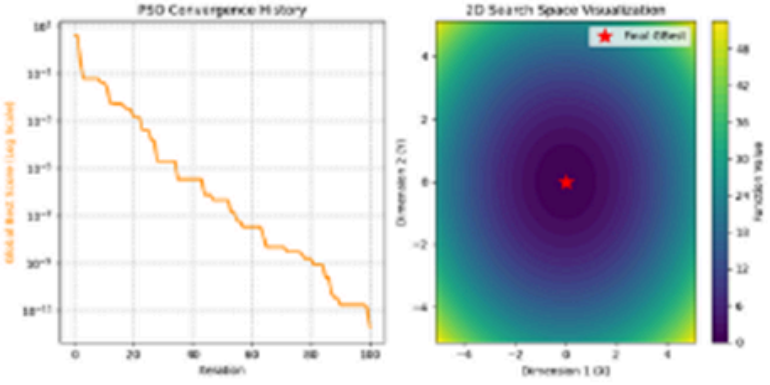
```
Args:
    objective_func (callable): The function to minimize.
    num_particles (int): Number of particles (S).
    dimensions (int): Dimensionality of the search space.
    search_range (tuple): (min, max) bounds for particle positions.
    max_iterations (int): Maximum number of generations.
    w (float): Inertia weight.
    c1 (float): Cognitive constant.
    c2 (float): Social constant.
"""
min_bound, max_bound = search_range

# 1. Initialize particle positions and velocities (x_i and v_i)
# Positions: [num_particles, dimensions]
positions = np.random.uniform(min_bound, max_bound, (num_particles, dimensions))
# Velocities: [num_particles, dimensions]
velocities = np.random.uniform(-1, 1, (num_particles, dimensions))
# 2. Initialize Personal Best (PBest_i)
pbest_positions = positions.copy()
pbest_scores = np.array([objective_func(p) for p in positions])
# 3. Initialize Global Best (GBest)
gbest_index = np.argmin(pbest_scores)
gbest_position = pbest_positions[gbest_index].copy()
gbest_score = pbest_scores[gbest_index]
history = [(gbest_score, gbest_position)]
print(f"Starting PSO for {dimensions} dimensions with {num_particles} particles...")
print(f"Initial GBest Score: {gbest_score:.4f}")
for iteration in range(max_iterations):
    # --- Phase 1: Update PBest Positions ---
    for i in range(num_particles):
        current_score = objective_func(positions[i])
        # Check if current position is better than particle's personal best
        if current_score < pbest_scores[i]:
            pbest_scores[i] = current_score
            pbest_positions[i] = positions[i].copy()

    # --- Update GBest (Global Best) ---
    # Find the overall best position among all PBest's
    current_gbest_index = np.argmin(pbest_scores)
    current_gbest_score = pbest_scores[current_gbest_index]

    # Update GBest only if a better PBest was found
    if current_gbest_score < gbest_score:
        gbest_score = current_gbest_score
        gbest_position = pbest_positions[current_gbest_index].copy()
    # Record history for convergence plot
    history.append((gbest_score, gbest_position.copy()))
    # --- Phase 2: Update Velocity and Position ---

    # Generate two sets of random numbers R1 and R2
    r1 = np.random.rand(num_particles, dimensions) # Random_1
    r2 = np.random.rand(num_particles, dimensions) # Random_2
    # 1. Inertia component: W * v_i^t
    inertia_comp = w * velocities
    # 2. Cognitive component (PBest influence): C1 * r1 * (PBest_i - x_i^t)
    cognitive_comp = c1 * r1 * (pbest_positions - positions)
    # 3. Social component (GBest influence): C2 * r2 * (GBest - x_i^t)
```

```
        # NumPy handles broadcasting of the 1D gbest_position to the 2D positions matrix
        social_comp = c2 * r2 * (gbest_position - positions)
        # Update velocity: v_i^{t+1} = Inertia + Cognitive + Social
        velocities = inertia_comp + cognitive_comp + social_comp
        # Update position: x_i^{t+1} = x_i^t + v_i^{t+1}
        positions = positions + velocities
        # Apply position constraints (clipping to search space)
        positions = np.clip(positions, min_bound, max_bound)
        print(f"Iteration {iteration+1}/{max_iterations}: GBest Score =
{gbest_score:.4e}")
    return gbest_position, gbest_score, history
# --- 3. Example Usage and Visualization ---
def run_pso_example():
    # Set up PSO parameters
    search_range = (-5.12, 5.12) # Common range for Sphere function
    max_iter = 100
    # Run the PSO solver
    best_position, best_score, history = pso_optimizer(
        objective_func=sphere_function,
        num_particles=30,
        dimensions=2, # Using 2D for simple visualization
        search_range=search_range,
        max_iterations=max_iter
    )
    print("\n--- Results ---")
    print(f"Objective Function: Sphere Function")
    print(f"Best Position Found: {best_position}")
    print(f"Minimum Score (Fitness): {best_score:.6e}")
    # --- Visualization ---
    # Extract scores for convergence plot
    scores = [item[0] for item in history]

    plt.figure(figsize=(10, 5))

    # Plot 1: Convergence History
    plt.subplot(1, 2, 1)
    plt.plot(range(len(scores)), scores, color='darkorange', linewidth=2)
    plt.title('PSO Convergence History')
    plt.xlabel('Iteration')
    plt.ylabel('Global Best Score (Log Scale)', color='darkorange')
    plt.yscale('log') # Use log scale for better visualization of minimization
    plt.grid(True, which="both", ls="--")
    # Plot 2: Particle movement (only for 2D problems)
    if history and len(history[0][1]) == 2:
        plt.subplot(1, 2, 2)

        # Create a contour plot of the objective function
        x = np.linspace(search_range[0], search_range[1], 100)
        y = np.linspace(search_range[0], search_range[1], 100)
        X, Y = np.meshgrid(x, y)

        # Calculate Z values for the Sphere function across the grid
        Z = np.array([[sphere_function(np.array([X[i, j], Y[i, j]])) for j in range(100)]
for i in range(100)])

        plt.contourf(X, Y, Z, levels=50, cmap='viridis')
        plt.colorbar(label='Function Value')
```

```
        # Plot the final best position
        plt.plot(best_position[0], best_position[1], 'r*', markersize=15, label='Final
GBest')
        plt.title('2D Search Space Visualization')
        plt.xlabel('Dimension 1 (X)')
        plt.ylabel('Dimension 2 (Y)')
        plt.legend()
    plt.tight_layout()
    plt.show()
# Execute the example
if __name__ == '__main__':
    run_pso_example()
```

Output:

