VISVESVARAYA TECHNOLOGICAL UNIVERSITY

"JnanaSangama", Belgaum -590014, Karnataka.

LAB RECORD

Bio Inspired Systems (23CS5BSBIS)

Submitted by

Archita V(1BM23CS050)

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 **Archita V (1BM23CS050),** 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.

Prameetha Pai Assistant Professor Department of CSE, BMSCE	Dr. Kavitha Sooda Professor & HOD Department of CSE, BMSCE

Index

Github Link:

https://github.com/1BM23CS050/BIS-LA

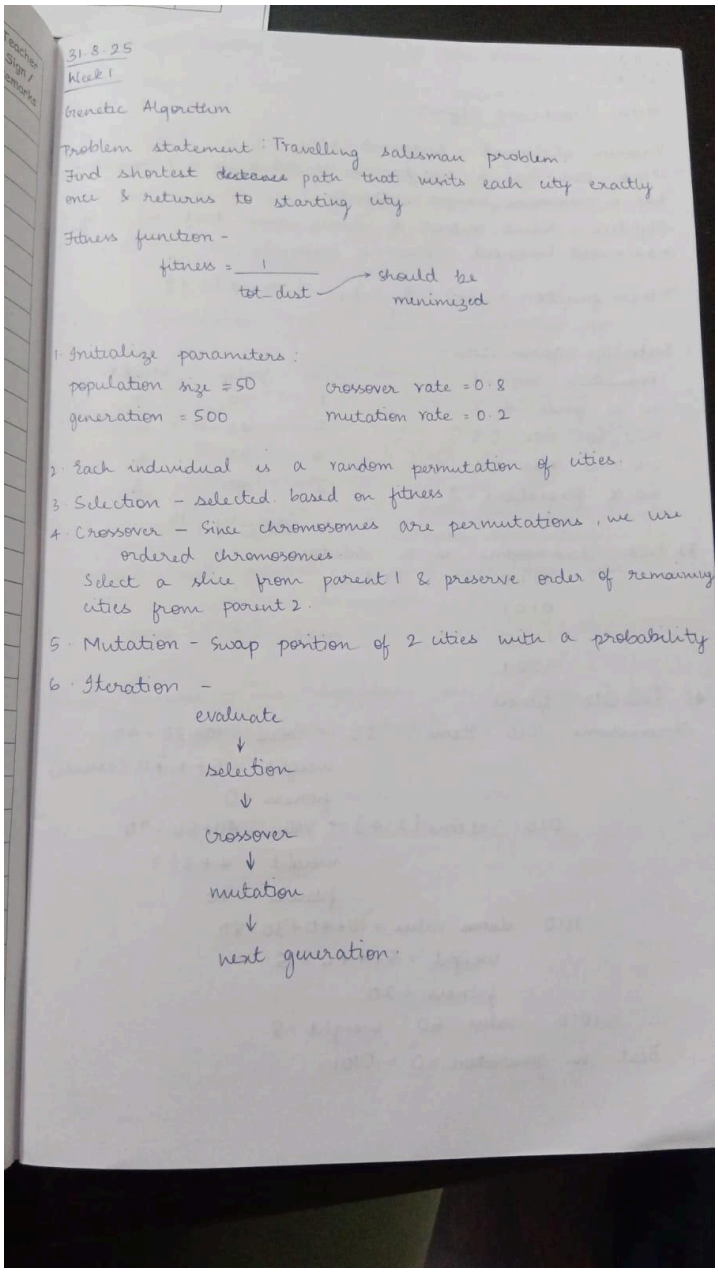
Program 1

Problem Statement:

Genetic Algorithm for Optimization Problems:

Find the best routes for delivery vehicles with limited capacity so that all customers are served once and total travel distance is minimized using a Genetic Algorithm.

Algorithm:



Code:

```
import math
import random
import copy
from typing import List, Tuple

random.seed(1)

# --------------------------
# Utility functions
# --------------------------

def euclidean(a: Tuple[float, float], b: Tuple[float, float]) -> float:
    return math.hypot(a[0] - b[0], a[1] - b[1])

def compute_distance_matrix(coords: List[Tuple[float, float]]) -> List[List[float]]:
    n = len(coords)
    dist = [[0.0]*n for _ in range(n)]
    for i in range(n):
        for j in range(n):
            dist[i][j] = euclidean(coords[i], coords[j])
    return dist

# --------------------------
# Problem instance generator
# --------------------------

def generate_instance(num_customers=15, grid_size=100, max_demand=10, depot=(50,50)):
    """
    Generates random customers with demands and coordinates.
    Index 0 is the depot, customers are 1..num_customers
    """
    coords = [depot]
    demands = [0]
    for _ in range(num_customers):
        coords.append((random.uniform(0, grid_size), random.uniform(0, grid_size)))
        demands.append(random.randint(1, max_demand))
    return coords, demands

# --------------------------
# Chromosome -> Routes (split by capacity)
# --------------------------

def split_routes_from_perm(perm: List[int], demands: List[int], capacity: int) -> List[List[int]]:
    """
    Greedy split: take customer sequence in perm, keep adding to current route
```

```
    while capacity is not exceeded. Start new route when needed.
    """
    routes = []
    cur_route = []
    cur_load = 0
    for cust in perm:
        d = demands[cust]
        if cur_load + d <= capacity:
            cur_route.append(cust)
            cur_load += d
        else:
            # close route and start new
            if cur_route:
                routes.append(cur_route)
            cur_route = [cust]
            cur_load = d
    if cur_route:
        routes.append(cur_route)
    return routes

def route_cost(route: List[int], dist_matrix: List[List[float]]) -> float:
    if not route:
        return 0.0
    cost = 0.0
    prev = 0  # depot
    for cust in route:
        cost += dist_matrix[prev][cust]
        prev = cust
    cost += dist_matrix[prev][0]  # return to depot
    return cost

def total_cost(routes: List[List[int]], dist_matrix: List[List[float]]) -> float:
    return sum(route_cost(r, dist_matrix) for r in routes)

# --------------------------
# Genetic Algorithm components
# --------------------------

def init_population(pop_size: int, customer_ids: List[int]) -> List[List[int]]:
    pop = []
    for _ in range(pop_size):
        perm = customer_ids[:]
        random.shuffle(perm)
        pop.append(perm)
    return pop

def tournament_selection(pop: List[List[int]], fitnesses: List[float], k=3) -> List[int]:
```

```
    selected = random.sample(list(range(len(pop))), k)
    best = min(selected, key=lambda i: fitnesses[i])
    return copy.deepcopy(pop[best])

def order_crossover(parent1: List[int], parent2: List[int]) -> Tuple[List[int], List[int]]:
    """Order Crossover (OX) for permutations."""
    n = len(parent1)
    a, b = sorted(random.sample(range(n), 2))
    def ox(p1, p2):
        child = [None]*n
        # copy slice
        child[a:b+1] = p1[a:b+1]
        # fill remaining from p2 order
        pos = (b+1) % n
        p2_i = (b+1) % n
        while None in child:
            val = p2[p2_i]
            if val not in child:
                child[pos] = val
                pos = (pos+1) % n
            p2_i = (p2_i+1) % n
        return child
    return ox(parent1, parent2), ox(parent2, parent1)

def swap_mutation(perm: List[int], mut_rate=0.2) -> None:
    """In-place swap mutation (may perform multiple swaps based on mut_rate)."""
    n = len(perm)
    for i in range(n):
        if random.random() < mut_rate:
            j = random.randrange(n)
            perm[i], perm[j] = perm[j], perm[i]

# --------------------------
# GA main loop
# --------------------------

def evaluate_population(pop: List[List[int]], demands: List[int], capacity: int, dist_matrix:
List[List[float]]):
    fitnesses = []
    for indiv in pop:
        routes = split_routes_from_perm(indiv, demands, capacity)
        fitnesses.append(total_cost(routes, dist_matrix))
    return fitnesses

def ga_vrp(
    coords: List[Tuple[float, float]],
    demands: List[int],
```

```
    capacity: int,
    pop_size=100,
    gens=300,
    crossover_rate=0.8,
    mutation_rate=0.15,
    elitism=2
):
    n = len(coords) - 1  # number of customers
    customer_ids = list(range(1, n+1))
    dist_matrix = compute_distance_matrix(coords)

    pop = init_population(pop_size, customer_ids)
    fitnesses = evaluate_population(pop, demands, capacity, dist_matrix)

    best_history = []
    for gen in range(gens):
        new_pop = []
        # Elitism: keep best individuals
        sorted_idx = sorted(range(len(pop)), key=lambda i: fitnesses[i])
        for i in range(elitism):
            new_pop.append(copy.deepcopy(pop[sorted_idx[i]]))

        while len(new_pop) < pop_size:
            # Selection
            p1 = tournament_selection(pop, fitnesses, k=3)
            p2 = tournament_selection(pop, fitnesses, k=3)
            # Crossover
            if random.random() < crossover_rate:
                c1, c2 = order_crossover(p1, p2)
            else:
                c1, c2 = p1, p2
            # Mutation
            swap_mutation(c1, mutation_rate)
            swap_mutation(c2, mutation_rate)
            new_pop.append(c1)
            if len(new_pop) < pop_size:
                new_pop.append(c2)

        pop = new_pop
        fitnesses = evaluate_population(pop, demands, capacity, dist_matrix)
        gen_best = min(fitnesses)
        best_history.append(gen_best)

        if (gen+1) % (max(1, gens//10)) == 0 or gen == 0:
            print(f"Gen {gen+1:4d} | best cost = {gen_best:.2f}")

    # final best
```

```
    best_idx = min(range(len(pop)), key=lambda i: fitnesses[i])
    best_perm = pop[best_idx]
    best_routes = split_routes_from_perm(best_perm, demands, capacity)
    best_cost = fitnesses[best_idx]
    return {
        'best_perm': best_perm,
        'best_routes': best_routes,
        'best_cost': best_cost,
        'dist_matrix': dist_matrix,
        'history': best_history
    }

# --------------------------
# Example / Run
# --------------------------

if __name__ == "__main__":
    # Example instance
    NUM_CUSTOMERS = 12
    VEHICLE_CAPACITY = 30
    coords, demands = generate_instance(num_customers=NUM_CUSTOMERS, grid_size=100,
max_demand=12, depot=(50,50))

    print("Depot coords:", coords[0])
    for i in range(1, len(coords)):
        print(f"Customer {i:2d} at {coords[i]} demand={demands[i]}")

    result = ga_vrp(
        coords=coords,
        demands=demands,
        capacity=VEHICLE_CAPACITY,
        pop_size=120,
        gens=300,
        crossover_rate=0.9,
        mutation_rate=0.12,
        elitism=4
    )

    print("\n=== BEST SOLUTION ===")
    print(f"Total cost: {result['best_cost']:.2f}")
    print("Routes (customers listed, depot implicit at start/end):")
    for idx, r in enumerate(result['best_routes'], start=1):
        load = sum(demands[c] for c in r)
        rcost = route_cost(r, result['dist_matrix'])
        print(f" Vehicle {idx}: {r} | load={load} | route_cost={rcost:.2f}")
    print("\nBest permutation (customer visit order):", result['best_perm'])
```

