VISVESVARAYA TECHNOLOGICAL UNIVERSITY

"JnanaSangama", Belgaum -590014, Karnataka.

LAB RECORD

Bio Inspired Systems (23CS5BSBIS)

Submitted by

Shashank U (1BM23CS314)

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 **Shashank U (1BM23CS314),** who is bonafide student of **B.M.S. College of Engineering.** It is in partial fulfillment for the award of **Bachelor of Engineering in Computer Science and Engineering** of the Visvesvaraya Technological University, Belgaum. The Lab report has been approved as it satisfies the academic requirements of the above mentioned subject and the work prescribed for the said degree.

Prof. Mayanka Gupta Assistant Professor Department of CSE, BMSCE	Dr. Kavitha Sooda Professor & HOD Department of CSE, BMSCE

Index

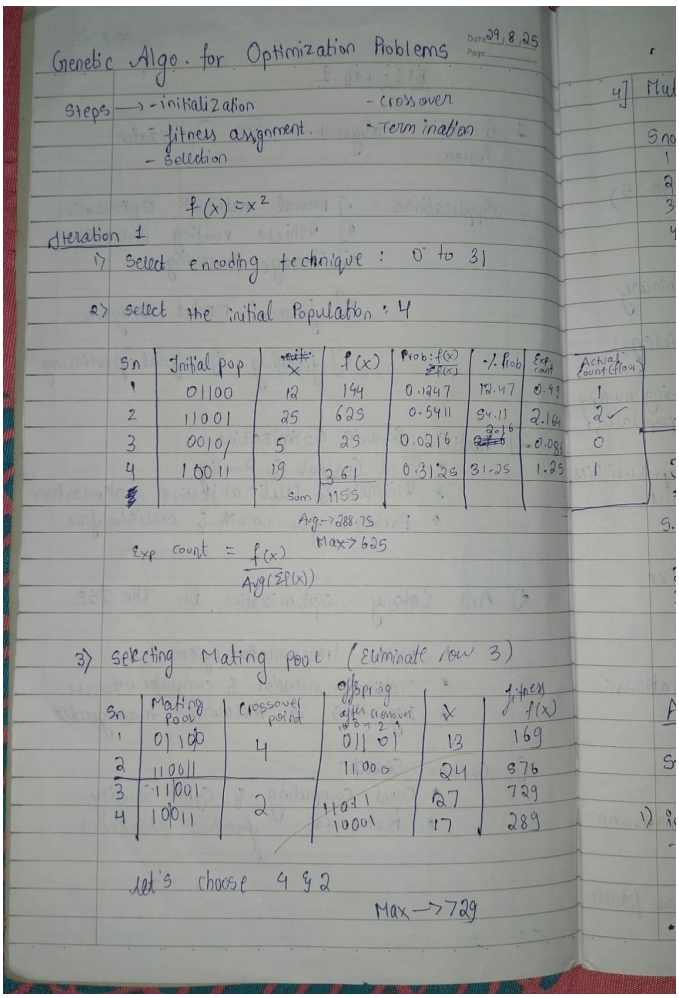
Github Link: https://github.com/Shashank-u803/BIS-Lab

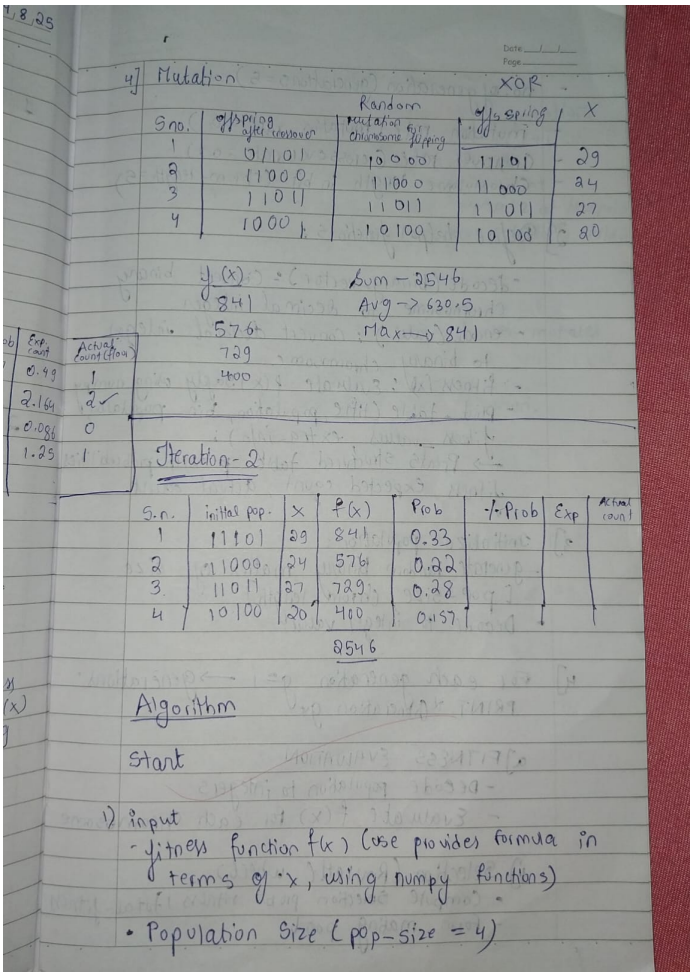
Program 1

Genetic Algorithm for Optimization Problems

We have a set of jobs that must be completed and a limited amount of resources available to perform them. The challenge is to determine how to assign each job to the available resources in a way that minimizes total completion time, reduces overall cost, or maximizes efficiency. The goal is to find an optimal scheduling strategy under these constraints.

Algorithm:





- no. of generation (generations = 5)
- mutation rate (mutation_rate = 0.1)
- Crossover rate (crossover_rate = 0.7)
- Chromosome length in bit (chrom_length = 5)

2] Define helper functions:

- decode (binary_vector): Convert binary chromosome to decimal integer
- encode (value): Convert decimal integer to binary chromosome.
- fitness (x): Evaluate f(x) safely using numpy
- print_table (title, population, bin_population, fitness_values, extra_info):
  → Prints structured table with probabilities, fitness, expected count, actual count.

3] Initialize population:

- generate random binary matrix of size [pop_size, chrom_length]
- Decode to integer value

4] For each generation g = 1 → generations:
PRINT "Generation g"

a] FITNESS EVALUATION
- Decode population to integers
- Evaluate f(x) for each chromosome

b] Selection (Roulette wheel)
- Compute selection prob = fitness / total_fitness
- Form mating pool

c] Crossover
- initialize empty offspring population
- for every pair of parents (i, i+1):
  if (random < crossover_rate:
  Select random crossover point
  Split parent1 and parent2 at that point
  Swap tails to produce child1 & child2

d] Update
- replace old population with mutated offspring
- Decode again for next iteration.

END LOOP.

Output:
- after all generations, show last population table.
- best solution can be identified as chromosome with max fitness

End

Code:

```
import random

jobs = [3, 2, 7, 5, 9, 4]  # processing times of jobs
num_jobs = len(jobs)
population_size = 20
generations = 100
crossover_rate = 0.8
mutation_rate = 0.2

# ---------------------------------
# Fitness Function (Makespan)
# ---------------------------------
def fitness(chromosome):
    time = 0
    for job in chromosome:
        time += jobs[job]
    return 1 / time   # smaller time → higher fitness

def initial_population():
    population = []
    for _ in range(population_size):
        chromosome = list(range(num_jobs))
        random.shuffle(chromosome)
        population.append(chromosome)
    return population
def selection(population):
    contenders = random.sample(population, 3)
    contenders.sort(key=lambda chromo: fitness(chromo), reverse=True)
    return contenders[0]
def crossover(p1, p2):
    if random.random() < crossover_rate:
        a, b = sorted(random.sample(range(num_jobs), 2))
        child = [-1] * num_jobs
        child[a:b] = p1[a:b]
        fill = [x for x in p2 if x not in child]
        j = 0
        for i in range(num_jobs):
            if child[i] == -1:
                child[i] = fill[j]
                j += 1
        return child
    return p1[:]  # no crossover → copy parent
def mutate(chromosome):
    if random.random() < mutation_rate:
        a, b = random.sample(range(num_jobs), 2)
        chromosome[a], chromosome[b] = chromosome[b], chromosome[a]
```

```
    return chromosome
population = initial_population()
best_solution = None
best_fit = -1

for gen in range(generations):
    new_pop = []
    for _ in range(population_size):
        parent1 = selection(population)
        parent2 = selection(population)
        child = crossover(parent1, parent2)
        child = mutate(child)
        new_pop.append(child)

    population = new_pop

    # Track best
    for chromo in population:
        fit = fitness(chromo)
        if fit > best_fit:
            best_fit = fit
            best_solution = chromo
print("Best Job Order:", best_solution)
print("Job Times:", [jobs[j] for j in best_solution])
print("Total Completion Time (Makespan):", sum(jobs[j] for j in best_solution))
```