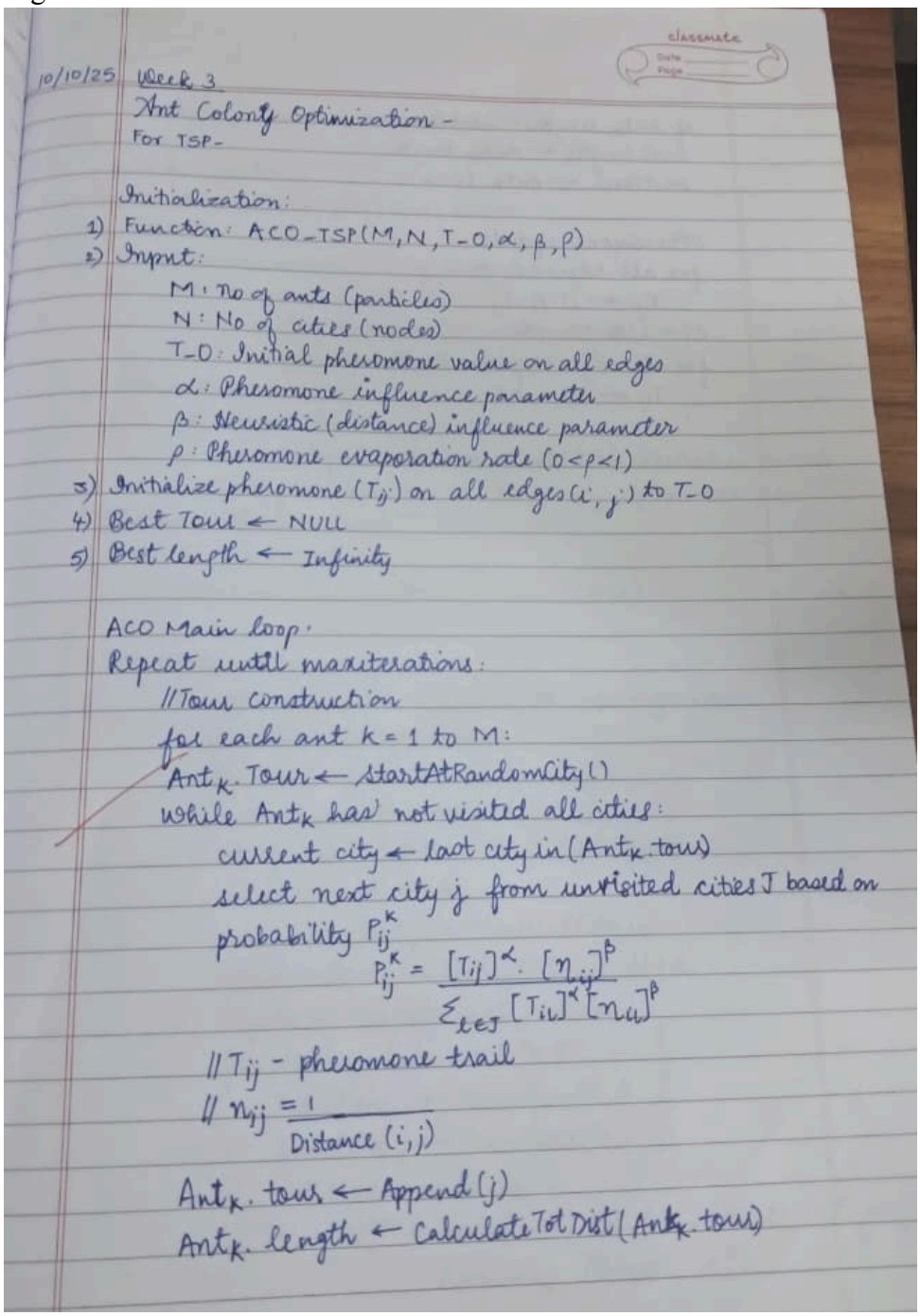
```
Starting PSO for 2 dimensions with 50 particles...
Initial Global Score: 5.7937
Iteration 1/100: Global Score = 5.7937e+00
Iteration 2/100: Global Score = 2.9180e-01
Iteration 3/100: Global Score = 9.7470e-02
Iteration 4/100: Global Score = 9.7470e-02
Iteration 5/100: Global Score = 9.7470e-02
Iteration 6/100: Global Score = 9.7470e-02
Iteration 7/100: Global Score = 9.7470e-02
Iteration 8/100: Global Score = 9.7470e-02
Iteration 9/100: Global Score = 5.8838e-02
Iteration 10/100: Global Score = 5.8838e-02
Iteration 11/100: Global Score = 2.9322e-02
Iteration 12/100: Global Score = 9.7894e-03
Iteration 13/100: Global Score = 9.7894e-03
Iteration 14/100: Global Score = 4.9934e-03
Iteration 15/100: Global Score = 4.9934e-03
Iteration 16/100: Global Score = 4.9934e-03
Iteration 17/100: Global Score = 3.9827e-03
Iteration 18/100: Global Score = 2.9962e-03
Iteration 19/100: Global Score = 2.9962e-03
Iteration 20/100: Global Score = 1.9994e-03
Iteration 21/100: Global Score = 1.9994e-03
Iteration 22/100: Global Score = 1.9994e-03
Iteration 23/100: Global Score = 5.8987e-04
Iteration 24/100: Global Score = 5.8987e-04
Iteration 25/100: Global Score = 5.8987e-04
Iteration 26/100: Global Score = 1.9864e-04
Iteration 27/100: Global Score = 1.9098e-04
Iteration 28/100: Global Score = 1.9113e-05
Iteration 29/100: Global Score = 1.9113e-05
Iteration 30/100: Global Score = 1.9113e-05
Iteration 31/100: Global Score = 1.9113e-05
Iteration 32/100: Global Score = 1.9113e-05
Iteration 33/100: Global Score = 1.9113e-05
Iteration 34/100: Global Score = 1.9113e-05
Iteration 35/100: Global Score = 5.2498e-06
Iteration 36/100: Global Score = 5.2498e-06
Iteration 37/100: Global Score = 5.2498e-06
Iteration 38/100: Global Score = 5.2498e-06
Iteration 39/100: Global Score = 5.2498e-06
Iteration 40/100: Global Score = 5.2498e-06
Iteration 41/100: Global Score = 5.2498e-06
Iteration 42/100: Global Score = 5.2498e-06
Iteration 43/100: Global Score = 5.2498e-06
Iteration 44/100: Global Score = 7.1491e-07
Iteration 45/100: Global Score = 7.1491e-07
Iteration 46/100: Global Score = 7.1491e-07
Iteration 47/100: Global Score = 6.5268e-07
Iteration 48/100: Global Score = 6.5268e-07
Iteration 49/100: Global Score = 6.5268e-07
Iteration 50/100: Global Score = 6.5268e-07
Iteration 51/100: Global Score = 6.5268e-07
Iteration 52/100: Global Score = 6.5268e-07
Iteration 53/100: Global Score = 1.5481e-07
Iteration 54/100: Global Score = 1.5481e-07
Iteration 55/100: Global Score = 9.7958e-08
Iteration 56/100: Global Score = 9.7958e-08
Iteration 57/100: Global Score = 5.1713e-08
Iteration 58/100: Global Score = 5.1713e-08
Iteration 59/100: Global Score = 5.1713e-08
Iteration 60/100: Global Score = 5.1713e-08
Iteration 61/100: Global Score = 5.1713e-08
Iteration 62/100: Global Score = 5.1713e-08
Iteration 63/100: Global Score = 5.1713e-08
Iteration 64/100: Global Score = 1.2872e-08
Iteration 65/100: Global Score = 4.4388e-09
Iteration 66/100: Global Score = 4.4388e-09
Iteration 67/100: Global Score = 4.4388e-09
Iteration 68/100: Global Score = 4.4388e-09
Iteration 69/100: Global Score = 4.4388e-09
Iteration 70/100: Global Score = 4.4388e-09
Iteration 71/100: Global Score = 4.4388e-09
Iteration 72/100: Global Score = 2.7951e-09
Iteration 73/100: Global Score = 2.7951e-09
Iteration 74/100: Global Score = 2.7951e-09
Iteration 75/100: Global Score = 2.7951e-09
Iteration 76/100: Global Score = 2.7951e-09
Iteration 77/100: Global Score = 2.2833e-09
Iteration 78/100: Global Score = 1.6689e-09
Iteration 79/100: Global Score = 1.6689e-09
Iteration 80/100: Global Score = 1.6689e-09
Iteration 81/100: Global Score = 8.8366e-10
Iteration 82/100: Global Score = 8.8366e-10
Iteration 83/100: Global Score = 8.8366e-10
Iteration 84/100: Global Score = 8.8366e-10
Iteration 85/100: Global Score = 3.9605e-10
Iteration 86/100: Global Score = 3.9605e-10
Iteration 87/100: Global Score = 9.8997e-11
Iteration 88/100: Global Score = 5.3221e-11
Iteration 89/100: Global Score = 5.3221e-11
Iteration 90/100: Global Score = 1.4175e-11
Iteration 91/100: Global Score = 1.4175e-11
Iteration 92/100: Global Score = 1.4175e-11
Iteration 93/100: Global Score = 1.4175e-11
Iteration 94/100: Global Score = 1.4175e-11
Iteration 95/100: Global Score = 1.4175e-11
Iteration 96/100: Global Score = 1.4175e-11
Iteration 97/100: Global Score = 1.4175e-11
Iteration 98/100: Global Score = 1.4175e-11
Iteration 99/100: Global Score = 1.9579e-11
Iteration 100/100: Global Score = 1.7771e-12

--- Results ---
Objective Function: Sphere Function
Best Position Found: [9.9915e-07 9.2079e-07]
Minimum Score (Fitness): 1.7771e-12
```



Program 4

Implement the Ant Colony Optimization algorithm to find the shortest possible route that visits all cities exactly once and returns to the starting city (Traveling Salesman Problem).