Output:

```
Depot coords: (50, 50)
Customer  1 at (13.436424411240122, 84.74337369372327) demand=2
Customer  2 at (25.50690257394217, 49.54350870919409) demand=8
Customer  3 at (47.224524357611664, 37.96152233237278) demand=4
Customer  4 at (9.385958677423488, 2.834747652200631) demand=7
Customer  5 at (43.27670679050534, 76.2280082457942) demand=1
Customer  6 at (69.58328667684435, 26.633056045725954) demand=4
Customer  7 at (59.11534350013039, 10.222715811004823) demand=6
Customer  8 at (3.0589983033553536, 2.54458609934608) demand=9
Customer  9 at (0.9204938554384978, 88.12338589221554) demand=11
Customer 10 at (21.659939713061338, 42.21165755827173) demand=1
Customer 11 at (52.76294143623982, 76.37009951314894) demand=8
Customer 12 at (55.28595762929651, 34.570041470875246) demand=11
Gen    1 | best cost = 483.53
Gen   30 | best cost = 393.75
Gen   60 | best cost = 381.97
Gen   90 | best cost = 381.97
Gen  120 | best cost = 381.97
Gen  150 | best cost = 381.97
Gen  180 | best cost = 381.97
Gen  210 | best cost = 381.97
Gen  240 | best cost = 381.97
Gen  270 | best cost = 381.97
Gen  300 | best cost = 381.97

=== BEST SOLUTION ===
Total cost: 381.97
Routes (customers listed, depot implicit at start/end):
 Vehicle 1: [9, 1, 5, 11] | load=22 | route_cost=142.14
 Vehicle 2: [12, 6, 7, 3] | load=25 | route_cost=94.66
 Vehicle 3: [2, 10, 8, 4] | load=25 | route_cost=145.16

Best permutation (customer visit order): [9, 1, 5, 11, 12, 6, 7, 3, 2, 10, 8, 4]
```

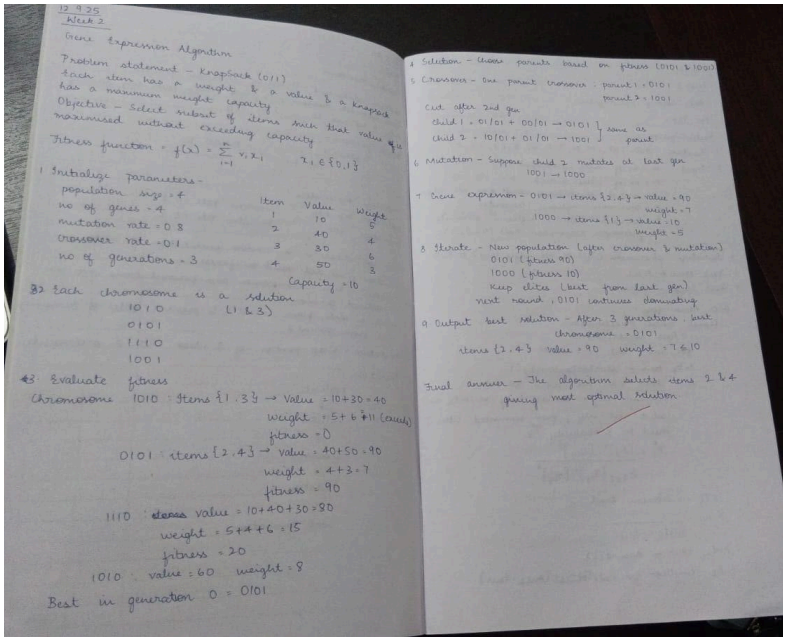
Program 2

Problem Statement:

Optimization via Gene Expression Algorithms:

Use Gene Expression Programming to find the best mathematical expression or variable values that optimize a given function.

Algorithm:



Code:

```
import random
import math

# Target function to approximate
def target_function(x):
    return x**2 + x + 1

# Terminal and function sets
TERMINALS = ['x']
FUNCTIONS = ['+', '-', '*', '/']

# Parameters
POP_SIZE = 100
MUTATION_RATE = 0.1
TOURNAMENT_SIZE = 3
```

```
GENERATIONS = 100

# ---------------------
# Utility functions
# ---------------------

def safe_div(a, b):
    """Safe division to avoid divide-by-zero."""
    try:
        return a / b if b != 0 else 1
    except:
        return 1

def random_gene(max_depth=3):
    """Randomly create an expression (gene) of limited depth."""
    if max_depth == 0 or (max_depth > 1 and random.random() < 0.3):
        return random.choice(TERMINALS)
    else:
        op = random.choice(FUNCTIONS)
        left = random_gene(max_depth - 1)
        right = random_gene(max_depth - 1)
        return f"({left}{op}{right})"

def evaluate_expression(expr, x):
    """Safely evaluate the evolved expression."""
    try:
        # Define the environment so eval can use math and safe_div properly
        env = {"x": x, "math": math, "safe_div": safe_div}
        # Replace '/' with safe_div(a,b)
        expr_safe = expr.replace("/", "safe_div")
        result = eval(expr_safe, env)
        if callable(result):  # in case a function is accidentally returned
            return 0
        return float(result)
    except Exception:
        return 0

def fitness(expr):
    """Mean squared error against target function."""
    xs = [i for i in range(-10, 11)]
    errors = []
    for x in xs:
        y_true = target_function(x)
        y_pred = evaluate_expression(expr, x)
        errors.append((y_true - y_pred)**2)
    mse = sum(errors) / len(errors)
    return mse
```

```
# ---------------------
# Genetic operators
# ---------------------

def mutate(expr):
    """Randomly mutate part of the expression."""
    expr_list = list(expr)
    for i in range(len(expr_list)):
        if random.random() < MUTATION_RATE:
            expr_list[i] = random.choice(['x', '+', '-', '*', '/'])
    return ''.join(expr_list)

def crossover(parent1, parent2):
    """Single-point crossover."""
    if len(parent1) < 2 or len(parent2) < 2:
        return parent1, parent2
    cut1 = random.randint(1, len(parent1) - 1)
    cut2 = random.randint(1, len(parent2) - 1)
    child1 = parent1[:cut1] + parent2[cut2:]
    child2 = parent2[:cut2] + parent1[cut1:]
    return child1, child2

def tournament_selection(pop, fits):
    """Select the best of k random individuals."""
    chosen = random.sample(list(zip(pop, fits)), TOURNAMENT_SIZE)
    return min(chosen, key=lambda x: x[1])[0]

# ---------------------
# Main GEP loop
# ---------------------

def gene_expression_optimization():
    population = [random_gene() for _ in range(POP_SIZE)]

    for gen in range(GENERATIONS):
        fitnesses = [fitness(expr) for expr in population]
        best_idx = min(range(len(fitnesses)), key=lambda i: fitnesses[i])
        best_expr = population[best_idx]
        best_fit = fitnesses[best_idx]

        if gen % 10 == 0 or gen == GENERATIONS - 1:
            print(f"Generation {gen:3d} | Best Expr: {best_expr} | Fitness = {best_fit:.6f}")

        new_pop = [best_expr]  # elitism
        while len(new_pop) < POP_SIZE:
            p1 = tournament_selection(population, fitnesses)
```

```
            p2 = tournament_selection(population, fitnesses)
            c1, c2 = crossover(p1, p2)
            new_pop.extend([mutate(c1), mutate(c2)])
        population = new_pop[:POP_SIZE]

    print("\n✅ Optimization Complete!")
    print(f"Best evolved expression: {best_expr}")
    return best_expr

# ---------------------
# Run
# ---------------------
if __name__ == "__main__":
    best = gene_expression_optimization()

    print("\nSample comparison:")
    for x in [-3, 0, 2, 5]:
        print(f"x={x:>2} | Target={target_function(x):>8.3f} | Evolved={evaluate_expression(best,
x):>8.3f}")
```