Output:

```
Enter a fitness function in terms of integer x (1 to 10)
Example: x * x or 10 - abs(x - 5)
Fitness function f(x) = x**2

=== Generation 1 ===
  Int |   Binary |     f(x) |     Prob
----------------------------------------
    3 |    00011 |   9.0000 |   0.0796
    2 |    00010 |   4.0000 |   0.0354
    8 |    01000 |  64.0000 |   0.5664
    6 |    00110 |  36.0000 |   0.3186

=== Generation 2 ===
  Int |   Binary |     f(x) |     Prob
----------------------------------------
    9 |    01001 |  81.0000 |   0.2755
    8 |    01000 |  64.0000 |   0.2177
   10 |    01010 | 100.0000 |   0.3401
    7 |    00111 |  49.0000 |   0.1667

=== Generation 3 ===
  Int |   Binary |     f(x) |     Prob
----------------------------------------
   10 |    01010 | 100.0000 |   0.2762
   10 |    01010 | 100.0000 |   0.2762
    9 |    01001 |  81.0000 |   0.2238
    9 |    01001 |  81.0000 |   0.2238
```

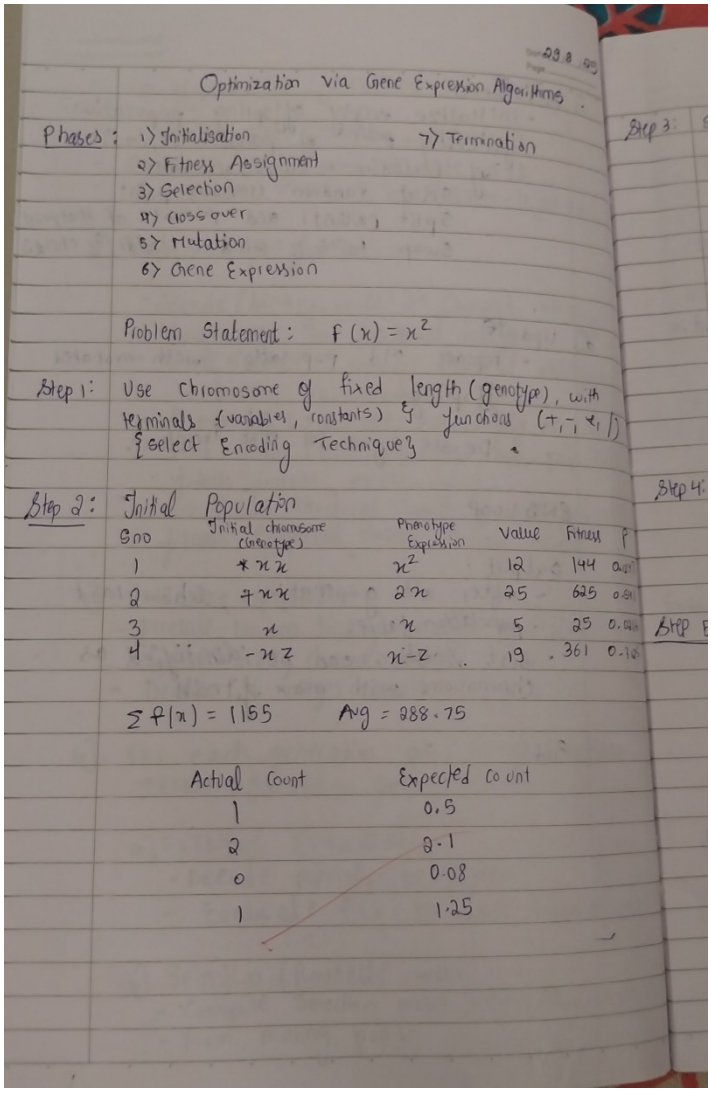
```
=== Generation 4 ===
  Int |   Binary |     f(x) |     Prob
----------------------------------------
   10 |    01010 | 100.0000 |   0.2625
    9 |    01001 |  81.0000 |   0.2126
   10 |    01010 | 100.0000 |   0.2625
   10 |    01010 | 100.0000 |   0.2625

=== Generation 5 ===
  Int |   Binary |     f(x) |     Prob
----------------------------------------
   10 |    01010 | 100.0000 |   0.2500
   10 |    01010 | 100.0000 |   0.2500
   10 |    01010 | 100.0000 |   0.2500
   10 |    01010 | 100.0000 |   0.2500

=== Final Generation ===
  Int |   Binary |     f(x)
------------------------------
   10 |    01010 | 100.0000
   10 |    01010 | 100.0000
   10 |    01010 | 100.0000
   10 |    01010 | 100.0000

Best Individual: x = 10, f(x) = 100.0000
```

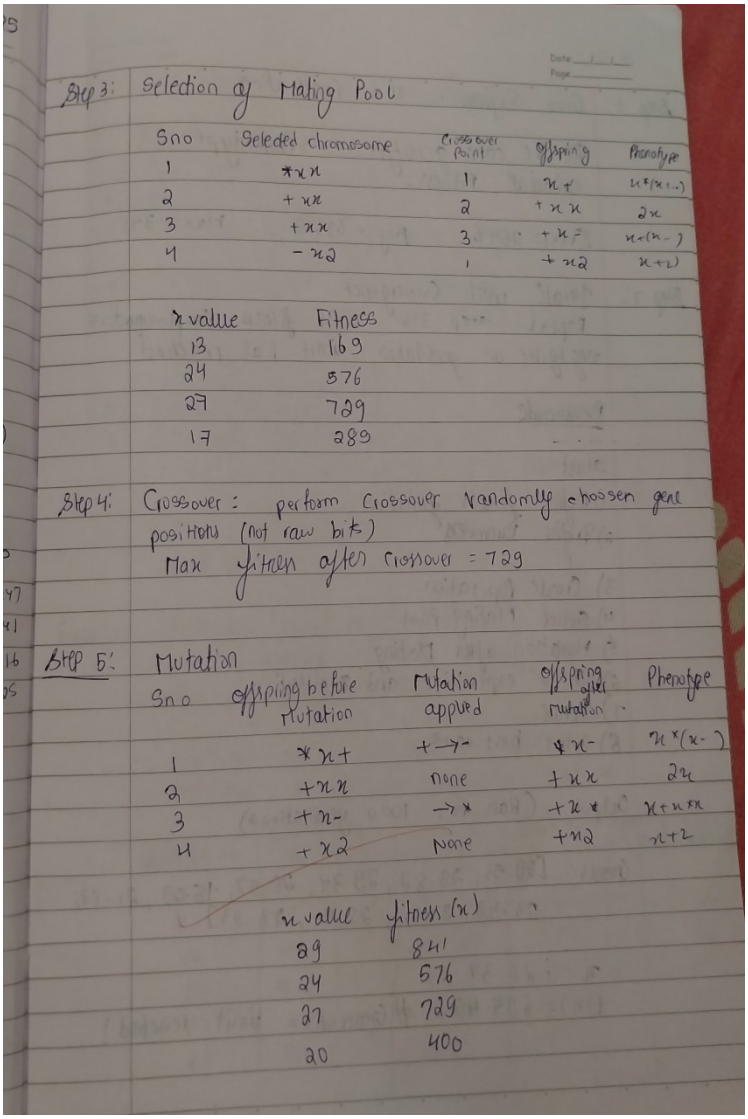
Program 2

Optimization via Gene Expression Algorithms

The Travelling Salesman Problem (TSP) asks for the shortest possible route that visits a given set of cities exactly once and returns to the starting city. The provided text describes using a Genetic Algorithm to solve this by evolving city sequences (chromosomes) through selection, crossover, and mutation to minimize the total tour distance.

Algorithm:





Step 6: Gene Expression and Evaluation

Decode each genotype ⟶ Phenotype
Calculate Fitness.

$\Sigma F(x) = 3546$ Avg = 636.5 Max = 341

Step 7: Iterate until Convergence
Repeat step 3-6 until fitness improvement is negligible or generation limit has reached

Pseudocode

Start
1) Define fitness function
2) Define Parameters
3) Create Population
4) Select Mating Pool
5) Mutation after Mating
6) Gene expression and Evaluation
7) Iterate
8) output best value

Output: (Ran for 1000 generations)

Genes: [29.53, 29.82, 29.34, 28.57, 15.09, 21.83, 23.53, 30.81, 28.51, 26.23]

x : 26.37
$f(x) = 695.45$ # Generation limit reached!