Algorithm:



10/10/25 Week 3

Ant Colony Optimization -
For TSP-

Initialization:

1) Function: ACO_TSP(M, N, T_0, α, β, ρ)
2) Input:
   - M: No of ants (particles)
   - N: No of cities (nodes)
   - T_0: Initial pheromone value on all edges
   - α: Pheromone influence parameter
   - β: Heuristic (distance) influence parameter
   - ρ: Pheromone evaporation rate ($0 < \rho < 1$)
3) Initialize pheromone (T_{ij}) on all edges (i, j) to T_0
4) Best Tour ← NULL
5) Best length ← Infinity

ACO Main loop:

Repeat until maxiterations:

//Tour construction

for each ant k = 1 to M:

Ant_k.Tour ← StartAtRandomCity()

while Ant_k has not visited all cities:

current city ← last city in (Ant_k.tour)

select next city j from unvisited cities J based on probability P_{ij}^k

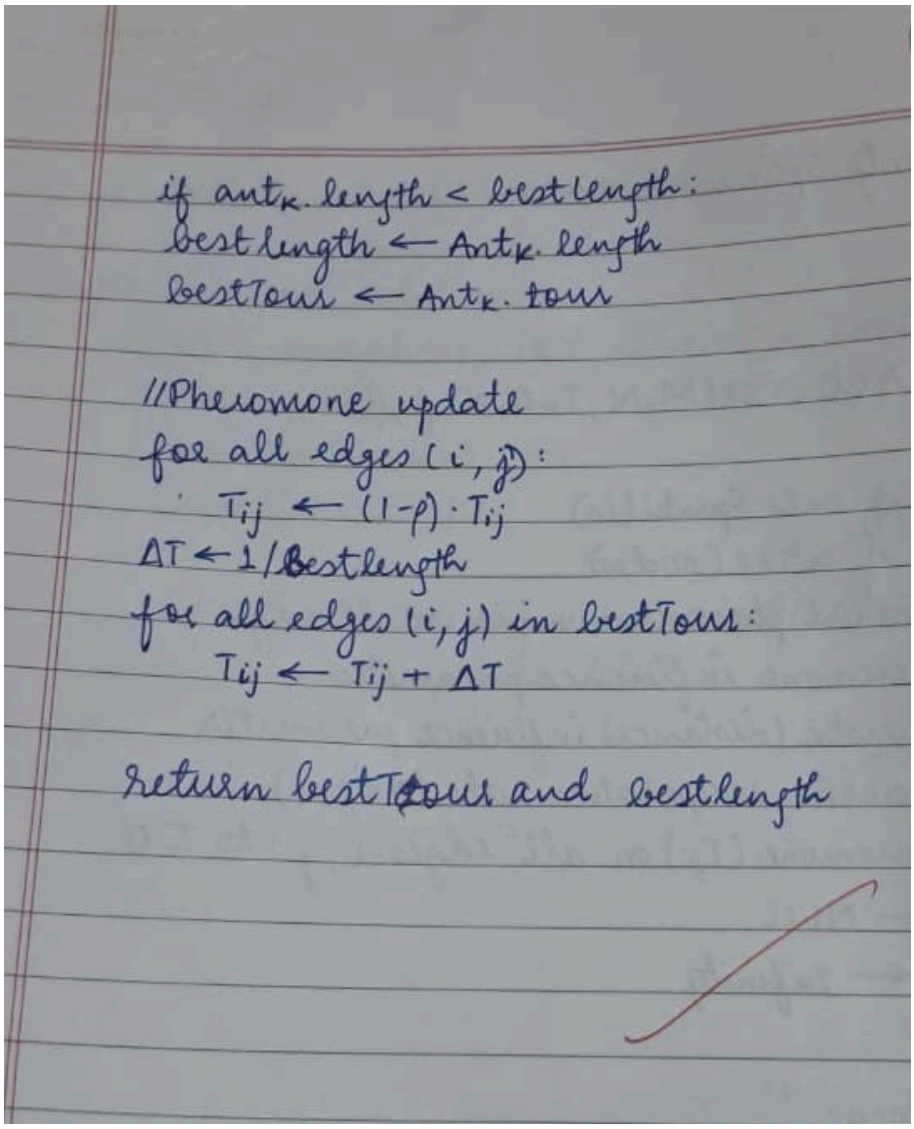
$$P_{ij}^k = \frac{[T_{ij}]^\alpha \cdot [\eta_{ij}]^\beta}{\sum_{l \in J} [T_{il}]^\alpha [\eta_{il}]^\beta}$$

// T_{ij} - pheromone trail

// $\eta_{ij} = \frac{1}{\text{Distance}(i,j)}$

Ant_k.tours ← Append(j)

Ant_k.length ← CalculateTotDist(Ant_k.tours)



Code:

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

# --- 1. Utility Functions ---
def create_distance_matrix(cities):
    """Calculates the Euclidean distance matrix between all pairs of cities."""
    num_cities = len(cities)
    dist_matrix = np.zeros((num_cities, num_cities))
    for i in range(num_cities):
        for j in range(i + 1, num_cities):
            # Calculate Euclidean distance
            distance = np.sqrt((cities[i][0] - cities[j][0])**2 + (cities[i][1] -
cities[j][1])**2)
            dist_matrix[i, j] = dist_matrix[j, i] = distance
    return dist_matrix
```

```
def calculate_tour_length(tour, dist_matrix):
    """Calculates the total length of a given tour (sequence of city indices)."""
    length = 0
    num_cities = len(tour)
    for i in range(num_cities):
        # Add distance from current city to next city in the tour
        city_a = tour[i]
        city_b = tour[(i + 1) % num_cities]  # Wrap around to the start city
        length += dist_matrix[city_a, city_b]
    return length


# --- 2. ACO Algorithm Implementation ---
def aco_tsp_solver(cities, num_ants=10, max_iterations=100, alpha=1.0, beta=5.0, rho=0.5,
initial_pheromone=1.0):
    """
    Ant Colony Optimization (ACO) algorithm for the Traveling Salesman Problem (TSP).
    Args:
        cities (list of tuples): List of (x, y) coordinates for each city.
        num_ants (int): Number of artificial ants (M).
        max_iterations (int): Maximum number of generations to run.
        alpha (float): Influence of the pheromone trail (tau).
        beta (float): Influence of the heuristic information (eta, 1/distance).
        rho (float): Pheromone evaporation rate.
        initial_pheromone (float): Initial pheromone value (tau_0).
    """
    num_cities = len(cities)
    dist_matrix = create_distance_matrix(cities)

    # Heuristic matrix (eta_ij = 1 / distance_ij)
    eta_matrix = 1.0 / (dist_matrix + np.finfo(float).eps)
    np.fill_diagonal(eta_matrix, 0)

    # Initialize pheromone matrix (tau_ij)
    pheromone_matrix = np.full((num_cities, num_cities), initial_pheromone)

    # Initialize best tour found so far
    best_tour = None
    best_length = float('inf')
    history = []

    print(f"Starting ACO for {num_cities} cities with {num_ants} ants...")

    for iteration in range(max_iterations):
        all_tours = []
        all_lengths = []