Output:

```
Generation   0 | Best Expr: x | Fitness = 2487.000000
Generation  10 | Best Expr: x | Fitness = 2487.000000
Generation  20 | Best Expr: x | Fitness = 2487.000000
Generation  30 | Best Expr: x | Fitness = 2487.000000
Generation  40 | Best Expr: x | Fitness = 2487.000000
Generation  50 | Best Expr: x | Fitness = 2487.000000
Generation  60 | Best Expr: x | Fitness = 2487.000000
Generation  70 | Best Expr: x | Fitness = 2487.000000
Generation  80 | Best Expr: x | Fitness = 2487.000000
Generation  90 | Best Expr: x | Fitness = 2487.000000
Generation  99 | Best Expr: x | Fitness = 2487.000000

✅ Optimization Complete!
Best evolved expression: x

Sample comparison:
x=-3 | Target=   7.000 | Evolved=  -3.000
x= 0 | Target=   1.000 | Evolved=   0.000
x= 2 | Target=   7.000 | Evolved=   2.000
x= 5 | Target=  31.000 | Evolved=   5.000
```

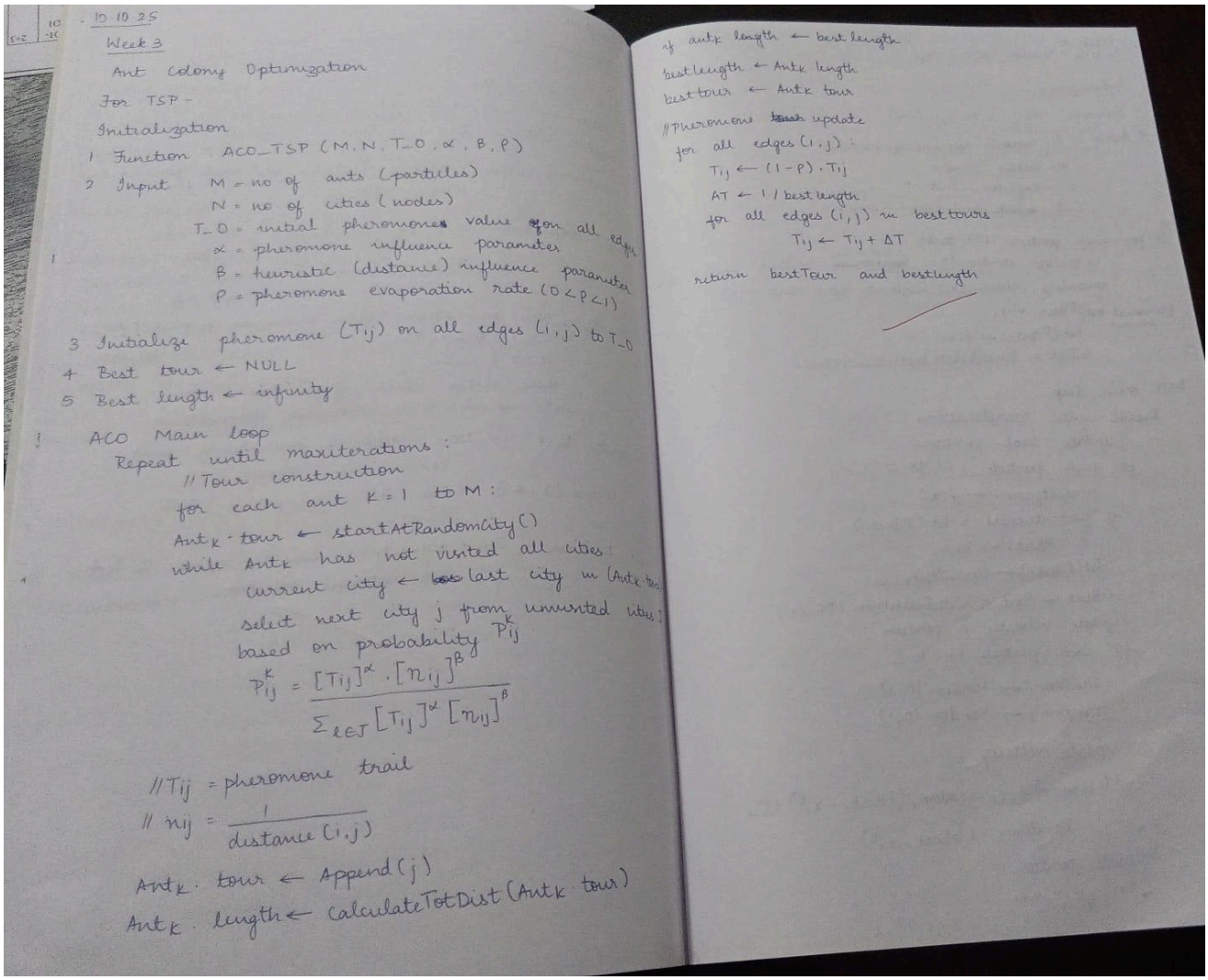
Program 3

Problem Statement:

Ant Colony Optimization for the Traveling Salesman Problem:

Use Ant Colony Optimization to find the shortest possible route that visits all cities exactly once and returns to the starting city.

Algorithm:



Code:

```
import random
import math

# ----------------------------
# PARAMETERS
# ----------------------------
```

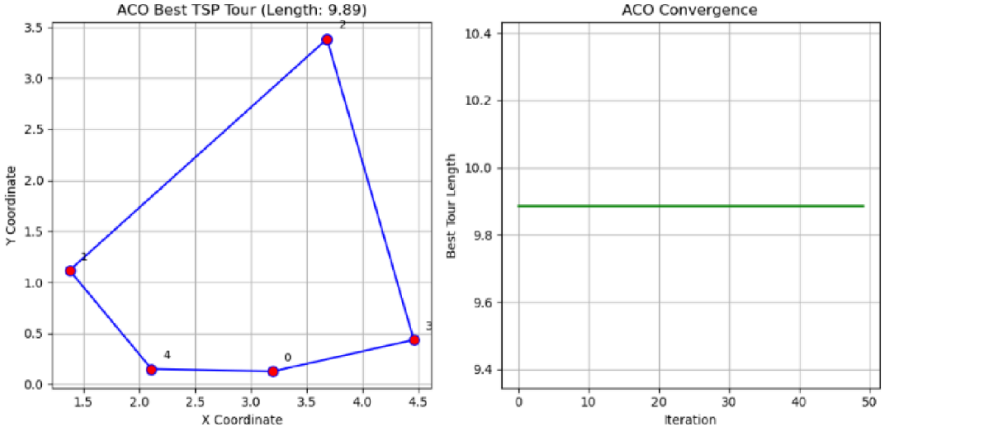
```
NUM_CITIES = 10
NUM_ANTS = 20
ALPHA = 1.0       # pheromone importance
BETA = 5.0        # distance importance
RHO = 0.5         # evaporation rate
Q = 100           # pheromone deposit factor
ITERATIONS = 100

random.seed(1)

# ------------------------------
# CREATE TSP INSTANCE
# ------------------------------
def generate_cities(n, grid_size=100):
    """Randomly generate city coordinates."""
    return [(random.uniform(0, grid_size), random.uniform(0, grid_size)) for _ in range(n)]

def distance(a, b):
    return math.hypot(a[0] - b[0], a[1] - b[1])

def compute_distance_matrix(cities):
    n = len(cities)
    dist = [[0]*n for _ in range(n)]
    for i in range(n):
        for j in range(n):
            if i != j:
                dist[i][j] = distance(cities[i], cities[j])
    return dist

# ------------------------------
# ACO Core
# ------------------------------
def initialize_pheromones(n):
    """Initialize pheromone levels between cities."""
    return [[1.0 for _ in range(n)] for _ in range(n)]

def select_next_city(probabilities):
    """Roulette-wheel selection."""
    r = random.random()
    cumulative = 0.0
    for i, p in enumerate(probabilities):
        cumulative += p
        if r <= cumulative:
            return i
    return len(probabilities) - 1

def ant_tour(dist_matrix, pheromone):
```

```
    """Construct a complete tour for one ant."""
    n = len(dist_matrix)
    unvisited = list(range(n))
    start = random.choice(unvisited)
    tour = [start]
    unvisited.remove(start)

    while unvisited:
        current = tour[-1]
        probs = []
        denom = 0.0
        for j in unvisited:
            denom += (pheromone[current][j] ** ALPHA) * ((1.0 / dist_matrix[current][j]) ** BETA)
        for j in unvisited:
            num = (pheromone[current][j] ** ALPHA) * ((1.0 / dist_matrix[current][j]) ** BETA)
            probs.append(num / denom if denom > 0 else 0)
        next_city = unvisited[select_next_city(probs)]
        tour.append(next_city)
        unvisited.remove(next_city)
    return tour

def tour_length(tour, dist_matrix):
    """Calculate total length of the tour."""
    total = 0.0
    for i in range(len(tour) - 1):
        total += dist_matrix[tour[i]][tour[i+1]]
    total += dist_matrix[tour[-1]][tour[0]]  # return to start
    return total

def update_pheromones(pheromone, all_tours, all_lengths):
    """Evaporate and deposit pheromones based on ant tours."""
    n = len(pheromone)
    # Evaporation
    for i in range(n):
        for j in range(n):
            pheromone[i][j] *= (1 - RHO)
            if pheromone[i][j] < 1e-6:
                pheromone[i][j] = 1e-6

    # Deposit pheromones
    for tour, L in zip(all_tours, all_lengths):
        deposit = Q / L
        for i in range(len(tour) - 1):
            a, b = tour[i], tour[i+1]
            pheromone[a][b] += deposit
            pheromone[b][a] += deposit
        # return edge
```

```
        pheromone[tour[-1]][tour[0]] += deposit
        pheromone[tour[0]][tour[-1]] += deposit

# ----------------------------
# ACO MAIN LOOP
# ----------------------------
def aco_tsp(num_cities=NUM_CITIES):
    cities = generate_cities(num_cities)
    dist_matrix = compute_distance_matrix(cities)
    pheromone = initialize_pheromones(num_cities)

    best_tour = None
    best_length = float('inf')

    for it in range(ITERATIONS):
        all_tours = []
        all_lengths = []

        for ant in range(NUM_ANTS):
            tour = ant_tour(dist_matrix, pheromone)
            L = tour_length(tour, dist_matrix)
            all_tours.append(tour)
            all_lengths.append(L)

            if L < best_length:
                best_length = L
                best_tour = tour

        update_pheromones(pheromone, all_tours, all_lengths)

        if (it + 1) % 10 == 0 or it == 0:
            print(f"Iteration {it+1:3d} | Best Length: {best_length:.2f}")

    print("\n✅ Optimization Complete!")
    print(f"Best Tour Length: {best_length:.2f}")
    print(f"Best Tour: {best_tour}")
    return best_tour, best_length, cities

# ----------------------------
# RUN
# ----------------------------
if __name__ == "__main__":
    best_tour, best_length, cities = aco_tsp()
```