Code:

```
import random
import math

# --------------------------------
# Problem: TSP cities
# --------------------------------
cities = [(0,0), (1,5), (5,2), (6,6), (8,3)]  # coordinates
num_cities = len(cities)

# Parameters
population_size = 30
generations = 200
crossover_rate = 0.8
mutation_rate = 0.2

# --------------------------------
# Distance Function
# --------------------------------
def distance(a, b):
    return math.sqrt((a[0]-b[0])**2 + (a[1]-b[1])**2)

def tour_length(chromosome):
    length = 0
    for i in range(num_cities):
        length += distance(cities[chromosome[i]], cities[chromosome[(i+1)%num_cities]]
```

```
return length

    # --------------------------------
    # Fitness Function
    # --------------------------------
    def fitness(chromosome):
        return 1 / tour_length(chromosome)

    def initial_population():
        population = []
        for _ in range(population_size):
            chromosome = list(range(num_cities))
            random.shuffle(chromosome)
            population.append(chromosome)
        return population

    def selection(population):
        contenders = random.sample(population, 3)
        contenders.sort(key=lambda c: fitness(c), reverse=True)
        return contenders[0]

    def crossover(p1, p2):
        if random.random() < crossover_rate:
            a, b = sorted(random.sample(range(num_cities), 2))
            child = [-1]*num_cities
            child[a:b] = p1[a:b]
            fill = [x for x in p2 if x not in child]
            j = 0
            for i in range(num_cities):
                if child[i] == -1:
                    child[i] = fill[j]
                    j += 1
            return child
        return p1[:]

    def mutate(chromosome):
        if random.random() < mutation_rate:
            a, b = random.sample(range(num_cities), 2)
            chromosome[a], chromosome[b] = chromosome[b], chromosome[a]
        return chromosome

    population = initial_population()
    best_solution = None
    best_distance = float("inf")

    for g in range(generations):
        new_pop = []
        for _ in range(population_size):
            parent1 = selection(population)
```

```
        parent2 = selection(population)
        child = crossover(parent1, parent2)
        child = mutate(child)
        new_pop.append(child)

    population = new_pop

    # Track best solution
    for chromo in population:
        d = tour_length(chromo)
        if d < best_distance:
            best_distance = d
            best_solution = chromo
print("Best Tour (order of cities):", best_solution)
print("Best Tour Distance:", best_distance)
```

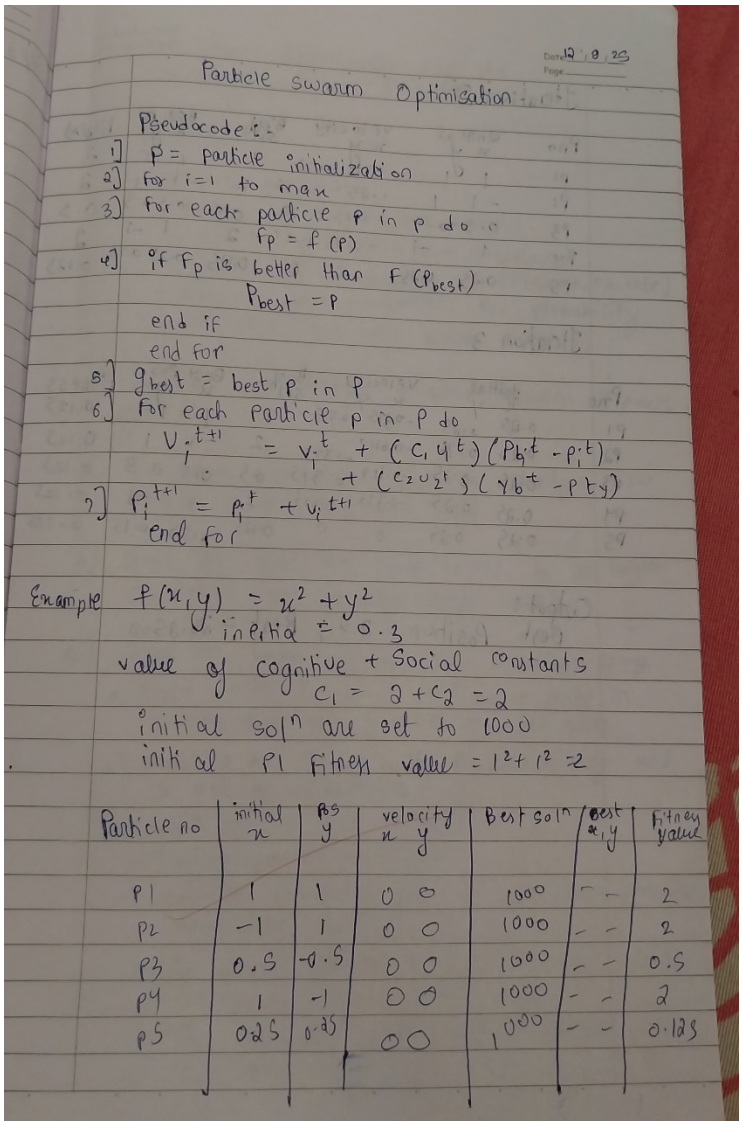
Output:

```
Gen   Best x     Best f(x)
------------------------------
1     0.99707    0.98833
2     0.99707    0.98833
3     0.99985    0.99939
4     0.99985    0.99939
5     0.99985    0.99939
6     0.99985    0.99939
7     0.99985    0.99939
8     0.99985    0.99939
9     0.99985    0.99939
10    0.99985    0.99939
11    0.99985    0.99939
12    0.99985    0.99939
13    0.99985    0.99939
14    0.99985    0.99939
15    0.99985    0.99939
16    0.99985    0.99939
17    0.99985    0.99939
18    0.99985    0.99939
19    0.99985    0.99939
20    0.99985    0.99939
21    0.99985    0.99939
22    0.99985    0.99939
23    0.99985    0.99939
24    0.99985    0.99939
25    0.99985    0.99939
26    0.99985    0.99939
27    0.99985    0.99939
28    0.99985    0.99939
29    0.99985    0.99939
30    0.99985    0.99939
31    0.99985    0.99939
32    0.99985    0.99939
33    0.99985    0.99939
34    0.99985    0.99939
35    0.99985    0.99939
36    0.99985    0.99939
37    0.99985    0.99939
38    0.99985    0.99939
39    0.99985    0.99939
40    0.99985    0.99939
41    0.99985    0.99939
42    0.99985    0.99939
43    0.99985    0.99939
44    0.99985    0.99939
45    0.99985    0.99939
46    0.99985    0.99939
47    0.99985    0.99939
48    0.99985    0.99939
49    0.99985    0.99939
50    0.99985    0.99939

Best Solution Found:
x = 0.99985, f(x) = 0.99939
```

Program 3

Particle Swarm Optimization for Function Optimization
Portfolio Optimization (Selecting assets) using Particle Swarm Optimization is about choosing how much money to allocate to different assets (stocks, bonds, etc.) to maximize expected return while minimizing risk (variance).

Algorithm:



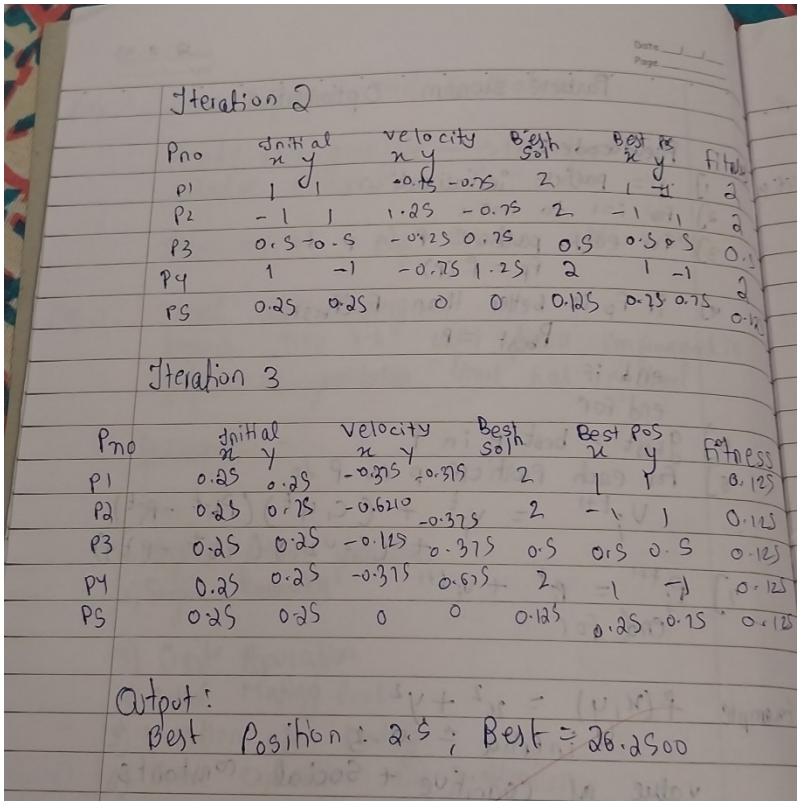
Particle swarm Optimisation

Pseudocode :-

1] P = particle initialization
2] for i=1 to max
3] for each particle p in P do
fp = f(P)
4] if fp is better than f(Pbest)
Pbest = P
end if
end for
5] gbest = best P in P
6] for each particle p in P do
$V_i^{t+1} = V_i^t + (C_1 y^t)(P_b^t - P_i^t) + (C_2 z^t)(Y_b^t - P_y^t)$
7] $P_i^{t+1} = P_i^t + V_i^{t+1}$
end for

Example $f(x,y) = x^2 + y^2$
inertia = 0.3
value of cognitive + social constants
$C_1 = 2 + C_2 = 2$
initial soln are set to 1000
initial P1 fitness value $= 1^2 + 1^2 = 2$