        # --- Phase 1: Tour Construction ---
        for ant in range(num_ants):
            start_city = random.randint(0, num_cities - 1)
            tour = [start_city]
            visited = {start_city}

            for _ in range(num_cities - 1):
                current_city = tour[-1]
                unvisited_cities = [c for c in range(num_cities) if c not in visited]
```

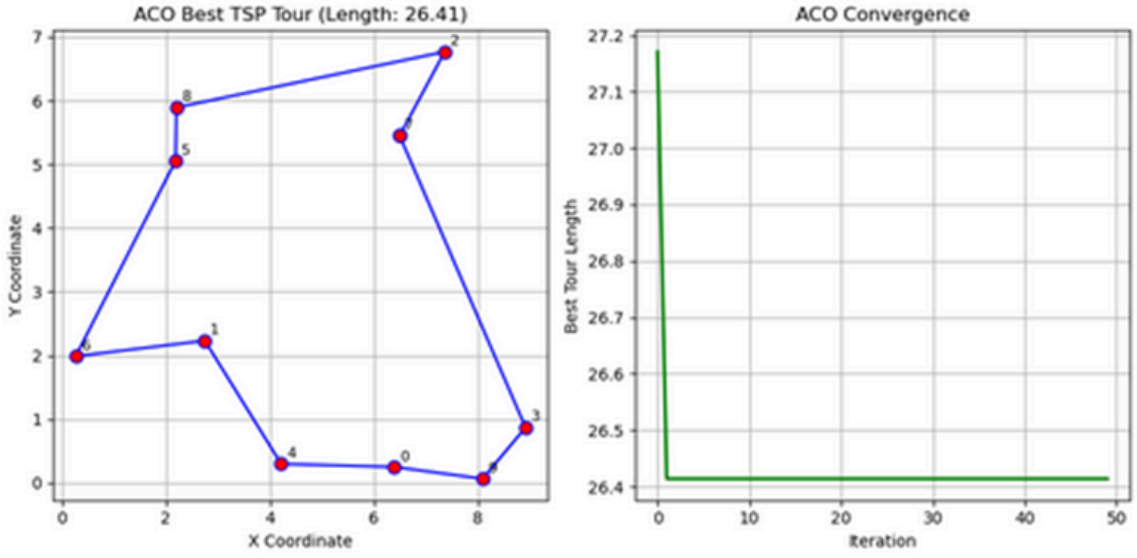
```
                if not unvisited_cities:
                    break  # Should not happen if TSP is solvable

                # Calculate probabilities P_ij
                probabilities = []
                denominator = 0.0

                for next_city in unvisited_cities:
                    tau = pheromone_matrix[current_city, next_city] ** alpha
                    eta = eta_matrix[current_city, next_city] ** beta
                    numerator = tau * eta
                    probabilities.append((next_city, numerator))
                    denominator += numerator

                if denominator == 0:
                    # Fallback to random choice if all probabilities are zero (rare)
                    next_city = random.choice(unvisited_cities)
                else:
                    # Select next city based on roulette wheel selection (weighted
probability)
                    prob_values = [p[1] / denominator for p in probabilities]
                    next_city = random.choices(
                        [p[0] for p in probabilities],
                        weights=prob_values,
                        k=1
                    )[0]

                tour.append(next_city)
                visited.add(next_city)

            tour_length = calculate_tour_length(tour, dist_matrix)
            all_tours.append(tour)
            all_lengths.append(tour_length)

            # Update personal best (used for global best update below)
            if tour_length < best_length:
                best_length = tour_length
                best_tour = tour

        history.append(best_length)

        # --- Phase 2: Pheromone Update ---
        # 1. Evaporation: tau_ij = (1 - rho) * tau_ij
        pheromone_matrix = (1 - rho) * pheromone_matrix

        # 2. Deposition: The best ant deposits pheromone (Ant System variant)
        if best_tour is not None:
            # Pheromone deposit (Delta_tau = 1 / BestLength)
            delta_tau = 1.0 / best_length
            # Deposit pheromone along the best tour
            for i in range(num_cities):
                city_a = best_tour[i]
                city_b = best_tour[(i + 1) % num_cities]
                pheromone_matrix[city_a, city_b] += delta_tau
                pheromone_matrix[city_b, city_a] += delta_tau  # TSP graph is symmetric

        print(f"Iteration {iteration+1}/{max_iterations}: Best Length =
```

```
{best_length:.2f}")

    return best_tour, best_length, history


# --- 3. Example Usage and Visualization ---
def run_aco_example():
    # Define a simple set of 10 cities (x, y coordinates)
    random.seed(42)  # for reproducibility
    np.random.seed(42)
    cities = [(random.uniform(0, 10), random.uniform(0, 10)) for _ in range(10)]

    # Run the ACO solver
    best_tour, best_length, history = aco_tsp_solver(
        cities,
        num_ants=20,
        max_iterations=50,
        alpha=1.0,
        beta=5.0,
        rho=0.1
    )

    print("\n--- Results ---")
    print(f"Cities: {cities}")
    print(f"Best Tour (City Indices): {best_tour}")
    print(f"Best Tour Length: {best_length:.4f}")

    # --- Visualization ---
    if best_tour:
        # Prepare coordinates for plotting the best tour path
        x_coords = [cities[i][0] for i in best_tour]
        y_coords = [cities[i][1] for i in best_tour]
        # Close the loop for visualization
        x_coords.append(x_coords[0])
        y_coords.append(y_coords[0])

        plt.figure(figsize=(10, 5))

        # Plot 1: Tour Path
        plt.subplot(1, 2, 1)
        plt.plot(x_coords, y_coords, 'o-', color='blue', markerfacecolor='red',
markersize=8)
        # Label cities with their index
        for i, (x, y) in enumerate(cities):
            plt.text(x + 0.1, y + 0.1, str(i), fontsize=9)
        plt.title(f'ACO Best TSP Tour (Length: {best_length:.2f})')
        plt.xlabel('X Coordinate')
        plt.ylabel('Y Coordinate')
        plt.grid(True)

        # Plot 2: Convergence History
        plt.subplot(1, 2, 2)
        plt.plot(history, color='green', linewidth=2)
        plt.title('ACO Convergence')
        plt.xlabel('Iteration')
        plt.ylabel('Best Tour Length')
        plt.grid(True)
```

```
        plt.tight_layout()
        plt.show()