Output:

```
Starting ACO for 5 cities with 10 ants...
Iteration 1/50: Best Length = 9.89
Iteration 2/50: Best Length = 9.89
Iteration 3/50: Best Length = 9.89
Iteration 4/50: Best Length = 9.89
Iteration 5/50: Best Length = 9.89
Iteration 6/50: Best Length = 9.89
Iteration 7/50: Best Length = 9.89
Iteration 8/50: Best Length = 9.89
Iteration 9/50: Best Length = 9.89
Iteration 10/50: Best Length = 9.89
Iteration 11/50: Best Length = 9.89
Iteration 12/50: Best Length = 9.89
Iteration 13/50: Best Length = 9.89
Iteration 14/50: Best Length = 9.89
Iteration 15/50: Best Length = 9.89
Iteration 16/50: Best Length = 9.89
Iteration 17/50: Best Length = 9.89
Iteration 18/50: Best Length = 9.89
Iteration 19/50: Best Length = 9.89
Iteration 20/50: Best Length = 9.89
Iteration 21/50: Best Length = 9.89
Iteration 22/50: Best Length = 9.89
Iteration 23/50: Best Length = 9.89
Iteration 24/50: Best Length = 9.89
Iteration 25/50: Best Length = 9.89
Iteration 26/50: Best Length = 9.89
Iteration 27/50: Best Length = 9.89
Iteration 28/50: Best Length = 9.89
Iteration 29/50: Best Length = 9.89
Iteration 30/50: Best Length = 9.89
Iteration 31/50: Best Length = 9.89
Iteration 32/50: Best Length = 9.89
Iteration 33/50: Best Length = 9.89
Iteration 34/50: Best Length = 9.89
Iteration 35/50: Best Length = 9.89
Iteration 36/50: Best Length = 9.89
Iteration 37/50: Best Length = 9.89
Iteration 38/50: Best Length = 9.89
Iteration 39/50: Best Length = 9.89
Iteration 40/50: Best Length = 9.89
Iteration 41/50: Best Length = 9.89
Iteration 42/50: Best Length = 9.89
Iteration 43/50: Best Length = 9.89
Iteration 44/50: Best Length = 9.89
Iteration 45/50: Best Length = 9.89
Iteration 46/50: Best Length = 9.89
Iteration 47/50: Best Length = 9.89
Iteration 48/50: Best Length = 9.89
Iteration 49/50: Best Length = 9.89
Iteration 50/50: Best Length = 9.89

--- Results ---
Cities: [(3.19713399222894186, 0.12505377611333468), (1.3751465918455963, 1.1160536907441139), (3.6823560708200062, 3.3834974371145563), (4.460897838524227, 0.43469416314708076), (2.1096809984263522, 0.14898609719035172)]
Best Tour (City Indices): [2, 3, 0, 4, 1]
Best Tour Length: 9.8880
```



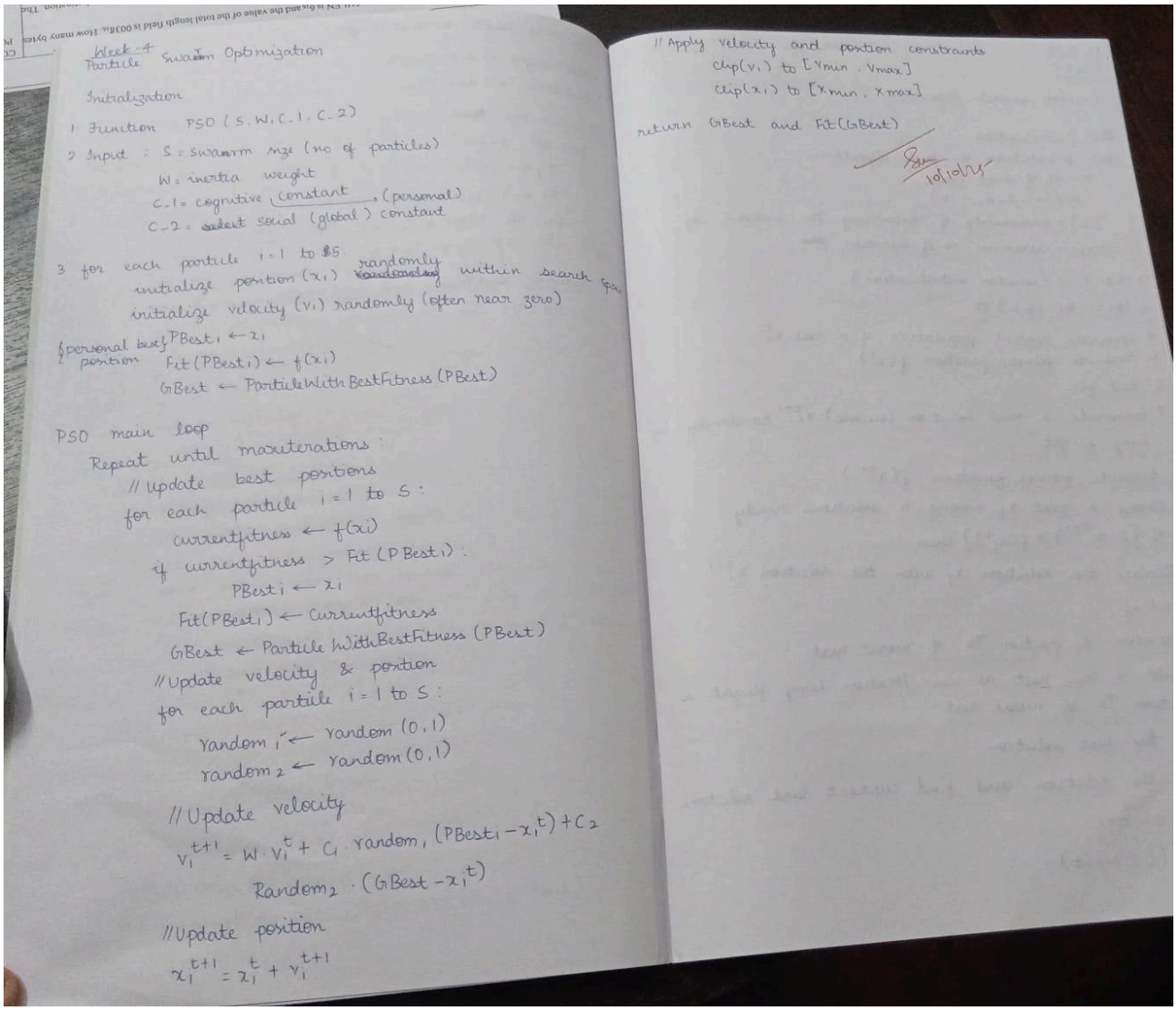
Program 4

Problem Statement:

Particle Swarm Optimization for Function Optimization:

Apply Particle Swarm Optimization to find the optimal solution (minimum or maximum) of a given mathematical function.

Algorithm:



Code:

```
import random
import math

# Define the mathematical function to optimize (minimize)
```

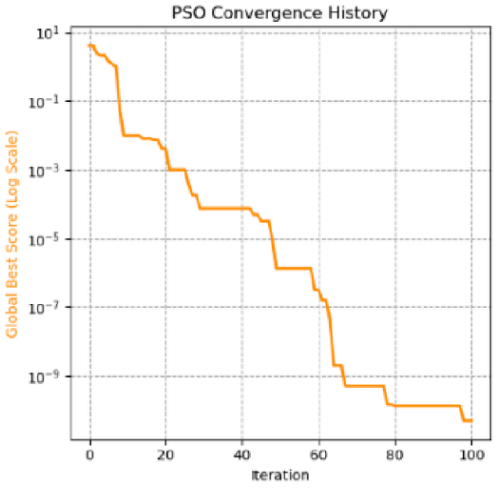
```
def objective_function(x, y):
    return x**2 + y**2  # Sphere function

# PSO parameters
num_particles = 30
max_iterations = 100
w = 0.7      # inertia weight
c1 = 1.5     # cognitive (particle) weight
c2 = 1.5     # social (swarm) weight

# Initialize particles
particles = []
for i in range(num_particles):
    x = random.uniform(-10, 10)
    y = random.uniform(-10, 10)
    velocity = [random.uniform(-1, 1), random.uniform(-1, 1)]
    fitness = objective_function(x, y)
    particles.append({
        "position": [x, y],
        "velocity": velocity,
        "best_pos": [x, y],
        "best_fit": fitness
    })

# Initialize global best
global_best = min(particles, key=lambda p: p["best_fit"])
global_best_pos = global_best["best_pos"]
global_best_fit = global_best["best_fit"]

# Main PSO loop
for t in range(max_iterations):
    for particle in particles:
        # Update velocity
        r1, r2 = random.random(), random.random()
        for i in range(2):
            cognitive = c1 * r1 * (particle["best_pos"][i] - particle["position"][i])
            social = c2 * r2 * (global_best_pos[i] - particle["position"][i])
            particle["velocity"][i] = w * particle["velocity"][i] + cognitive + social

        # Update position
        particle["position"][0] += particle["velocity"][0]
        particle["position"][1] += particle["velocity"][1]

        # Evaluate new fitness
        fit = objective_function(particle["position"][0], particle["position"][1])

        # Update personal best
```