Particle no	initial x	initial y	velocity x	velocity y	Best soln	Best x,y	Fitness value
P1	1	1	0	0	1000	- -	2
P2	-1	1	0	0	1000	- -	2
P3	0.5	-0.5	0	0	1000	- -	0.5
P4	1	-1	0	0	1000	- -	2
P5	0.25	0.25	0	0	1000	- -	0.125



Iteration 2

Pno	initial x	initial y	velocity x	velocity y	Best soln	Best Pos x	Best Pos y	Fitness
P1	1	1	-0.75	-0.75	2	1	1	2
P2	-1	1	1.25	-0.75	2	-1	1	2
P3	0.5	-0.5	-0.25	0.75	0.5	0.5	0.5	0.5
P4	1	-1	-0.75	1.25	2	1	-1	2
P5	0.25	0.25	0	0	0.125	0.25	0.25	0.125

Iteration 3

Pno	initial x	initial y	velocity x	velocity y	Best soln	Best Pos x	Best Pos y	Fitness
P1	0.25	0.25	-0.375	-0.375	2	1	1	0.125
P2	0.25	0.75	-0.6250	-0.375	2	-1	1	0.125
P3	0.25	0.25	-0.125	0.375	0.5	0.5	0.5	0.125
P4	0.25	0.25	-0.375	0.625	2	1	-1	0.125
P5	0.25	0.25	0	0	0.125	0.25	0.25	0.125

Output :
Best Position : 2.5 ; Best = 26.2500

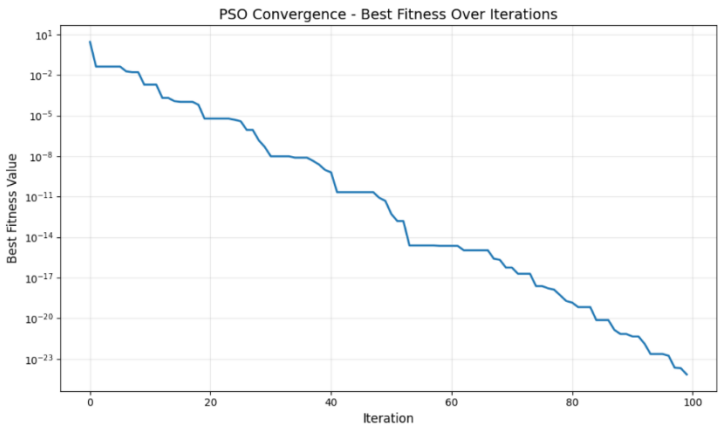
Code:

```
import numpy as np

# ---------- Step 1: Define Problem (Portfolio Optimization) ----------
# Expected returns for 4 assets (example data)
returns = np.array([0.12, 0.18, 0.15, 0.10])

# Covariance matrix of returns (risk measure)
cov_matrix = np.array([
    [0.010, 0.002, 0.001, 0.003],
    [0.002, 0.030, 0.002, 0.004],
    [0.001, 0.002, 0.020, 0.002],
    [0.003, 0.004, 0.002, 0.025]
])

# Fitness function: Sharpe ratio (maximize return / risk)
def fitness(weights):
    weights = np.array(weights)
    portfolio_return = np.dot(weights, returns)
    portfolio_risk = np.sqrt(np.dot(weights.T, np.dot(cov_matrix, weights)))
    if portfolio_risk == 0:  # avoid division by zero
        return -999
    return portfolio_return / portfolio_risk

# ---------- Step 2: Initialize PSO Parameters ----------
num_particles = 30
num_assets = len(returns)
iterations = 100

w = 0.7      # inertia weight
c1 = 1.5     # cognitive coefficient
c2 = 1.5     # social coefficient

# ---------- Step 3: Initialize Particles ----------
positions = np.random.dirichlet(np.ones(num_assets), size=num_particles)  # weights sum=1
velocities = np.random.rand(num_particles, num_assets) * 0.1

personal_best_positions = positions.copy()
personal_best_scores = np.array([fitness(p) for p in positions])

global_best_position = personal_best_positions[np.argmax(personal_best_scores)]
global_best_score = np.max(personal_best_scores)

# ---------- Step 4: Main Loop ----------
for _ in range(iterations):
    for i in range(num_particles):
```

```
        # Update velocity
        r1, r2 = np.random.rand(num_assets), np.random.rand(num_assets)
        velocities[i] = (w * velocities[i]
                         + c1 * r1 * (personal_best_positions[i] - positions[i])
                         + c2 * r2 * (global_best_position - positions[i]))

        # Update position (weights must be valid portfolio)
        positions[i] += velocities[i]
        positions[i] = np.maximum(positions[i], 0)     # no negative weights
        positions[i] /= np.sum(positions[i])           # normalize to sum=1

        # Evaluate fitness
        score = fitness(positions[i])

        # Update personal best
        if score > personal_best_scores[i]:
            personal_best_scores[i] = score
            personal_best_positions[i] = positions[i].copy()

        # Update global best
        if score > global_best_score:
            global_best_score = score
            global_best_position = positions[i].copy()

# ---------- Step 5: Output Result ----------
print("Optimal Portfolio Weights:", global_best_position)
print("Best Sharpe Ratio:", global_best_score)
```

Output:

```
Starting Particle Swarm Optimization...
Initial Global Best Fitness: 2.796646
--------------------------------------------------
Iteration 20: Best Fitness = 0.000006
Iteration 40: Best Fitness = 0.000000
Iteration 60: Best Fitness = 0.000000
Iteration 80: Best Fitness = 0.000000
Iteration 100: Best Fitness = 0.000000
--------------------------------------------------

Optimization Complete!
Best Solution Found: [ 8.36995227e-13 -9.76276964e-14]
Best Fitness Value: 0.000000
Optimal Solution (Expected): [0, 0] with fitness 0
```

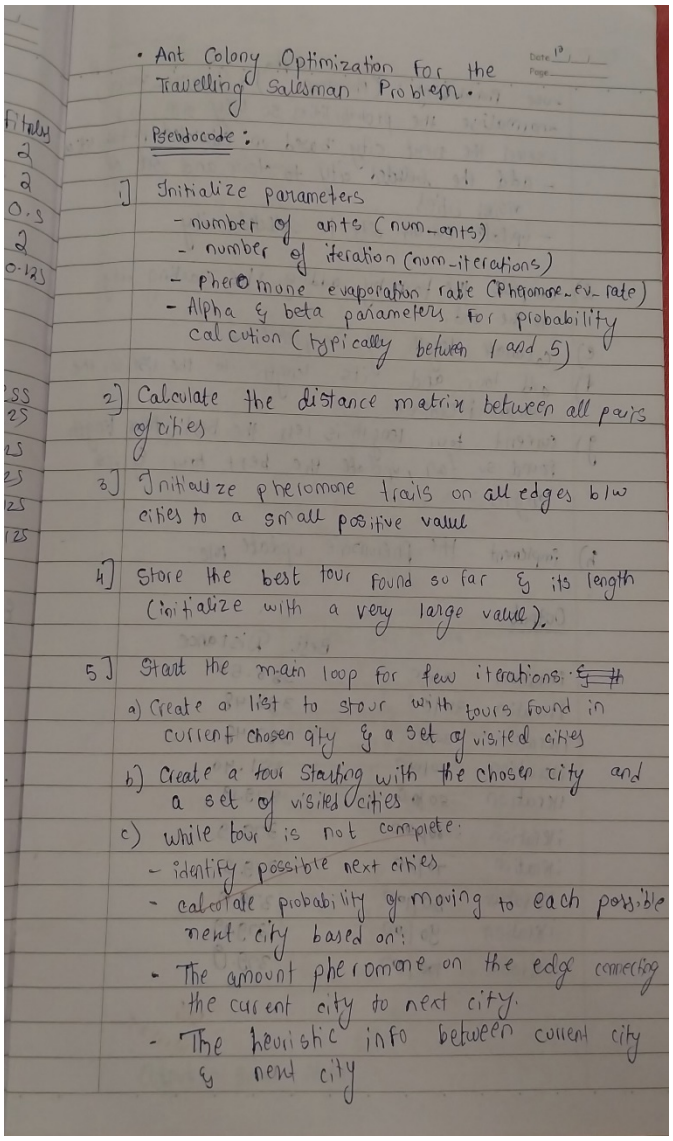


Program 4

Ant Colony Optimization for the Traveling Salesman Problem

Ant Colony Optimization (ACO) for the Vehicle Routing Problem (VRP): It involves finding optimal routes for multiple vehicles to deliver goods to a set of customers from a central depot.

Algorithm:



- Ant Colony Optimization for the Travelling Salesman Problem.