# Execute the example
if __name__ == '__main__':
    run_aco_example()
```

Output:

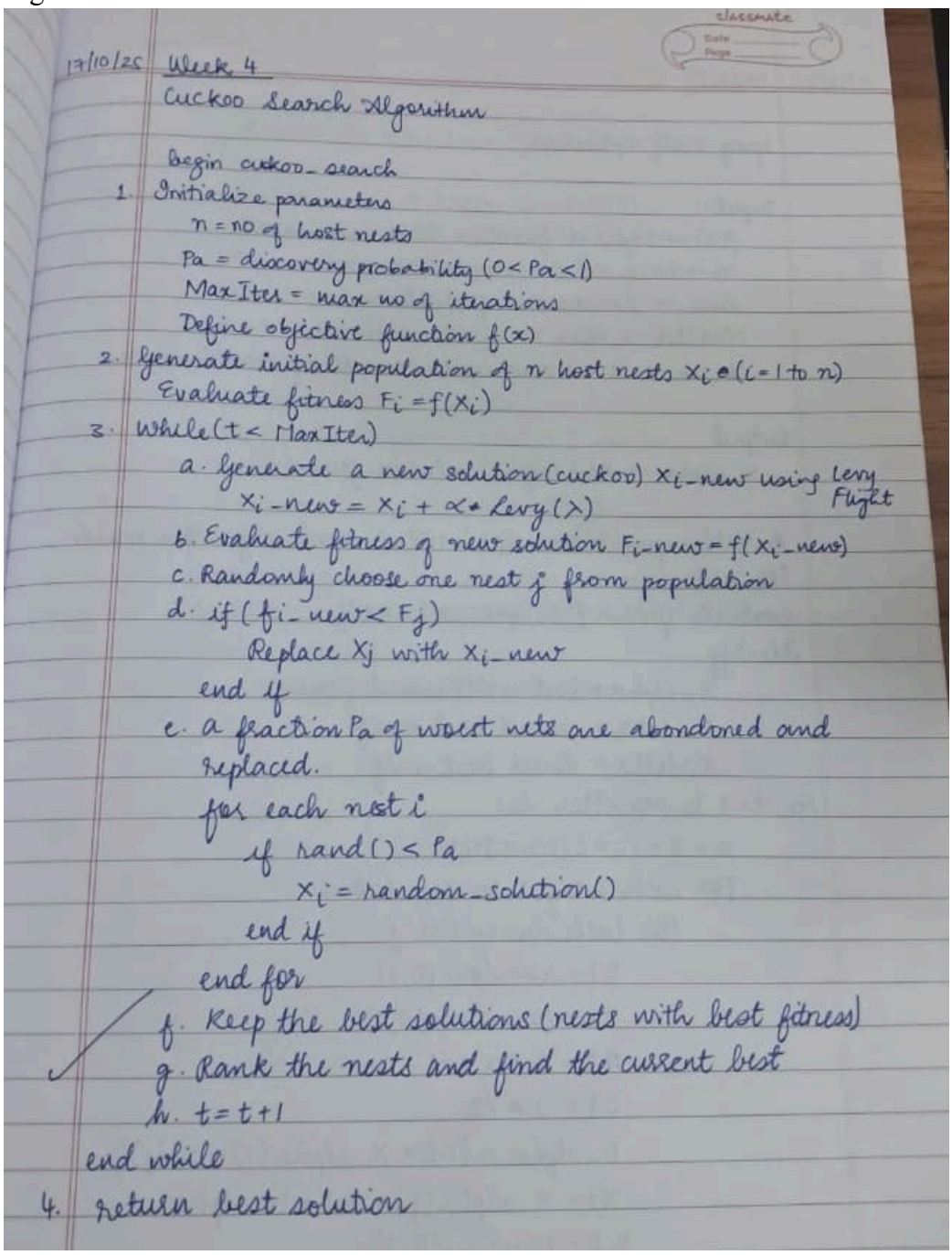
```
Starting ACO for 10 cities with 20 ants...
Iteration 1/50: Best Length = 27.17
Iteration 2/50: Best Length = 26.41
Iteration 3/50: Best Length = 26.41
Iteration 4/50: Best Length = 26.41
Iteration 5/50: Best Length = 26.41
Iteration 6/50: Best Length = 26.41
Iteration 7/50: Best Length = 26.41
Iteration 8/50: Best Length = 26.41
Iteration 9/50: Best Length = 26.41
Iteration 10/50: Best Length = 26.41
Iteration 11/50: Best Length = 26.41
Iteration 12/50: Best Length = 26.41
Iteration 13/50: Best Length = 26.41
Iteration 14/50: Best Length = 26.41
Iteration 15/50: Best Length = 26.41
Iteration 16/50: Best Length = 26.41
Iteration 17/50: Best Length = 26.41
Iteration 18/50: Best Length = 26.41
Iteration 19/50: Best Length = 26.41
Iteration 20/50: Best Length = 26.41
Iteration 21/50: Best Length = 26.41
Iteration 22/50: Best Length = 26.41
Iteration 23/50: Best Length = 26.41
Iteration 24/50: Best Length = 26.41
Iteration 25/50: Best Length = 26.41
Iteration 26/50: Best Length = 26.41
Iteration 27/50: Best Length = 26.41
Iteration 28/50: Best Length = 26.41
Iteration 29/50: Best Length = 26.41
Iteration 30/50: Best Length = 26.41
Iteration 31/50: Best Length = 26.41
Iteration 32/50: Best Length = 26.41
Iteration 33/50: Best Length = 26.41
Iteration 34/50: Best Length = 26.41
Iteration 35/50: Best Length = 26.41
Iteration 36/50: Best Length = 26.41
Iteration 37/50: Best Length = 26.41
Iteration 38/50: Best Length = 26.41
Iteration 39/50: Best Length = 26.41
Iteration 40/50: Best Length = 26.41
Iteration 41/50: Best Length = 26.41
Iteration 42/50: Best Length = 26.41
Iteration 43/50: Best Length = 26.41
Iteration 44/50: Best Length = 26.41
Iteration 45/50: Best Length = 26.41
Iteration 46/50: Best Length = 26.41
Iteration 47/50: Best Length = 26.41
Iteration 48/50: Best Length = 26.41
Iteration 49/50: Best Length = 26.41
Iteration 50/50: Best Length = 26.41

--- Results ---
Cities: [(6.394267984578837, 0.2501075522266936), (2.7582931836911926, 2.2321073814882277), (7.364712141640124, 6.766994874229113), (8.921795677048454,
8.869388326294161 5), (4.2192181968527045, 0.29797219438870344), (2.1863797480360336, 5.053552881033624), (0.2653596968386362 5), (1.988376586866485), (6.49884
4377795232, 5.440414806032166), (2.204406220406967, 5.892656838750888), (8.094304566778266, 0.06408759678061017)]
Best Tour (City Indices): [7, 2, 8, 5, 6, 1, 4, 0, 9, 3]
Best Tour Length: 26.4139
```



Program 5

Use the Cuckoo Search metaheuristic to minimize a given objective function (e.g., Sphere function) by iteratively improving candidate solutions through Lévy flights and random nest replacement.

Algorithm:



17/10/25 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_new using Levy Flight
      X_i_new = X_i + α * Levy(λ)
   b. Evaluate fitness of new solution F_i_new = f(X_i_new)
   c. Randomly choose one nest j from population
   d. if (f_i_new < F_j)
      Replace X_j with X_i_new
      end if
   e. a fraction Pa of worst nests are abandoned 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 nests and find the current best
   h. t = t + 1
   end while
4. return best solution

Code:

```
import numpy as np
import math

# Objective Function (example: minimize Sphere function)
def objective_function(x):
```

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

# Lévy flight function
def levy_flight(Lambda):
    # Using math module instead of np.math
    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.randn() * sigma
    v = np.random.randn()
    step = u / abs(v) ** (1 / Lambda)
    return step

# Cuckoo Search Algorithm
def cuckoo_search(obj_func, dim=2, n=15, pa=0.25, alpha=0.01, max_iter=100):
    # Initialize nests randomly
    nests = np.random.uniform(-5, 5, size=(n, dim))
    fitness = np.array([obj_func(x) for x in nests])

    # Find the current best nest
    best_idx = np.argmin(fitness)
    best = nests[best_idx].copy()

    for t in range(max_iter):
        for i in range(n):
            # Generate a new solution via Lévy flight
            step_size = alpha * levy_flight(1.5)
            new_nest = nests[i] + step_size * (nests[i] - best)
            new_nest = np.clip(new_nest, -5, 5)

            # Evaluate fitness
            f_new = obj_func(new_nest)

            # Replace if the new solution is better
            if f_new < fitness[i]:
                nests[i] = new_nest
                fitness[i] = f_new

        # Abandon some nests (with probability pa)
        for i in range(n):
            if np.random.rand() < pa:
                nests[i] = np.random.uniform(-5, 5, dim)
                fitness[i] = obj_func(nests[i])

        # Update the best solution
        best_idx = np.argmin(fitness)
        if fitness[best_idx] < obj_func(best):
            best = nests[best_idx].copy()

        # Print progress
        print(f"Iteration {t+1}: Best fitness = {obj_func(best):.6f}")

    return best, obj_func(best)

# Run the algorithm
best_solution, best_value = cuckoo_search(objective_function, dim=2, n=20, max_iter=50)
print("\nBest Solution:", best_solution)
print("Best Fitness Value:", best_value)
```