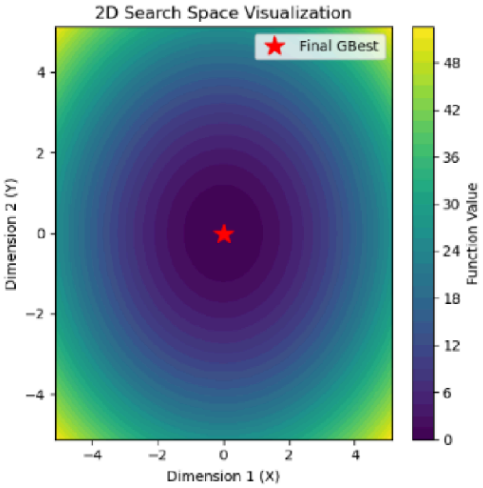
```
        if fit < particle["best_fit"]:
            particle["best_fit"] = fit
            particle["best_pos"] = particle["position"][:]

        # Update global best
        if fit < global_best_fit:
            global_best_fit = fit
            global_best_pos = particle["position"][:]

    print(f"Iteration {t+1}/{max_iterations} - Best Fitness: {global_best_fit:.6f}")

print("\n✅ Optimization complete!")
print(f"Best position found: {global_best_pos}")
print(f"Minimum value of function: {global_best_fit:.6f}")
```

Output:

```
Starting PSO for 2 dimensions with 10 particles...
Initial GBest Score: 4.0825
Iteration 1/100: GBest Score = 4.0825e+00
Iteration 2/100: GBest Score = 2.3789e+00
Iteration 3/100: GBest Score = 2.1673e+00
Iteration 4/100: GBest Score = 2.1673e+00
Iteration 5/100: GBest Score = 1.4520e+00
Iteration 6/100: GBest Score = 1.2043e+00
Iteration 7/100: GBest Score = 9.8974e-01
Iteration 8/100: GBest Score = 5.5153e-02
Iteration 9/100: GBest Score = 9.9545e-03
Iteration 10/100: GBest Score = 9.9545e-03
Iteration 11/100: GBest Score = 9.9545e-03
Iteration 12/100: GBest Score = 9.9545e-03
Iteration 13/100: GBest Score = 9.9545e-03
Iteration 14/100: GBest Score = 8.0772e-03
Iteration 15/100: GBest Score = 8.0772e-03
Iteration 16/100: GBest Score = 8.0772e-03
Iteration 17/100: GBest Score = 7.3521e-03
Iteration 18/100: GBest Score = 7.3521e-03
Iteration 19/100: GBest Score = 4.1808e-03
Iteration 20/100: GBest Score = 4.1808e-03
Iteration 21/100: GBest Score = 1.0079e-03
Iteration 22/100: GBest Score = 1.0079e-03
Iteration 23/100: GBest Score = 1.0079e-03
Iteration 24/100: GBest Score = 1.0079e-03
Iteration 25/100: GBest Score = 1.0079e-03
Iteration 26/100: GBest Score = 3.8768e-04
Iteration 27/100: GBest Score = 1.8362e-04
Iteration 28/100: GBest Score = 1.8362e-04
Iteration 29/100: GBest Score = 7.5330e-05
Iteration 30/100: GBest Score = 7.5330e-05
Iteration 31/100: GBest Score = 7.5330e-05
Iteration 32/100: GBest Score = 7.5330e-05
Iteration 33/100: GBest Score = 7.5330e-05
Iteration 34/100: GBest Score = 7.5330e-05
Iteration 35/100: GBest Score = 7.5330e-05
Iteration 36/100: GBest Score = 7.5330e-05
Iteration 37/100: GBest Score = 7.5330e-05
Iteration 38/100: GBest Score = 7.5330e-05
Iteration 39/100: GBest Score = 7.5330e-05
Iteration 40/100: GBest Score = 7.5330e-05
Iteration 41/100: GBest Score = 7.5330e-05
Iteration 42/100: GBest Score = 7.5330e-05
Iteration 43/100: GBest Score = 4.9330e-05
Iteration 44/100: GBest Score = 4.9330e-05
Iteration 45/100: GBest Score = 3.2129e-05
Iteration 46/100: GBest Score = 3.2129e-05
Iteration 47/100: GBest Score = 3.2129e-05
Iteration 48/100: GBest Score = 8.4985e-06
Iteration 49/100: GBest Score = 1.3416e-06
Iteration 50/100: GBest Score = 1.3416e-06
Iteration 51/100: GBest Score = 1.3416e-06
Iteration 52/100: GBest Score = 1.3416e-06
Iteration 53/100: GBest Score = 1.3416e-06
Iteration 54/100: GBest Score = 1.3416e-06
Iteration 55/100: GBest Score = 1.3416e-06
Iteration 56/100: GBest Score = 1.3416e-06
Iteration 57/100: GBest Score = 1.3416e-06
Iteration 58/100: GBest Score = 1.3416e-06
Iteration 59/100: GBest Score = 3.1802e-07
Iteration 60/100: GBest Score = 3.1802e-07
Iteration 61/100: GBest Score = 1.5823e-07
```

```
Iteration 72/100: GBest Score = 5.0642e-10
Iteration 73/100: GBest Score = 5.0642e-10
Iteration 74/100: GBest Score = 5.0642e-10
Iteration 75/100: GBest Score = 5.0642e-10
Iteration 76/100: GBest Score = 5.0642e-10
Iteration 77/100: GBest Score = 5.0642e-10
Iteration 78/100: GBest Score = 1.4724e-10
Iteration 79/100: GBest Score = 1.4724e-10
Iteration 80/100: GBest Score = 1.3409e-10
Iteration 81/100: GBest Score = 1.3409e-10
Iteration 82/100: GBest Score = 1.3409e-10
Iteration 83/100: GBest Score = 1.3409e-10
Iteration 84/100: GBest Score = 1.3409e-10
Iteration 85/100: GBest Score = 1.3409e-10
Iteration 86/100: GBest Score = 1.3409e-10
Iteration 87/100: GBest Score = 1.3409e-10
Iteration 88/100: GBest Score = 1.3409e-10
Iteration 89/100: GBest Score = 1.3409e-10
Iteration 90/100: GBest Score = 1.3409e-10
Iteration 91/100: GBest Score = 1.3409e-10
Iteration 92/100: GBest Score = 1.3409e-10
Iteration 93/100: GBest Score = 1.3409e-10
Iteration 94/100: GBest Score = 1.3409e-10
Iteration 95/100: GBest Score = 1.3409e-10
Iteration 96/100: GBest Score = 1.3409e-10
Iteration 97/100: GBest Score = 1.3409e-10
Iteration 98/100: GBest Score = 4.9838e-11
Iteration 99/100: GBest Score = 4.9838e-11
Iteration 100/100: GBest Score = 4.9838e-11

--- Results ---
Objective Function: Sphere Function
Best Position Found: [ 6.78638486e-06 -1.94488851e-06]
Minimum Score (Fitness): 4.983761e-11
```