Pseudocode:

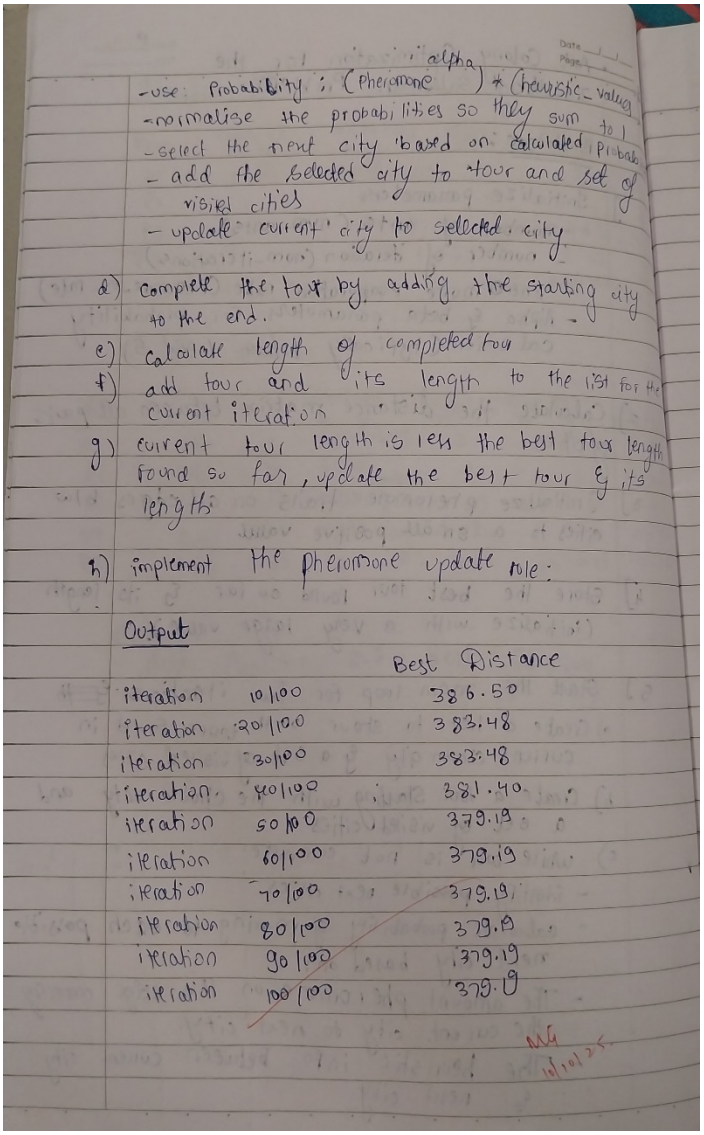
1] Initialize parameters
- number of ants (num_ants)
- number of iteration (num_iterations)
- Pheromone evaporation rate (Pheromone_ev_rate)
- Alpha & beta parameters for probability calcution (typically between 1 and 5)

2] Calculate the distance matrix between all pairs of cities

3] Initialize pheromone trails on all edges b/w cities to a small positive value

4] Store the best tour found so far & its length (initialize with a very large value).

5] Start the main loop for few iterations:
a) Create a list to store with tours found in current chosen city & a set of visited cities
b) Create a tour starting with the chosen city and a set of visited cities.
c) while tour is not complete:
- identify possible next cities
- calculate probability of moving to each possible next city based on:
- The amount pheromone on the edge connecting the cursent city to next city.
- The heuristic info between current city & next city



- use Probability : (Pheromone ^ alpha) * (heuristic_value
- normalize the probabilities so they sum to 1
- select the next city based on calculated Probab
- add the selected city to tour and set of visited cities
- update current city to selected city

d) Complete the tour by adding the starting city to the end.
e) Calculate length of completed tour
f) add tour and its length to the list for the current iteration
g) current tour length is less the best tour length found so far, update the best tour & its length
h) implement the pheromone update role:

Output

		Best Distance
iteration	10/100	386.50
iteration	20/100	383.48
iteration	30/100	383.48
iteration	40/100	381.40
iteration	50/100	379.19
iteration	60/100	379.19
iteration	70/100	379.19
iteration	80/100	379.19
iteration	90/100	379.19
iteration	100/100	379.19

Code:

```
import numpy as np
import random

# Coordinates of depot + customers (0 is depot)
coords = np.array([
    [40, 50],  # depot
    [45, 68], [50, 30], [55, 20], [60, 80], [65, 60], [70, 40]
])

num_vehicles = 2
num_ants = 10
num_iterations = 100
alpha = 1.0  # pheromone importance
beta = 5.0  # heuristic importance (inverse distance)
rho = 0.5    # pheromone evaporation rate
initial_pheromone = 1.0

num_cities = len(coords)

# Distance matrix
dist_matrix = np.sqrt(((coords[:, None] - coords[None, :])**2).sum(axis=2))

# Heuristic matrix (inverse distance), avoid division by zero
heuristic = 1 / (dist_matrix + np.diag([np.inf]*num_cities))

# Initialize pheromone trails
pheromone = np.ones((num_cities, num_cities)) * initial_pheromone
```

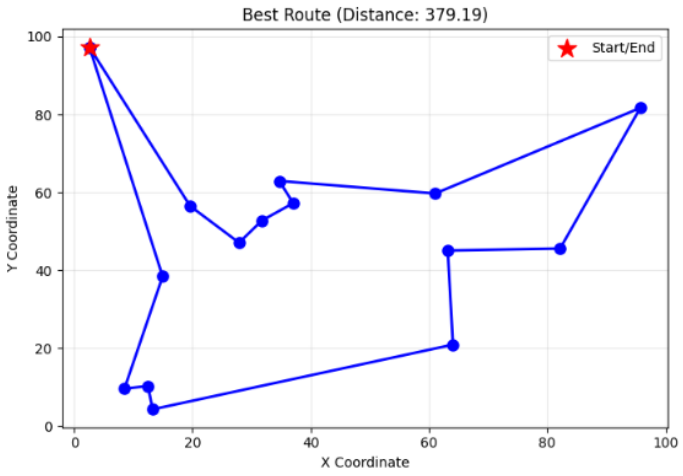
```
def choose_next_city(current_city, unvisited, pheromone, heuristic):
    pheromone_vals = pheromone[current_city][unvisited] ** alpha
    heuristic_vals = heuristic[current_city][unvisited] ** beta
    probs = pheromone_vals * heuristic_vals
    probs /= probs.sum()
    return np.random.choice(unvisited, p=probs)

def construct_solution():
    routes = [[] for _ in range(num_vehicles)]
    unvisited = set(range(1, num_cities))  # customers only
    for v in range(num_vehicles):
        routes[v].append(0)  # start from depot

    while unvisited:
        for v in range(num_vehicles):
            current_city = routes[v][-1]
            candidates = list(unvisited)
            if not candidates:
                break
            next_city = choose_next_city(current_city, candidates, pheromone, heuristic)
            routes[v].append(next_city)
            unvisited.remove(next_city)
            if not unvisited:
                break

    # Return to depot
    for v in range(num_vehicles):
        routes[v].append(0)
    return routes

def route_length(route):
    length = 0
    for i in range(len(route)-1):
        length += dist_matrix[route[i], route[i+1]]
    return length

best_routes = None
best_length = float('inf')

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

    for _ in range(num_ants):
        routes = construct_solution()
        total_length = sum(route_length(r) for r in routes)
        all_routes.append(routes)
        all_lengths.append(total_length)
```

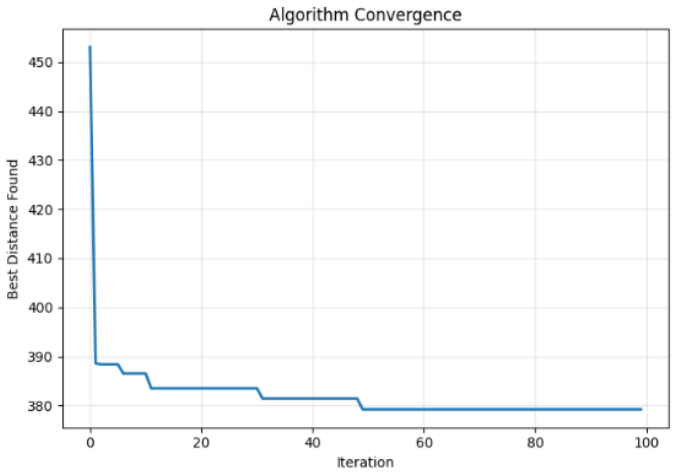
```
        if total_length < best_length:
            best_length = total_length
            best_routes = routes

    # Pheromone evaporation
    pheromone *= (1 - rho)

    # Pheromone update (only best ant deposits pheromone)
    for route in best_routes:
        for i in range(len(route)-1):
            from_city = route[i]
            to_city = route[i+1]
            pheromone[from_city][to_city] += 1 / best_length
            pheromone[to_city][from_city] += 1 / best_length

print("Best total route length:", best_length)
for v, route in enumerate(best_routes):
    print(f"Vehicle {v+1} route: {route}")
```

Output:

```
Created 15 cities
Starting Ant Colony Optimization...
Iteration 10/100, Best Distance: 386.50
Iteration 20/100, Best Distance: 383.48
Iteration 30/100, Best Distance: 383.48
Iteration 40/100, Best Distance: 381.40
Iteration 50/100, Best Distance: 379.19
Iteration 60/100, Best Distance: 379.19
Iteration 70/100, Best Distance: 379.19
Iteration 80/100, Best Distance: 379.19
Iteration 90/100, Best Distance: 379.19
Iteration 100/100, Best Distance: 379.19

Optimization Complete!

==================================================
BEST SOLUTION FOUND
==================================================
Route: [4, np.int64(1), np.int64(7), np.int64(2), np.int64(12), np.int64(9), np.int64(0), np.int64(13), np.int64(6), np.int
64(14), np.int64(5), np.int64(3), np.int64(10), np.int64(8), 11]
Total Distance: 379.19
==================================================
```