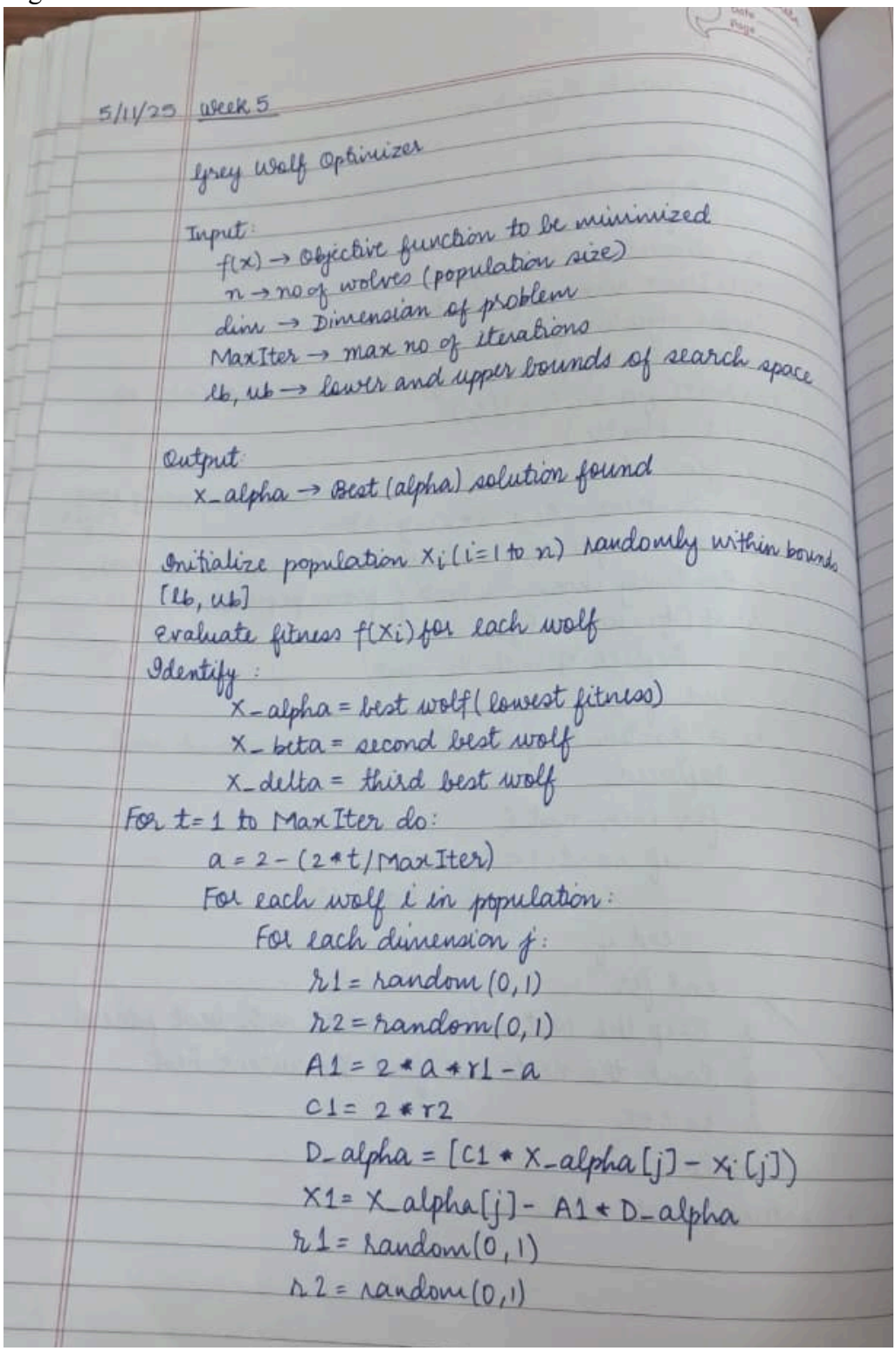
Output:

```
Iteration 1: Best fitness = 0.108432
Iteration 2: Best fitness = 0.108432
Iteration 3: Best fitness = 0.108432
Iteration 4: Best fitness = 0.108432
Iteration 5: Best fitness = 0.108432
Iteration 6: Best fitness = 0.108432
Iteration 7: Best fitness = 0.108432
Iteration 8: Best fitness = 0.108432
Iteration 9: Best fitness = 0.108432
Iteration 10: Best fitness = 0.108432
Iteration 11: Best fitness = 0.108432
Iteration 12: Best fitness = 0.108432
Iteration 13: Best fitness = 0.108432
Iteration 14: Best fitness = 0.108432
Iteration 15: Best fitness = 0.108432
Iteration 16: Best fitness = 0.108432
Iteration 17: Best fitness = 0.108432
Iteration 18: Best fitness = 0.108432
Iteration 19: Best fitness = 0.108432
Iteration 20: Best fitness = 0.108432
Iteration 21: Best fitness = 0.108432
Iteration 22: Best fitness = 0.108432
Iteration 23: Best fitness = 0.108432
Iteration 24: Best fitness = 0.108432
Iteration 25: Best fitness = 0.108432
Iteration 26: Best fitness = 0.108432
Iteration 27: Best fitness = 0.108432
Iteration 28: Best fitness = 0.108432
Iteration 29: Best fitness = 0.108432
Iteration 30: Best fitness = 0.108432
Iteration 31: Best fitness = 0.108432
Iteration 32: Best fitness = 0.108432
Iteration 33: Best fitness = 0.108432
Iteration 34: Best fitness = 0.108432
Iteration 35: Best fitness = 0.108432
Iteration 36: Best fitness = 0.108432
Iteration 37: Best fitness = 0.108432
Iteration 38: Best fitness = 0.108432
Iteration 39: Best fitness = 0.108432
Iteration 40: Best fitness = 0.108432
Iteration 41: Best fitness = 0.108432
Iteration 42: Best fitness = 0.108432
Iteration 43: Best fitness = 0.108432
Iteration 44: Best fitness = 0.108432
Iteration 45: Best fitness = 0.108432
Iteration 46: Best fitness = 0.108432
Iteration 47: Best fitness = 0.108432
Iteration 48: Best fitness = 0.108432
Iteration 49: Best fitness = 0.108432
Iteration 50: Best fitness = 0.108432

Best Solution: [0.27812108 0.17629828]
Best Fitness Value: 0.10843242171891111
```

Program 6

Implement the Grey Wolf Optimizer algorithm to minimize a mathematical objective function (e.g., f(x)=x2−4x+4f(x) = x^2 - 4x + 4f(x)=x2−4x+4) by simulating the social hierarchy and hunting behavior of grey wolves to find the global optimum.

Algorithm:



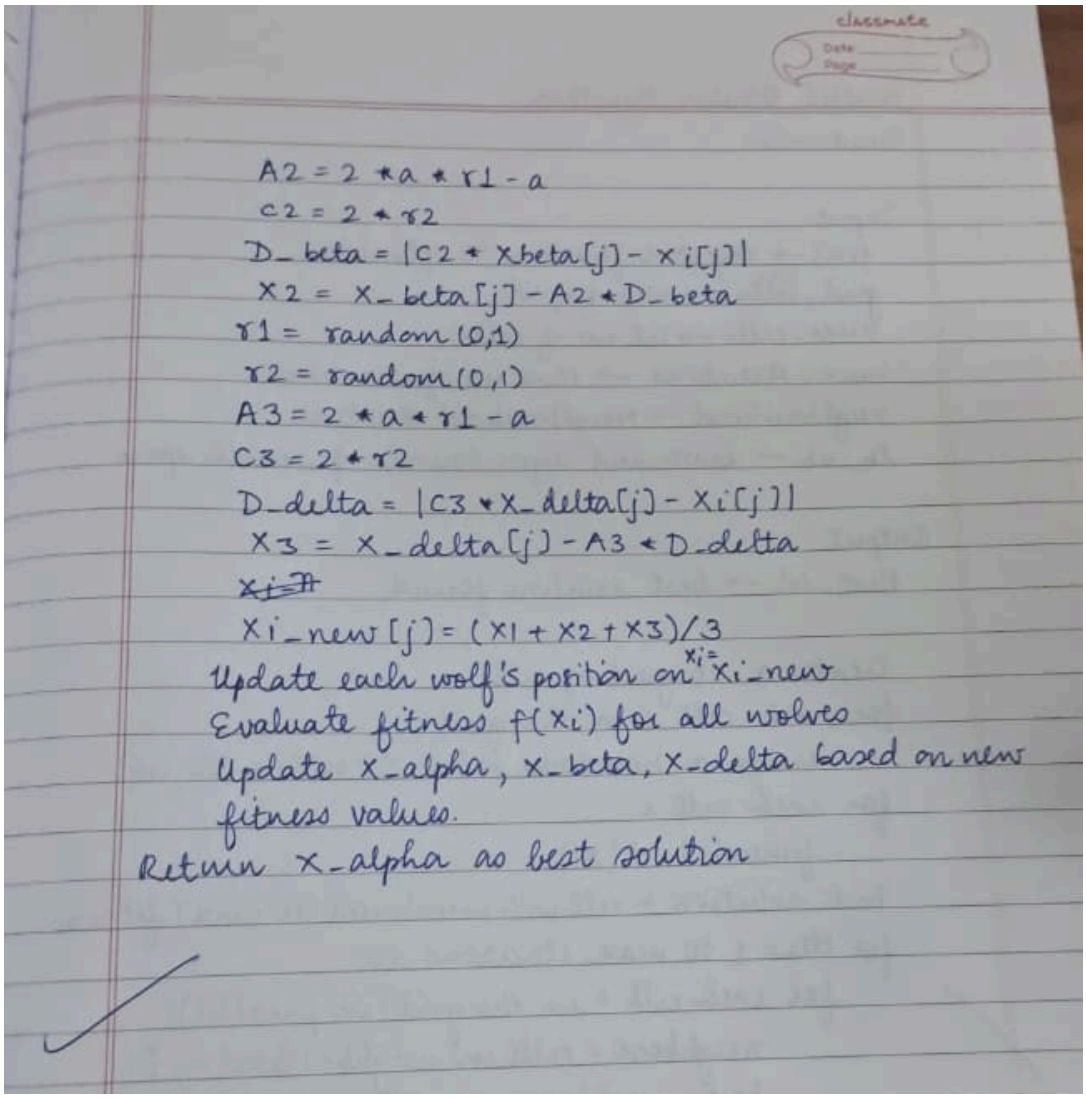
5/11/25 Week 5

Grey Wolf Optimizer

Input:
f(x) → Objective function to be minimized
n → no of wolves (population size)
dim → Dimension of problem
MaxIter → max no of iterations
lb, ub → lower and upper bounds of search space

Output:
X_alpha → Best (alpha) solution found

Initialize population X_i (i=1 to n) randomly within bounds [lb, ub]
evaluate fitness f(X_i) for each wolf
Identify :
X_alpha = best wolf (lowest fitness)
X_beta = second best wolf
X_delta = third best wolf
For t=1 to MaxIter do:
a = 2 − (2*t/MaxIter)
For each wolf i in population:
For each dimension j:
r1 = random(0,1)
r2 = random(0,1)
A1 = 2*a*r1 − a
C1 = 2*r2
D_alpha = [C1 * X_alpha[j] − X_i[j])
X1 = X_alpha[j] − A1 * D_alpha
r1 = random(0,1)
r2 = random(0,1)



Code:

```
import numpy as np

# Objective function (example: minimize f(x) = x^2 - 4x + 4)
def objective_function(x):
    return x[0]**2 - 4*x[0] + 4  # single-variable function

# Grey Wolf Optimizer
def GWO(obj_func, dim, n_wolves, max_iter, lb, ub):
    # Initialize positions of wolves randomly within [lb, ub]
    wolves = np.random.uniform(lb, ub, (n_wolves, dim))

    # Initialize alpha, beta, delta positions and fitness
    alpha_pos = np.zeros(dim)
    beta_pos = np.zeros(dim)
    delta_pos = np.zeros(dim)
    alpha_score = float("inf")
    beta_score = float("inf")
    delta_score = float("inf")
```

```
    # Main optimization loop
    for t in range(max_iter):
        for i in range(n_wolves):
            # Clip wolves to the boundary
            wolves[i] = np.clip(wolves[i], lb, ub)

            # Calculate fitness
            fitness = obj_func(wolves[i])

            # Update alpha, beta, delta
            if fitness < alpha_score:
                alpha_score, alpha_pos = fitness, wolves[i].copy()
            elif fitness < beta_score:
                beta_score, beta_pos = fitness, wolves[i].copy()
            elif fitness < delta_score:
                delta_score, delta_pos = fitness, wolves[i].copy()

        # Linearly decreasing coefficient 'a' from 2 to 0
        a = 2 - t * (2 / max_iter)

        # Update position of each wolf
        for i in range(n_wolves):
            for j in range(dim):
                r1, r2 = np.random.rand(), np.random.rand()
                A1 = 2 * a * r1 - a
                C1 = 2 * r2
                D_alpha = abs(C1 * alpha_pos[j] - wolves[i][j])
                X1 = alpha_pos[j] - A1 * D_alpha

                r1, r2 = np.random.rand(), np.random.rand()
                A2 = 2 * a * r1 - a
                C2 = 2 * r2
                D_beta = abs(C2 * beta_pos[j] - wolves[i][j])
                X2 = beta_pos[j] - A2 * D_beta

                r1, r2 = np.random.rand(), np.random.rand()
                A3 = 2 * a * r1 - a
                C3 = 2 * r2
                D_delta = abs(C3 * delta_pos[j] - wolves[i][j])
                X3 = delta_pos[j] - A3 * D_delta

                # Average of three best positions
                wolves[i][j] = (X1 + X2 + X3) / 3

        # Optional: Print progress
        print(f"Iteration {t+1}/{max_iter} → Best fitness: {alpha_score:.6f}")

    return alpha_pos, alpha_score

# Example usage
best_position, best_score = GWO(
    obj_func=objective_function,
    dim=1,              # single-variable function
    n_wolves=10,        # number of wolves
    max_iter=50,        # number of iterations
    lb=-10,             # lower bound
    ub=10               # upper bound
```

```
)

print("\nBest Solution Found:")
print("x =", best_position)
print("Fitness =", best_score)
```