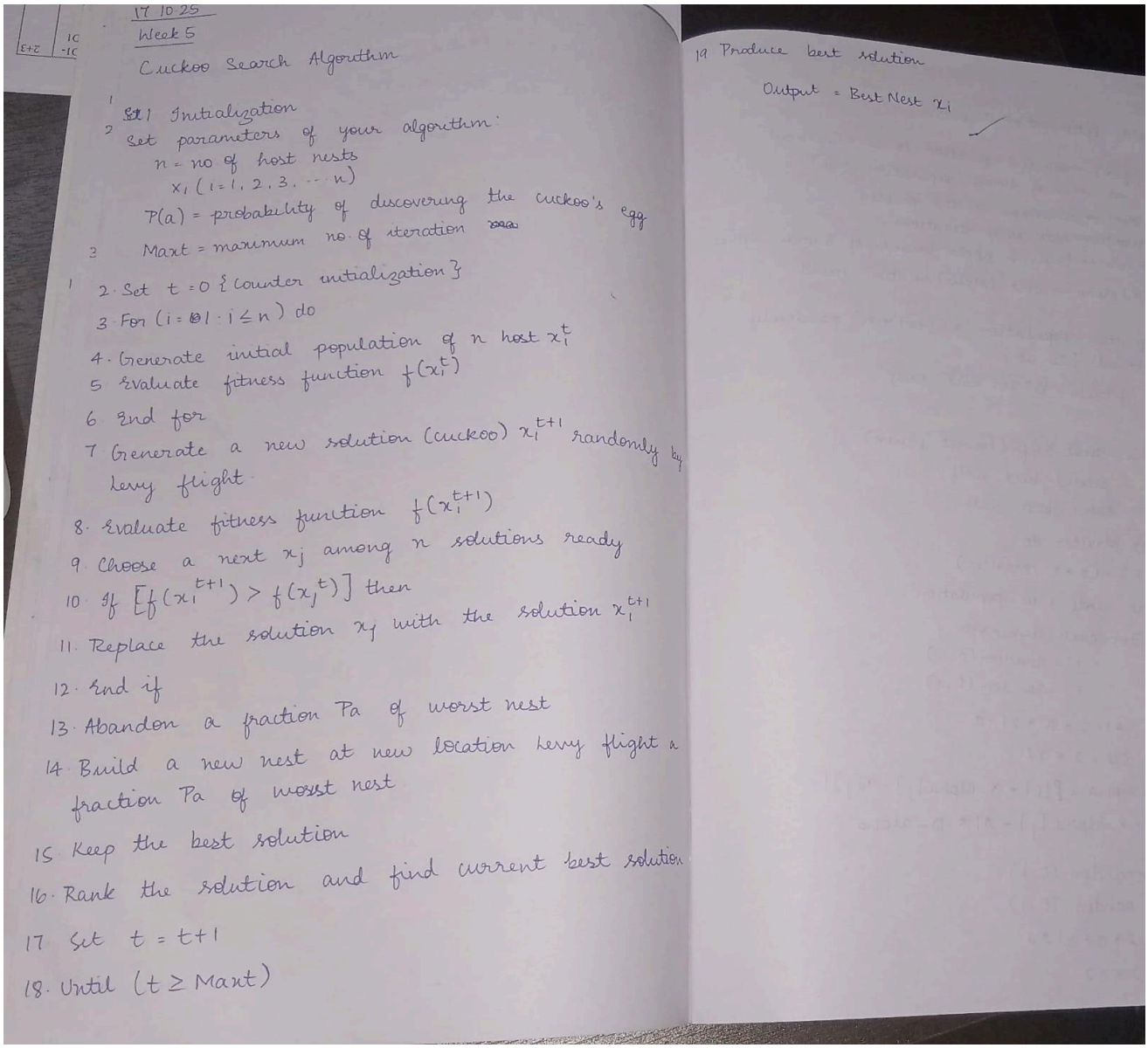
Program 5

Problem Statement:

Cuckoo Search (CS) for travel salesman problem :

Use the Cuckoo Search Algorithm to determine the shortest route for a salesman to visit all cities and return to the starting point.

Algorithm:



Code:

```
import random
import math

# ------------------------------
# Helper functions
# ------------------------------

def distance(city1, city2):
    """Euclidean distance between two cities"""
    return math.sqrt((city1[0] - city2[0])**2 + (city1[1] - city2[1])**2)

def total_distance(path, cities):
    """Compute total distance of the path"""
    dist = 0
    for i in range(len(path)):
        city1 = cities[path[i]]
        city2 = cities[path[(i + 1) % len(path)]]
        dist += distance(city1, city2)
    return dist

def levy_flight(Lambda):
    """Generate step size using Lévy distribution"""
    sigma = (math.gamma(1 + Lambda) * math.sin(math.pi * Lambda / 2) /
             (math.gamma((1 + Lambda) / 2) * Lambda * 2**((Lambda - 1) / 2)))**(1 / Lambda)
    u = random.gauss(0, sigma)
    v = random.gauss(0, 1)
    step = u / abs(v)**(1 / Lambda)
    return step

def get_new_solution(path):
    """Generate a new random permutation by swapping two cities"""
    new_path = path[:]
    i, j = random.sample(range(len(new_path)), 2)
    new_path[i], new_path[j] = new_path[j], new_path[i]
    return new_path

# ------------------------------
# Cuckoo Search Algorithm
# ------------------------------

def cuckoo_search_tsp(cities, n_nests=15, pa=0.25, alpha=1, Lambda=1.5, max_iter=100):
    # Initialize nests (random permutations)
    nests = [random.sample(range(len(cities)), len(cities)) for _ in range(n_nests)]
    fitness = [total_distance(nest, cities) for nest in nests]

    best_nest = nests[fitness.index(min(fitness))]
```

```
    best_fitness = min(fitness)

    for iteration in range(max_iter):
        for i in range(n_nests):
            # Lévy flight (small random move)
            step_size = levy_flight(Lambda)
            new_nest = get_new_solution(nests[i])

            new_fitness = total_distance(new_nest, cities)

            # If better, replace
            if new_fitness < fitness[i]:
                nests[i] = new_nest
                fitness[i] = new_fitness

            # Update best
            if new_fitness < best_fitness:
                best_nest = new_nest[:]
                best_fitness = new_fitness

        # Abandon some nests with probability pa
        for i in range(n_nests):
            if random.random() < pa:
                nests[i] = random.sample(range(len(cities)), len(cities))
                fitness[i] = total_distance(nests[i], cities)

        print(f"Iteration {iteration+1}/{max_iter} - Best Distance: {best_fitness:.4f}")

    return best_nest, best_fitness

# ------------------------------
# Example Run
# ------------------------------

if __name__ == "__main__":
    # Example: 10 cities with random coordinates
    cities = [(random.uniform(0, 100), random.uniform(0, 100)) for _ in range(10)]

    best_path, best_distance = cuckoo_search_tsp(cities, n_nests=20, max_iter=100)

    print("\n✅ Best Path Found:")
    print(best_path)
    print(f"Total Distance: {best_distance:.4f}")
```

Output:

```
print("\n✅ Best Path Found:")
print(best_path)
print(f"Total Distance: {best_distance:.4f}")
```

```
Iteration 1/100 - Best Distance: 363.5972
Iteration 2/100 - Best Distance: 363.5972
Iteration 3/100 - Best Distance: 363.5972
Iteration 4/100 - Best Distance: 363.5972
Iteration 5/100 - Best Distance: 363.5972
Iteration 6/100 - Best Distance: 348.3731
Iteration 7/100 - Best Distance: 348.3731
Iteration 8/100 - Best Distance: 348.3731
Iteration 9/100 - Best Distance: 348.3731
Iteration 10/100 - Best Distance: 348.3731
Iteration 11/100 - Best Distance: 348.3731
Iteration 12/100 - Best Distance: 348.3731
Iteration 13/100 - Best Distance: 348.3731
Iteration 14/100 - Best Distance: 327.6580
Iteration 15/100 - Best Distance: 327.6580
Iteration 16/100 - Best Distance: 327.6580
Iteration 17/100 - Best Distance: 327.6580
Iteration 18/100 - Best Distance: 327.6580
Iteration 19/100 - Best Distance: 327.6580
Iteration 20/100 - Best Distance: 327.6580
Iteration 21/100 - Best Distance: 327.6580
Iteration 22/100 - Best Distance: 327.6580
Iteration 23/100 - Best Distance: 327.6580
Iteration 24/100 - Best Distance: 327.6580
Iteration 25/100 - Best Distance: 327.6580
Iteration 26/100 - Best Distance: 327.6580
Iteration 27/100 - Best Distance: 327.6580
Iteration 28/100 - Best Distance: 327.6580
Iteration 29/100 - Best Distance: 327.6580
Iteration 30/100 - Best Distance: 327.6580
Iteration 31/100 - Best Distance: 327.6580
Iteration 32/100 - Best Distance: 327.6580
Iteration 33/100 - Best Distance: 327.6580
Iteration 34/100 - Best Distance: 327.6580
Iteration 35/100 - Best Distance: 327.6580
Iteration 36/100 - Best Distance: 327.6580
Iteration 37/100 - Best Distance: 327.6580
Iteration 38/100 - Best Distance: 327.6580
Iteration 39/100 - Best Distance: 327.6580
Iteration 40/100 - Best Distance: 327.6580
Iteration 41/100 - Best Distance: 327.6580
Iteration 42/100 - Best Distance: 327.6580
Iteration 43/100 - Best Distance: 327.6580
Iteration 44/100 - Best Distance: 327.6580
Iteration 45/100 - Best Distance: 327.6580
```

```
Iteration 43/100 - Best Distance: 327.6580
Iteration 44/100 - Best Distance: 327.6580
Iteration 45/100 - Best Distance: 327.6580
Iteration 46/100 - Best Distance: 325.8042
Iteration 47/100 - Best Distance: 320.2649
Iteration 48/100 - Best Distance: 320.2649
Iteration 49/100 - Best Distance: 320.2649
Iteration 50/100 - Best Distance: 320.2649
Iteration 51/100 - Best Distance: 320.2649
Iteration 52/100 - Best Distance: 320.2649
Iteration 53/100 - Best Distance: 320.2649
Iteration 54/100 - Best Distance: 320.2649
Iteration 55/100 - Best Distance: 320.2649
Iteration 56/100 - Best Distance: 320.2649
Iteration 57/100 - Best Distance: 320.2649
Iteration 58/100 - Best Distance: 320.2649
Iteration 59/100 - Best Distance: 320.2649
Iteration 60/100 - Best Distance: 320.2649
Iteration 61/100 - Best Distance: 320.2649
Iteration 62/100 - Best Distance: 320.2649
Iteration 63/100 - Best Distance: 320.2649
Iteration 64/100 - Best Distance: 320.2649
Iteration 65/100 - Best Distance: 320.2649
Iteration 66/100 - Best Distance: 320.2649
Iteration 67/100 - Best Distance: 320.2649
Iteration 68/100 - Best Distance: 320.2649
Iteration 69/100 - Best Distance: 320.2649
Iteration 70/100 - Best Distance: 320.2649
Iteration 71/100 - Best Distance: 320.2649
Iteration 72/100 - Best Distance: 320.2649
Iteration 73/100 - Best Distance: 320.2649
Iteration 74/100 - Best Distance: 320.2649
Iteration 75/100 - Best Distance: 320.2649
Iteration 76/100 - Best Distance: 320.2649
Iteration 77/100 - Best Distance: 320.2649
Iteration 78/100 - Best Distance: 320.2649
Iteration 79/100 - Best Distance: 320.2649
Iteration 80/100 - Best Distance: 320.2649
Iteration 81/100 - Best Distance: 320.2649
Iteration 82/100 - Best Distance: 320.2649
Iteration 83/100 - Best Distance: 320.2649
Iteration 84/100 - Best Distance: 320.2649
Iteration 85/100 - Best Distance: 320.2649
Iteration 86/100 - Best Distance: 320.2649
Iteration 87/100 - Best Distance: 320.2649
Iteration 88/100 - Best Distance: 320.2649
Iteration 89/100 - Best Distance: 320.2649
Iteration 90/100 - Best Distance: 320.2649
Iteration 91/100 - Best Distance: 320.2649
```

```
Iteration 61/100 - Best Distance: 320.2649
Iteration 62/100 - Best Distance: 320.2649
Iteration 63/100 - Best Distance: 320.2649
Iteration 64/100 - Best Distance: 320.2649
Iteration 65/100 - Best Distance: 320.2649
Iteration 66/100 - Best Distance: 320.2649
Iteration 67/100 - Best Distance: 320.2649
Iteration 68/100 - Best Distance: 320.2649
Iteration 69/100 - Best Distance: 320.2649
Iteration 70/100 - Best Distance: 320.2649
Iteration 71/100 - Best Distance: 320.2649
Iteration 72/100 - Best Distance: 320.2649
Iteration 73/100 - Best Distance: 320.2649
Iteration 74/100 - Best Distance: 320.2649
Iteration 75/100 - Best Distance: 320.2649
Iteration 76/100 - Best Distance: 320.2649
Iteration 77/100 - Best Distance: 320.2649
Iteration 78/100 - Best Distance: 320.2649
Iteration 79/100 - Best Distance: 320.2649
Iteration 80/100 - Best Distance: 320.2649
Iteration 81/100 - Best Distance: 320.2649
Iteration 82/100 - Best Distance: 320.2649
Iteration 83/100 - Best Distance: 320.2649
Iteration 84/100 - Best Distance: 320.2649
Iteration 85/100 - Best Distance: 320.2649
Iteration 86/100 - Best Distance: 320.2649
Iteration 87/100 - Best Distance: 320.2649
Iteration 88/100 - Best Distance: 320.2649
Iteration 89/100 - Best Distance: 320.2649
Iteration 90/100 - Best Distance: 320.2649
Iteration 91/100 - Best Distance: 320.2649
Iteration 92/100 - Best Distance: 320.2649
Iteration 93/100 - Best Distance: 320.2649
Iteration 94/100 - Best Distance: 320.2649
Iteration 95/100 - Best Distance: 320.2649
Iteration 96/100 - Best Distance: 320.2649
Iteration 97/100 - Best Distance: 320.2649
Iteration 98/100 - Best Distance: 320.2649
Iteration 99/100 - Best Distance: 320.2649
Iteration 100/100 - Best Distance: 320.2649

✅ Best Path Found:
[1, 8, 3, 5, 4, 0, 6, 9, 7, 2]
Total Distance: 320.2649
```