==========



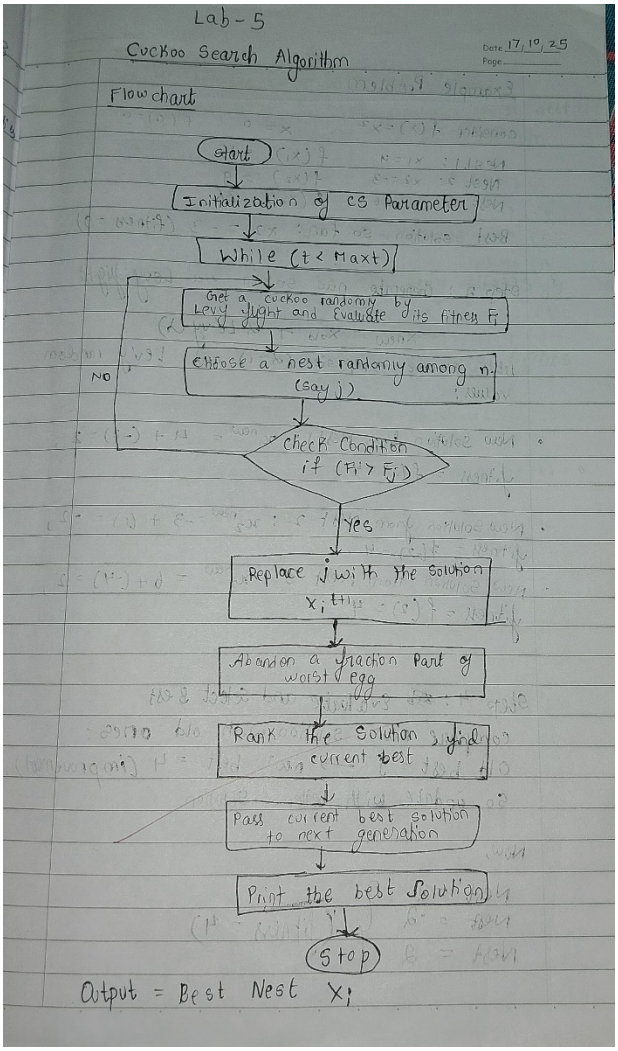


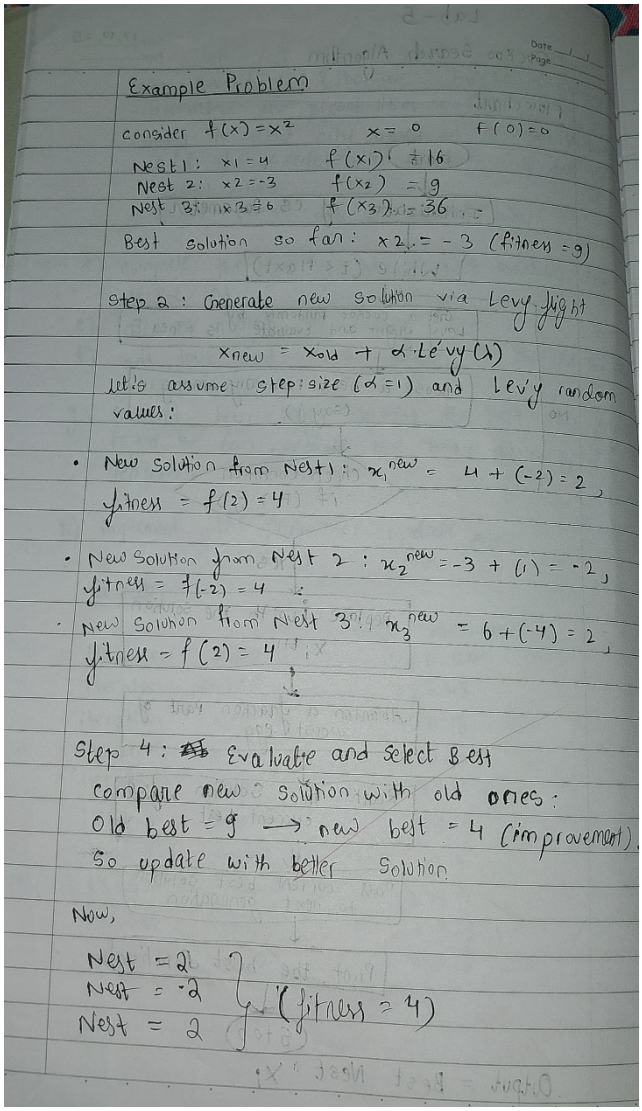
Program 5

Cuckoo Search (CS)

Cuckoo Search Algorithms: We need to maximize the total value of selected items without exceeding the knapsack's weight capacity. Using the Cuckoo Search Algorithm, each solution is a binary vector, new solutions are generated via Lévy flights, and the best feasible solution is iteratively improved while abandoning poor solutions with a probability.

Algorithm:





Step 4: Abandon worst Nest

- with probability $P_a = 0.25$, abandon worst nest
- Nest 3 abandoned, replaced with random
  $x3 = -1 \longrightarrow f(-1) = 1$
- Best solution now: $x = -1 \quad f(x) = 1$.

Step 5: Iterate Until Convergence

- Continue process with new Levy flights & abandonment
- Next iterations improve solution:
  - Continue process with new Levy flights and abandonment.
  - Next iteration improve solution:
    Ex: $x = -0.5 \longrightarrow f(x) = 0.25$
    Eventually converges near $x = 0 \quad f(x) = 0$
  - Global minimum found!

Code:

```
import numpy as np
import random

# ---------------- Knapsack Problem Setup ----------------
# Example items: (value, weight)
items = [(60, 10), (100, 20), (120, 30)]
capacity = 50
n = len(items)

def fitness(solution):
    total_value = total_weight = 0
    for i in range(n):
        if solution[i] == 1:
            total_value += items[i][0]
            total_weight += items[i][1]
    if total_weight > capacity:
        return 0  # invalid solution
    return total_value

# ---------------- Cuckoo Search Algorithm ----------------
def levy_flight(Lambda):
    u = np.random.normal(0, 1) * np.power(abs(np.random.normal(0, 1)), -1.0 / Lambda)
    v = np.random.normal(0, 1)
    step = u / abs(v) ** (1 / Lambda)
    return step

def get_random_solution():
    return [random.randint(0, 1) for _ in range(n)]

def cuckoo_search(num_nests=10, pa=0.25, max_iter=100):
```

```
    nests = [get_random_solution() for _ in range(num_nests)]
    best = max(nests, key=fitness)

    for _ in range(max_iter):
        # Generate new solution via Levy flight
        cuckoo = best[:]
        step = int(abs(round(levy_flight(1.5)))) % n
        pos = random.randint(0, n-1)
        cuckoo[pos] = 1 - cuckoo[pos]  # flip bit

        # Replace a random nest if better
        j = random.randint(0, num_nests-1)
        if fitness(cuckoo) > fitness(nests[j]):
            nests[j] = cuckoo

        # Abandon some nests with probability pa
        for i in range(num_nests):
            if random.random() < pa:
                nests[i] = get_random_solution()

        # Update best
        best = max(nests, key=fitness)

    return best, fitness(best)

# ------------------ Run the algorithm ------------------
solution, value = cuckoo_search()
print("Best solution:", solution)
print("Total value:", value)
```