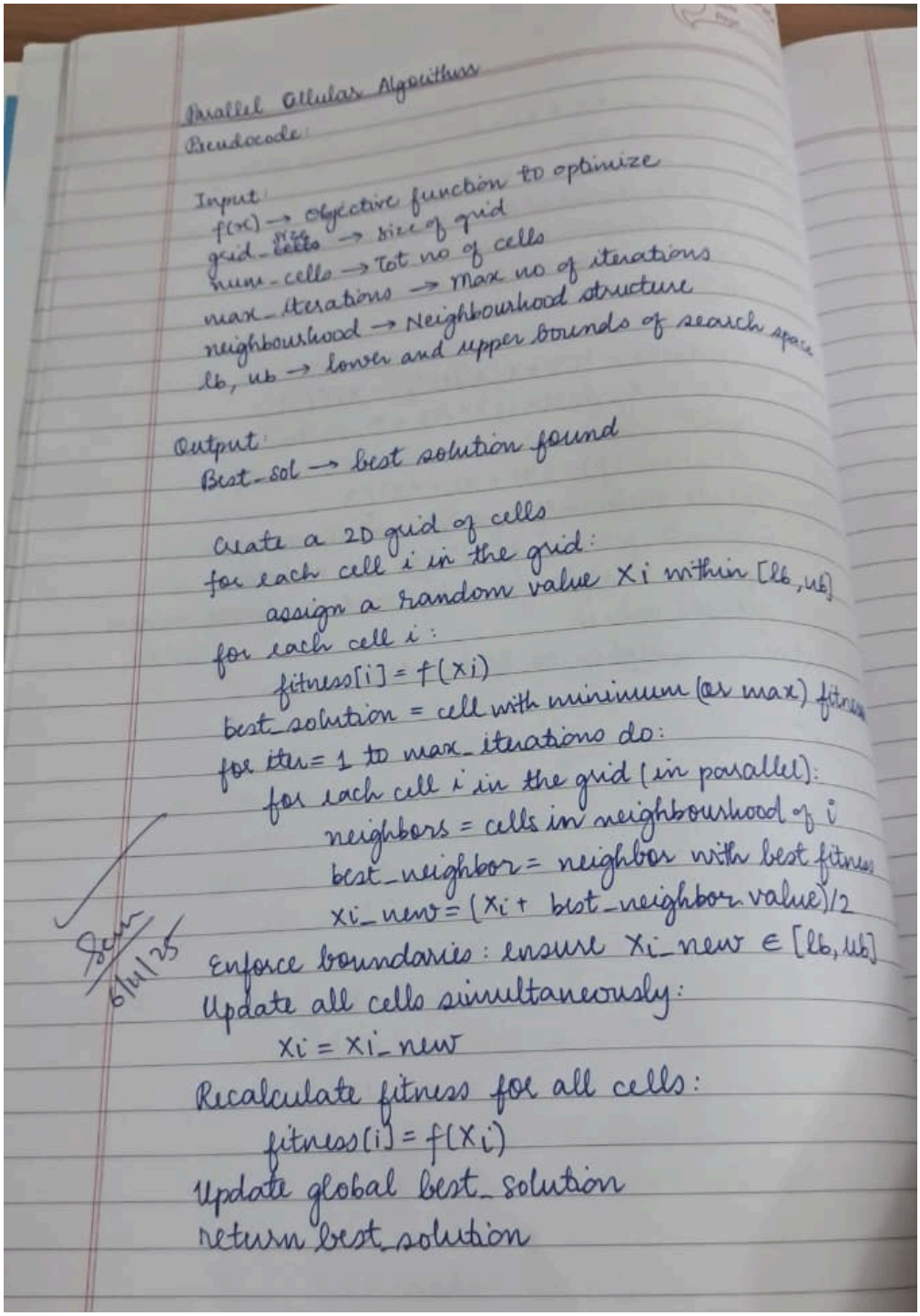
Output:

```
Iteration 1/50 → Best fitness: 0.041188
Iteration 2/50 → Best fitness: 0.041188
Iteration 3/50 → Best fitness: 0.041188
Iteration 4/50 → Best fitness: 0.041188
Iteration 5/50 → Best fitness: 0.004941
Iteration 6/50 → Best fitness: 0.002758
Iteration 7/50 → Best fitness: 0.002758
Iteration 8/50 → Best fitness: 0.002758
Iteration 9/50 → Best fitness: 0.001457
Iteration 10/50 → Best fitness: 0.001457
Iteration 11/50 → Best fitness: 0.000078
Iteration 12/50 → Best fitness: 0.000078
Iteration 13/50 → Best fitness: 0.000040
Iteration 14/50 → Best fitness: 0.000040
Iteration 15/50 → Best fitness: 0.000040
Iteration 16/50 → Best fitness: 0.000040
Iteration 17/50 → Best fitness: 0.000040
Iteration 18/50 → Best fitness: 0.000040
Iteration 19/50 → Best fitness: 0.000040
Iteration 20/50 → Best fitness: 0.000040
Iteration 21/50 → Best fitness: 0.000040
Iteration 22/50 → Best fitness: 0.000040
Iteration 23/50 → Best fitness: 0.000040
Iteration 24/50 → Best fitness: 0.000040
Iteration 25/50 → Best fitness: 0.000007
Iteration 26/50 → Best fitness: 0.000007
Iteration 27/50 → Best fitness: 0.000007
Iteration 28/50 → Best fitness: 0.000007
Iteration 29/50 → Best fitness: 0.000007
Iteration 30/50 → Best fitness: 0.000007
Iteration 31/50 → Best fitness: 0.000007
Iteration 32/50 → Best fitness: 0.000007
Iteration 33/50 → Best fitness: 0.000007
Iteration 34/50 → Best fitness: 0.000007
Iteration 35/50 → Best fitness: 0.000005
Iteration 36/50 → Best fitness: 0.000005
Iteration 37/50 → Best fitness: 0.000005
Iteration 38/50 → Best fitness: 0.000005
Iteration 39/50 → Best fitness: 0.000005
Iteration 40/50 → Best fitness: 0.000005
Iteration 41/50 → Best fitness: 0.000005
Iteration 42/50 → Best fitness: 0.000002
Iteration 43/50 → Best fitness: 0.000001
Iteration 44/50 → Best fitness: 0.000001
Iteration 45/50 → Best fitness: 0.000001
Iteration 46/50 → Best fitness: 0.000001
Iteration 47/50 → Best fitness: 0.000001
Iteration 48/50 → Best fitness: 0.000001
Iteration 49/50 → Best fitness: 0.000001
Iteration 50/50 → Best fitness: 0.000001

Best Solution Found:
x = [2.00108665]
Fitness = 1.1808108792976668e-06
```

Program 7

Implement the Parallel Cellular Algorithm to minimize a mathematical objective function (e.g., f(x)=x2−4x+4f(x) = x^2 - 4x + 4f(x)=x2−4x+4) by modeling a population of solutions arranged in a 2D grid, where each cell evolves based on the best solution in its local neighborhood to gradually approach the global optimum.

Algorithm:



Parallel Cellular Algorithm
Pseudocode:

Input:
f(x) → objective function to optimize
grid_cells → size of grid
num_cells → Tot no of cells
max_iterations → Max no of iterations
neighbourhood → Neighbourhood structure
lb, ub → lower and upper bounds of search space

Output:
Best_sol → best solution found

Create a 2D grid of cells
for each cell i in the grid:
    assign a random value Xi within [lb, ub]
for each cell i:
    fitness[i] = f(Xi)
best_solution = cell with minimum (or max) fitness
for iter = 1 to max_iterations do:
    for each cell i in the grid (in parallel):
        neighbors = cells in neighbourhood of i
        best_neighbor = neighbor with best fitness
        Xi_new = (Xi + best_neighbor value)/2
Enforce boundaries: ensure Xi_new ∈ [lb, ub]
Update all cells simultaneously:
    Xi = Xi_new
Recalculate fitness for all cells:
    fitness[i] = f(Xi)
Update global best_solution
return best_solution

Sam 6/11/25

Code:

```
import numpy as np

# Objective function to minimize
def objective_function(x):
    return x**2 - 4*x + 4    # simple convex function, min at x = 2

# Parallel Cellular Algorithm
def parallel_cellular_algorithm(obj_func, grid_size=(10, 10), lb=-10, ub=10,
max_iter=100):
    rows, cols = grid_size
    num_cells = rows * cols

    # Initialize cells randomly in search space
    cells = np.random.uniform(lb, ub, (rows, cols))

    # Evaluate initial fitness
    fitness = obj_func(cells)

    # Define neighborhood (3x3 Moore neighborhood)
    def get_neighbors(r, c):
        neighbors = []
        for i in range(r-1, r+2):
            for j in range(c-1, c+2):
                if (0 <= i < rows) and (0 <= j < cols) and not (i == r and j == c):
                    neighbors.append((i, j))
        return neighbors

    # Track best solution
    best_value = np.min(fitness)
    best_position = np.unravel_index(np.argmin(fitness), fitness.shape)

    # Main iteration loop
    for t in range(max_iter):
        new_cells = np.copy(cells)

        for r in range(rows):
            for c in range(cols):
                neighbors = get_neighbors(r, c)
                # Find the best neighbor
                neighbor_values = np.array([cells[i, j] for i, j in neighbors])
                neighbor_fitness = np.array([obj_func(cells[i, j]) for i, j in neighbors])

                best_neighbor_value = neighbor_values[np.argmin(neighbor_fitness)]

                # Update rule: average of current cell and best neighbor
                new_cells[r, c] = (cells[r, c] + best_neighbor_value) / 2

        # Apply boundaries
        new_cells = np.clip(new_cells, lb, ub)

        # Update cells and fitness
        cells = new_cells
        fitness = obj_func(cells)

        # Track global best
        current_best = np.min(fitness)
```

```
        if current_best < best_value:
            best_value = current_best
            best_position = np.unravel_index(np.argmin(fitness), fitness.shape)

        # Optional: print progress
        if (t + 1) % 10 == 0 or t == 0:
            print(f"Iteration {t+1}/{max_iter} → Best fitness: {best_value:.6f}")

    print("\nBest solution found:")
    print(f"x = {cells[best_position]:.6f}, fitness = {best_value:.6f}")
    return cells[best_position], best_value

# Example run
best_x, best_fit = parallel_cellular_algorithm(
    objective_function,
    grid_size=(10, 10),
    lb=-10,
    ub=10,
    max_iter=50
)
```

Output:

```
Iteration 1/50 → Best fitness: 0.000303
Iteration 10/50 → Best fitness: 0.000000
Iteration 20/50 → Best fitness: 0.000000
Iteration 30/50 → Best fitness: 0.000000
Iteration 40/50 → Best fitness: 0.000000
Iteration 50/50 → Best fitness: 0.000000

Best solution found:
x = 2.000000, fitness = 0.000000
```