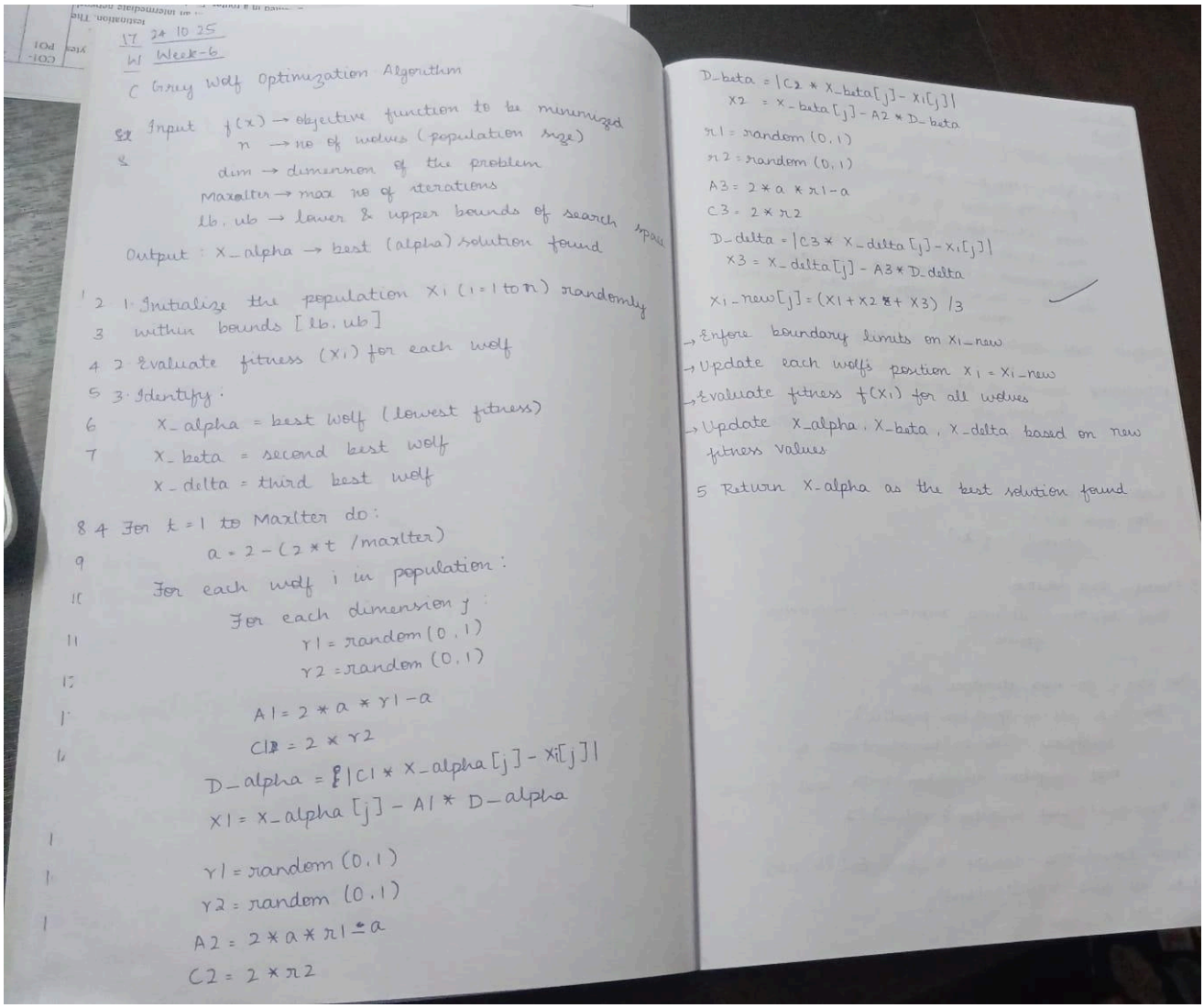
Program 6

Problem Statement:

Grey Wolf Optimizer (GWO) for Function Optimization:

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

Algorithm:



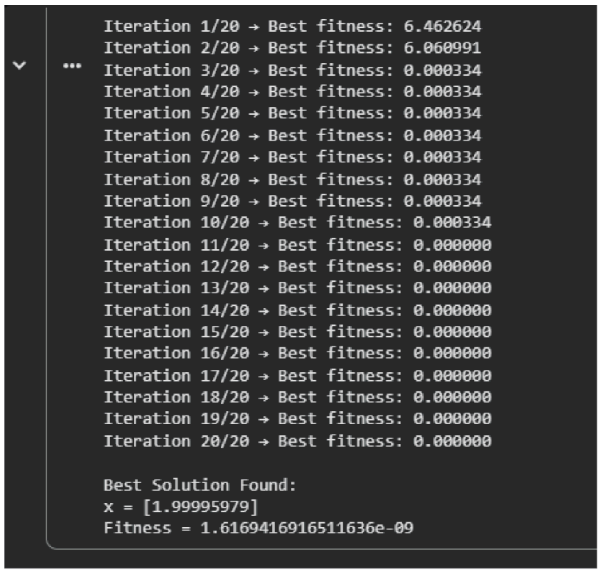
Code:

```
import random
import math
```

```
# ----------------------------
# Objective Function (to minimize)
# ----------------------------
def objective_function(position):
    # Sphere function: sum of squares
    return sum(x**2 for x in position)

# ----------------------------
# Grey Wolf Optimizer
# ----------------------------
def grey_wolf_optimizer(num_wolves=20, dim=2, max_iter=100, lower_bound=-10,
upper_bound=10):
    # Initialize the population of wolves randomly
    wolves = [[random.uniform(lower_bound, upper_bound) for _ in range(dim)] for _ in
range(num_wolves)]
    fitness = [objective_function(w) for w in wolves]

    # Initialize alpha, beta, delta (best 3 wolves)
    alpha, beta, delta = [0]*dim, [0]*dim, [0]*dim
    alpha_score, beta_score, delta_score = float("inf"), float("inf"), float("inf")

    # Main loop
    for t in range(max_iter):
        for i in range(num_wolves):
            score = fitness[i]

            # Update alpha, beta, delta
            if score < alpha_score:
                delta_score, delta = beta_score, beta[:]
                beta_score, beta = alpha_score, alpha[:]
                alpha_score, alpha = score, wolves[i][:]
            elif score < beta_score:
                delta_score, delta = beta_score, beta[:]
                beta_score, beta = score, wolves[i][:]
            elif score < delta_score:
                delta_score, delta = score, wolves[i][:]

        # Linearly decreasing a from 2 to 0
        a = 2 - t * (2 / max_iter)

        # Update each wolf's position
        for i in range(num_wolves):
            new_position = []
            for j in range(dim):
                r1, r2 = random.random(), random.random()
                A1 = 2 * a * r1 - a
                C1 = 2 * r2
```

```
            D_alpha = abs(C1 * alpha[j] - wolves[i][j])
            X1 = alpha[j] - A1 * D_alpha

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

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

            new_x = (X1 + X2 + X3) / 3
            # Keep within bounds
            new_x = max(lower_bound, min(upper_bound, new_x))
            new_position.append(new_x)

        wolves[i] = new_position
        fitness[i] = objective_function(new_position)

    print(f"Iteration {t+1}/{max_iter} - Best fitness: {alpha_score:.6f}")

  return alpha, alpha_score

# ----------------------------
# Example Run
# ----------------------------
if __name__ == "__main__":
  best_position, best_score = grey_wolf_optimizer(num_wolves=20, dim=5, max_iter=100)
  print("\n✅ Optimization complete!")
  print(f"Best position found: {best_position}")
  print(f"Minimum value of function: {best_score:.6f}")
```