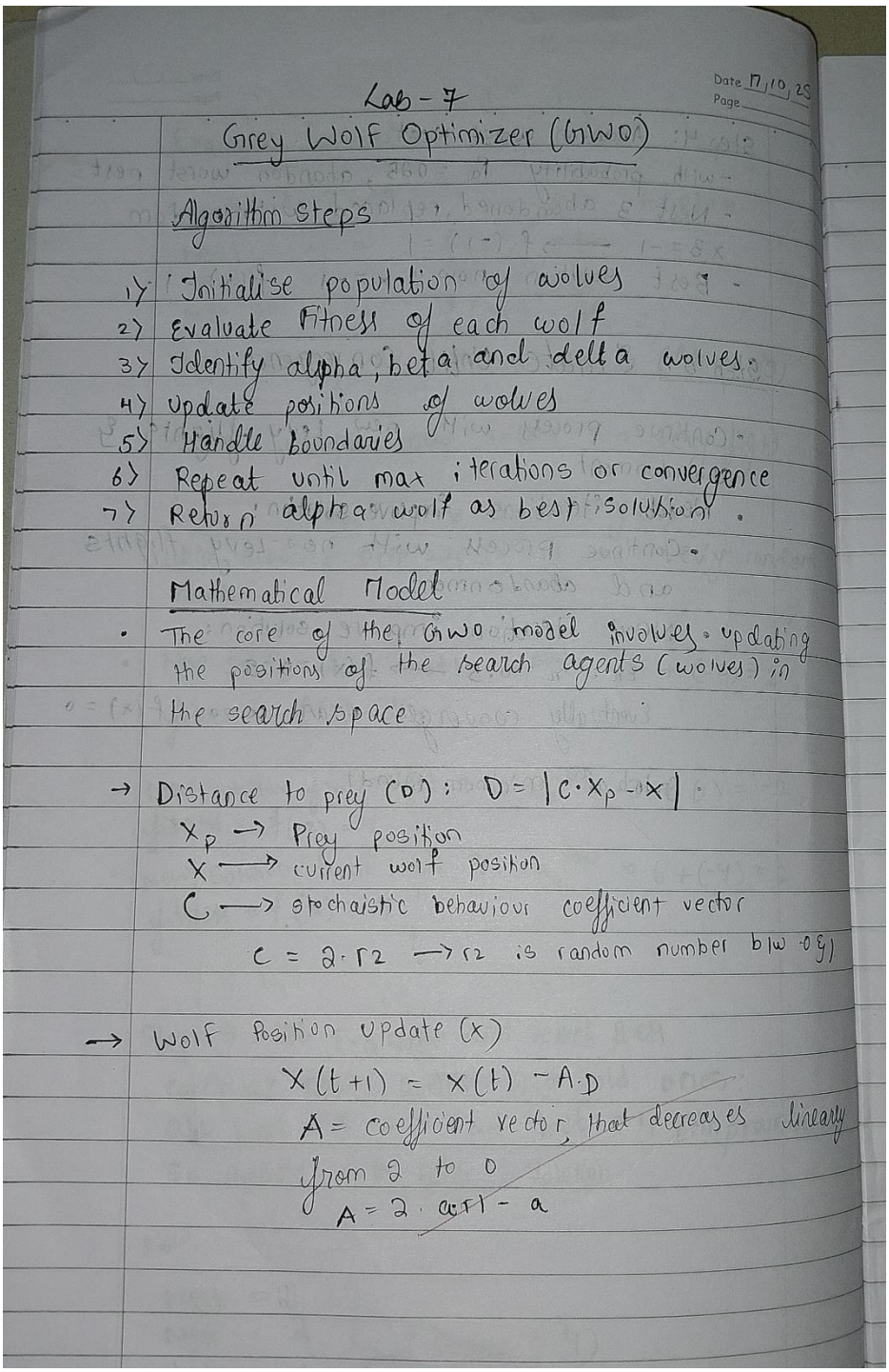
Output:

```
============================================================
        Cuckoo Search Optimization Algorithm
============================================================
Function Name     : sphere_function(x)
Mathematical Form: f(x) = Σ(x_i²)
Dimensions        : 2
Search Bounds     : [-10, 10]
Global Minimum    : f(0, 0) = 0
============================================================

Initial Best Solution: [-1.618897, 1.504998]
Initial Best Fitness: f(x) = 4.885847

Iteration  20: x = [0.000180, 0.013211] | f(x) = 0.000175
Iteration  40: x = [0.003584, -0.003925] | f(x) = 0.000028
Iteration  60: x = [-0.003263, -0.003471] | f(x) = 0.000023
Iteration  80: x = [-0.002295, -0.001751] | f(x) = 0.000008
Iteration 100: x = [-0.002295, -0.001751] | f(x) = 0.000008

============================================================
                        RESULTS
============================================================
Optimal Solution : x = [-0.002295, -0.001751]
Optimal Value    : f(x) = 0.000008
Function         : sphere_function(x) = Σ(x_i²)
============================================================
```

Program 6

Grey Wolf Optimizer (GWO)
Using the Grey Wolf Optimizer (GWO), we aim to find the shortest, obstacle-free path by modeling the search agents (wolves) to iteratively converge toward the best position (path node) in the environment. The algorithm simulates the grey wolves' hunting hierarchy and encircling behavior to efficiently navigate the space from the start point.

Algorithm:



Lab - 7

Date 11/10/25
Page

Grey Wolf Optimizer (GWO)

Algorithm steps

1) Initialise population of wolves
2) Evaluate fitness of each wolf
3) Identify alpha, beta and delta wolves.
4) Update positions of wolves
5) Handle boundaries
6) Repeat until max iterations or convergence
7) Return alpha wolf as best solution.

Mathematical Model

- The core of the GWO model involves updating the positions of the search agents (wolves) in the search space

→ Distance to prey (D): $D = |C \cdot X_p - X|$

X_p → Prey position
X → current wolf position
C → stochastic behaviour coefficient vector

$C = 2 \cdot r2$ → r2 is random number b/w 0 & 1

→ Wolf Position update (X)

$X(t+1) = X(t) - A \cdot D$

A = coefficient vector, that decreases linearly from 2 to 0

$A = 2 \cdot a \cdot r1 - a$

Code:

```
import numpy as np
import random

# === Grid setup ===
GRID_SIZE = 5
START = (0, 0)
GOAL = (4, 4)
OBSTACLES = [(2, i) for i in range(1, 4)]  # Vertical wall in column 2, rows 1 to 3

# === Parameters ===
POP_SIZE = 10
MAX_ITER = 50
PATH_LENGTH = 20  # fewer steps needed for small grid

# === Helper Functions ===

def is_valid(pos):
    x, y = pos
    return 0 <= x < GRID_SIZE and 0 <= y < GRID_SIZE and pos not in OBSTACLES

def move_toward_goal(current):
    moves = [(0,1), (1,0), (0,-1), (-1,0)]
    random.shuffle(moves)
```

```
cx, cy = current gx,
gy = GOAL
        moves.sort(key=lambda m: abs((cx + m[0]) - gx) + abs((cy + m[1]) - gy))
        for dx, dy in moves:
            new_pos = (cx + dx, cy + dy)
            if is_valid(new_pos):
                return new_pos
        return current

    def generate_random_path():
        path = [START]
        visited = set(path)
        current = START
        for _ in range(PATH_LENGTH):
            current = move_toward_goal(current)
            if current in visited:
                continue
            path.append(current)
            visited.add(current)
            if current == GOAL:
                break
        return path

    def path_cost(path):
        cost = len(path)
        if path[-1] != GOAL:
            dist = abs(path[-1][0] - GOAL[0]) + abs(path[-1][1] - GOAL[1])
            cost += 100 + dist
        for pos in path:
            if pos in OBSTACLES:
                cost += 50
        return cost

    # === GWO Optimization ===

    def gwo_optimize():
        wolves = [generate_random_path() for _ in range(POP_SIZE)]

        for iteration in range(MAX_ITER):
            wolves.sort(key=path_cost)
            alpha, beta, delta = wolves[0], wolves[1], wolves[2]
            a = 2 - iteration * (2 / MAX_ITER)

            for i in range(3, POP_SIZE):
                new_path = []
                for j in range(min(len(alpha), len(wolves[i]), PATH_LENGTH)):
                    A = 2 * a * random.random() - a
                    C = 2 * random.random()
                    x_alpha = np.array(alpha[j])
```

```
            x_wolf = np.array(wolves[i][j])
            D_alpha = abs(C * x_alpha - x_wolf)
            X1 = x_alpha - A * D_alpha

            A = 2 * a * random.random() - a
            C = 2 * random.random()
            x_beta = np.array(beta[j])
            D_beta = abs(C * x_beta - x_wolf)
            X2 = x_beta - A * D_beta

            A = 2 * a * random.random() - a
            C = 2 * random.random()
            x_delta = np.array(delta[j])
            D_delta = abs(C * x_delta - x_wolf)
            X3 = x_delta - A * D_delta

            X_new = (X1 + X2 + X3) / 3
            X_new = tuple(map(int, np.clip(np.round(X_new), 0, GRID_SIZE - 1)))

            if is_valid(X_new):
                new_path.append(X_new)
            else:
                if new_path:
                    new_path.append(move_toward_goal(new_path[-1]))
                else:
                    new_path.append(move_toward_goal(START))
        wolves[i] = new_path

    best_path = sorted(wolves, key=path_cost)[0]
    return best_path

# === Textual Output ===

def print_grid(path):
    grid = [["." for _ in range(GRID_SIZE)] for _ in range(GRID_SIZE)]

    for x, y in OBSTACLES:
        grid[y][x] = "#"  # Obstacle

    for x, y in path:
        if (x, y) != START and (x, y) != GOAL and grid[y][x] != "#":
            grid[y][x] = "*"

    sx, sy = START
    gx, gy = GOAL
    grid[sy][sx] = "S"
    grid[gy][gx] = "G"

    print("\n=== GWO Path Grid ===")
```

```
    for row in grid:
        print(" ".join(row))

    print("\nBest Path (coordinates):")
    print(path)

    print(f"\nPath Length: {len(path)}")
    print(f"Cost: {path_cost(path)}")

# === Run ===

best = gwo_optimize()
print_grid(best)
```

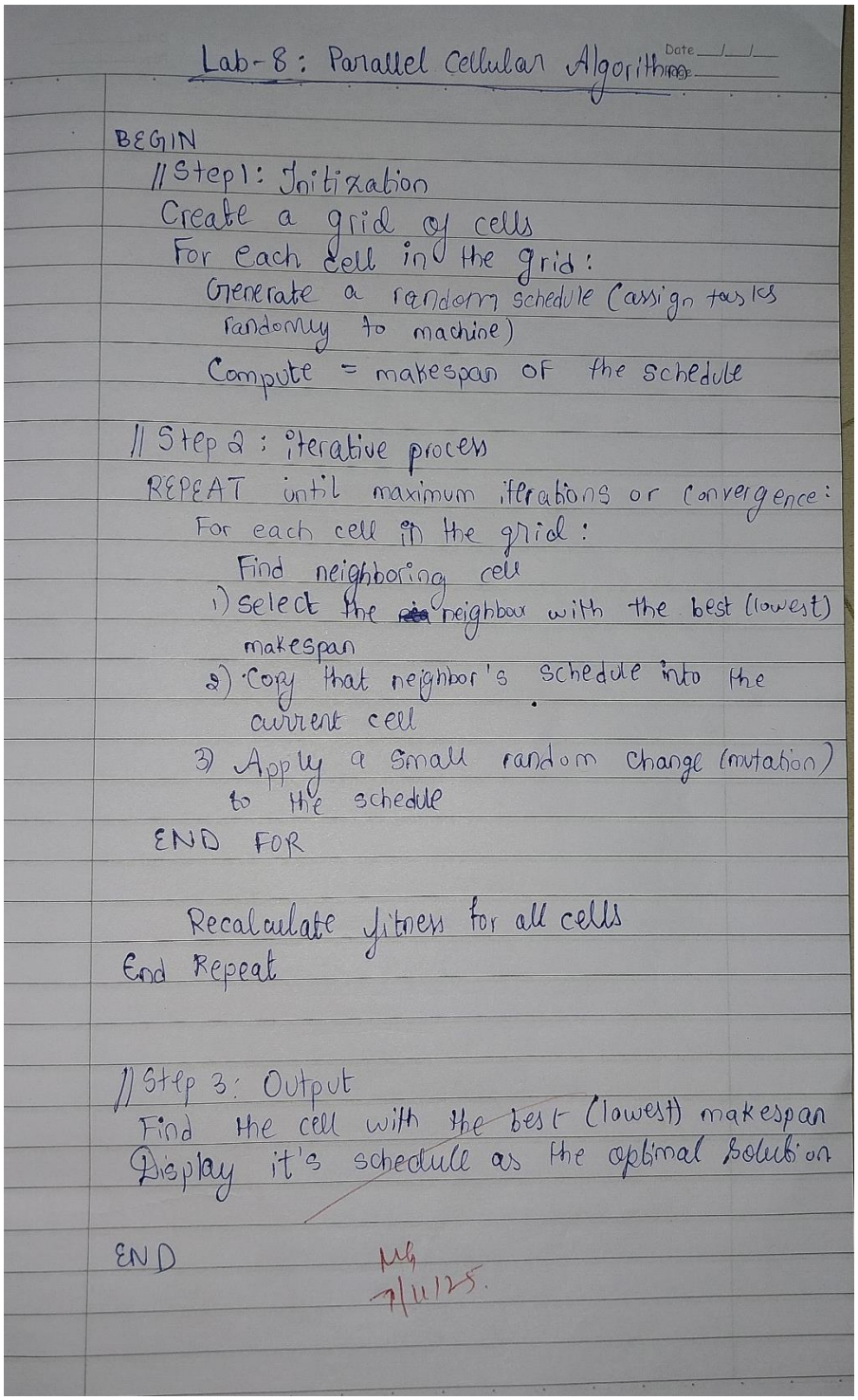
Output:

```
======================================================================
           Grey Wolf Optimizer (GWO) Algorithm
======================================================================
Function Name    : sin_squared_function(x)
Mathematical Form: f(x) = Σ(sin²(x_i))
Dimensions       : 2
Search Bounds    : [-π, π]
Global Minimum   : f(0, 0) = 0
Pack Size        : 30 wolves
======================================================================

Initial Alpha (Best) : x = [0.538482, 2.860881] | f(x) = 0.339749
Initial Beta  (2nd)  : x = [-2.502765, -0.079752] | f(x) = 0.361866
Initial Delta (3rd)  : x = [-2.591903, 0.342885] | f(x) = 0.385960

Iteration  20: Alpha x = [-3.141593, 0.000000] | f(x) = 0.000000
Iteration  40: Alpha x = [-3.141593, -0.000000] | f(x) = 0.000000
Iteration  60: Alpha x = [-3.141593, -0.000000] | f(x) = 0.000000
Iteration  80: Alpha x = [-3.141593, -0.000000] | f(x) = 0.000000
Iteration 100: Alpha x = [-3.141593, -0.000000] | f(x) = 0.000000

======================================================================
                          RESULTS
======================================================================
Optimal Solution : x = [-3.141593, -0.000000]
Optimal Value    : f(x) = 0.000000
Function         : sin_squared_function(x) = Σ(sin²(x_i))
Target Solution  : x = [0.0, 0.0] (or ±π, ±2π, ...)
======================================================================

Wolf Hierarchy:
  α (Alpha) - Leader, guides the hunt (best solution)
  β (Beta)  - Subordinate, assists alpha (2nd best)
  δ (Delta) - Subordinate, assists beta (3rd best)
  ω (Omega) - Rest of the pack follows guidance
======================================================================
```

Program 7

Parallel Cellular Algorithms and Programs
The task is to perform edge detection or noise reduction in an image using Parallel Cellular Automata (PCA), where each pixel (cell) interacts with its neighbors to enhance edges or reduce noise iteratively.

Algorithm:



Lab-8: Parallel Cellular Algorithms Date __/__/__

BEGIN
// Step 1: Initialization
Create a grid of cells
For each cell in the grid:
Generate a random schedule (assign tasks randomly to machine)
Compute = makespan of the schedule

// Step 2: iterative process
REPEAT until maximum iterations or convergence:
For each cell in the grid:
Find neighboring cell
1) select the neighbor with the best (lowest) makespan
2) Copy that neighbor's schedule into the current cell
3) Apply a small random change (mutation) to the schedule
END FOR

Recalculate fitness for all cells
End Repeat

// Step 3: Output
Find the cell with the best (lowest) makespan
Display it's schedule as the optimal solution

END

MG
7/11/25.

Code:

```
import numpy as np
import cv2
import matplotlib.pyplot as plt

# Function for Cellular Automata (Edge Detection or Noise Reduction)
def cellular_automata(image, iterations=10, threshold=30):
    grid = image.copy()  # Initialize grid (image as 2D array)
    neighbors = [(-1, -1), (-1, 0), (-1, 1), (0, -1), (0, 0), (0, 1), (1, -1), (1, 0), (1, 1)]

    for iteration in range(iterations):
        updated_grid = grid.copy()

        for i in range(1, len(grid) - 1):  # Loop through pixels (excluding borders)

            for j in range(1, len(grid[0]) - 1):
            pixel = grid[i, j]
                neighbor_vals = [grid[i+di, j+dj] for (di, dj) in neighbors]

                # Edge detection: large difference with neighbors indicates edge
                if max(neighbor_vals) - min(neighbor_vals) > threshold:
                    updated_grid[i, j] = 255  # Edge pixel
                else:
                    # Noise reduction: average with neighbors for smoothing
                    new_pixel_value = sum(np.clip(neighbor_vals, 0, 255)) // 8  # Clipping before averaging

                    # Clip the new pixel value to the range 0-255
                    updated_grid[i, j] = np.clip(new_pixel_value, 0, 255)

        grid = updated_grid  # Update the grid with new values

    return grid  # Output updated image

# Set numpy to ignore overflow warnings
np.seterr(over='ignore')

# Generate a smaller dummy grayscale image (random noise)
# Create a 5x5 pixel image with random values between 0 and 255
image = np.random.randint(0, 256, (5, 5), dtype=np.uint8)

# Print the original image
print("Original Image (Pixel Values):")
for row in image:
    print(row)

# Apply the cellular automata algorithm
iterations = 10
```

```
threshold = 30
processed_image = cellular_automata(image, iterations, threshold)

# Print the processed image
print("\nProcessed Image (Pixel Values):")
for row in processed_image:
    print(row)

# Visualize the images using matplotlib
plt.figure(figsize=(8,4))

plt.subplot(1,2,1)
plt.title('Original Image')
plt.imshow(image, cmap='gray', vmin=0, vmax=255)
plt.axis('off')

plt.subplot(1,2,2)
plt.title('Processed Image')
plt.imshow(processed_image, cmap='gray', vmin=0, vmax=255)
plt.axis('off')

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

Output:

```
Original Image (Pixel Values):
[229  27  88 154 254]
[125  20  90 185 144]
[200 161  35  19  61]
[108 251   0 187  28]
[ 12  50  54 119 225]

Processed Image (Pixel Values):
[229  27  88 154 254]
[125 255 255 255 144]
[200 255  30 255  61]
[108 255 255 255  28]
[ 12  50  54 119 225]
```