Output:


Iteration 1/20 → Best fitness: 6.462624
Iteration 2/20 → Best fitness: 6.060991
Iteration 3/20 → Best fitness: 0.000334
Iteration 4/20 → Best fitness: 0.000334
Iteration 5/20 → Best fitness: 0.000334
Iteration 6/20 → Best fitness: 0.000334
Iteration 7/20 → Best fitness: 0.000334
Iteration 8/20 → Best fitness: 0.000334
Iteration 9/20 → Best fitness: 0.000334
Iteration 10/20 → Best fitness: 0.000334
Iteration 11/20 → Best fitness: 0.000000
Iteration 12/20 → Best fitness: 0.000000
Iteration 13/20 → Best fitness: 0.000000
Iteration 14/20 → Best fitness: 0.000000
Iteration 15/20 → Best fitness: 0.000000
Iteration 16/20 → Best fitness: 0.000000
Iteration 17/20 → Best fitness: 0.000000
Iteration 18/20 → Best fitness: 0.000000
Iteration 19/20 → Best fitness: 0.000000
Iteration 20/20 → Best fitness: 0.000000
Best Solution Found:
x = [1.99995979]
Fitness = 1.6169416916511636e-09

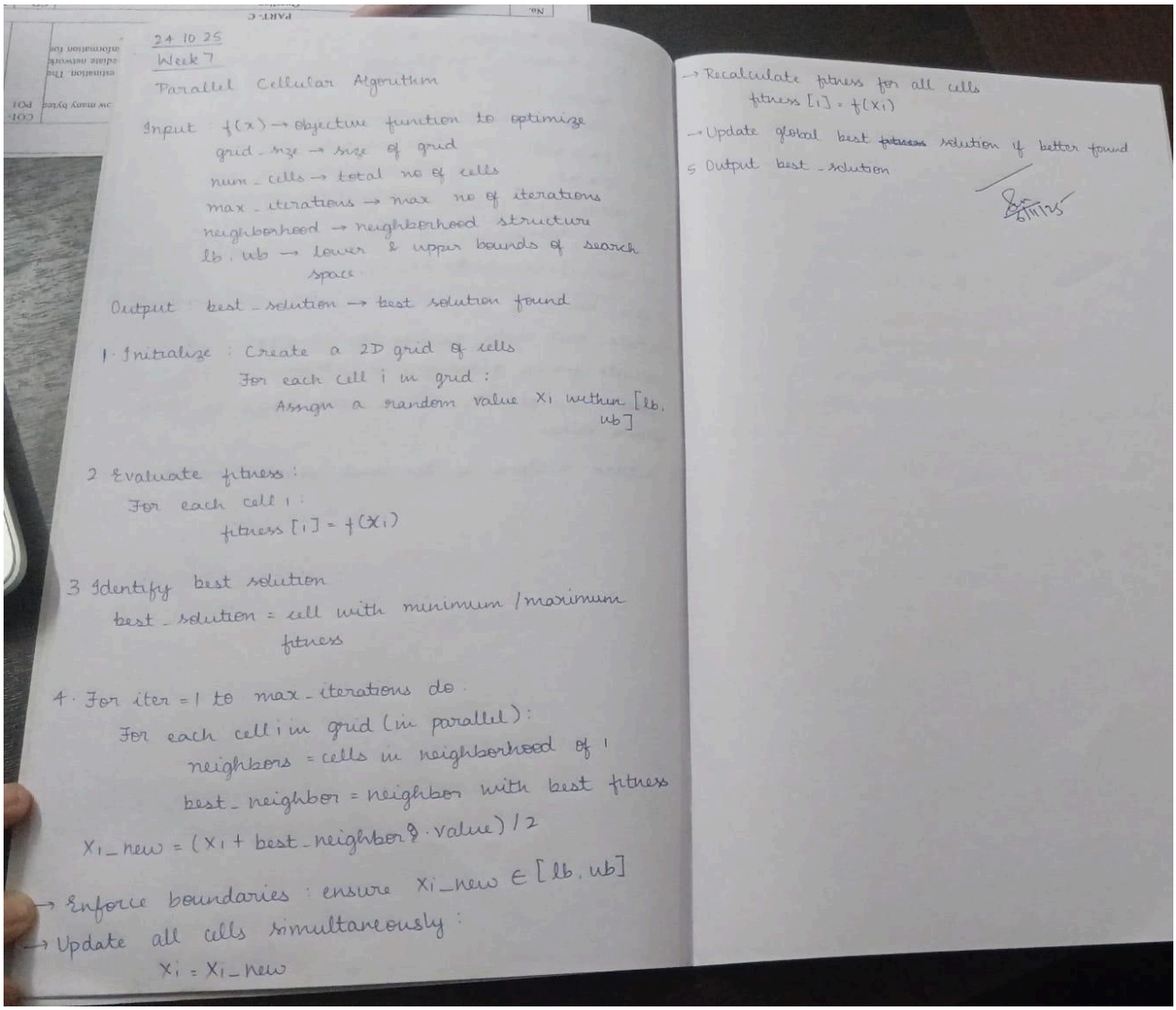
Program 7

Problem Statement:

Parallel Cellular Algorithm for Function Optimization:

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

Algorithm:



Code:

import numpy as np

```
def parallel_cellular_algorithm(f, grid_size=(10, 10), max_iterations=100, lb=-10, ub=10):
    rows, cols = grid_size
    num_cells = rows * cols

    # 1. Initialize grid with random values
    X = np.random.uniform(lb, ub, size=(rows, cols))

    # Evaluate fitness
    fitness = f(X)

    # Track best global solution
    best_value = np.min(fitness)
    best_pos = np.unravel_index(np.argmin(fitness), fitness.shape)
    best_solution = X[best_pos]

    # Neighborhood offsets for 3x3 neighborhood
    neighbors_offset = [(-1,-1), (-1,0), (-1,1),
                        (0,-1),  (0,0),  (0,1),
                        (1,-1),  (1,0),  (1,1)]

    for iteration in range(max_iterations):
        X_new = np.copy(X)

        # 4. Update each cell in parallel logic
        for i in range(rows):
            for j in range(cols):
                # Collect neighbors
                neighbor_values = []
                for dx, dy in neighbors_offset:
                    ni, nj = i + dx, j + dy
                    if 0 <= ni < rows and 0 <= nj < cols:
                        neighbor_values.append((X[ni, nj], fitness[ni, nj]))

                # Get best neighbor (minimum fitness)
                best_neighbor_value, _ = min(neighbor_values, key=lambda x: x[1])

                # Update rule: average
                X_new[i, j] = (X[i, j] + best_neighbor_value) / 2

        # Boundary enforcement
        X_new = np.clip(X_new, lb, ub)

        # Replace old values
        X = X_new

        # Recalculate fitness
        fitness = f(X)
```

```
        # Update global best
        current_best = np.min(fitness)
        if current_best < best_value:
            best_value = current_best
            best_pos = np.unravel_index(np.argmin(fitness), fitness.shape)
            best_solution = X[best_pos]

    return best_solution, best_value


# ------------------------------------------------------------
# Example: Optimize Sphere Function f(x) = x^2
# ------------------------------------------------------------

def sphere_function(x):
    return x**2  # works element-wise

best_x, best_val = parallel_cellular_algorithm(
    f=sphere_function,
    grid_size=(10, 10),
    max_iterations=100,
    lb=-5,
    ub=5
)

print("Best solution:", best_x)
print("Best function value:", best_val)
```

Output:

```
[4]
✓ 0s
            new_cells = np.clip(new_cells, lb, ub)
            cells = new_cells
            fitness = obj_func(cells)
            current_best = np.min(fitness)
            if current_best < best_value:
                best_value = current_best
                best_position = np.unravel_index(np.argmin(fitness), fitness.shape)
            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
    best_x, best_fit = parallel_cellular_algorithm(
        objective_function,
        grid_size=(10, 10),
        lb=-10,
        ub=10,
        max_iter=60
    )
```

```
    )

...  Iteration 1/60 → Best fitness: 0.001454
     Iteration 10/60 → Best fitness: 0.000000
     Iteration 20/60 → Best fitness: 0.000000
     Iteration 30/60 → Best fitness: 0.000000
     Iteration 40/60 → Best fitness: 0.000000
     Iteration 50/60 → Best fitness: 0.000000
     Iteration 60/60 → Best fitness: 0.000000

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