|  |
| --- |
| **VISVESVARAYA TECHNOLOGICAL UNIVERSITY**  **“JnanaSangama”, Belgaum -590014, Karnataka.**    **LAB RECORD**  **Bio Inspired Systems (23CS5BSBIS)**  ***Submitted by***  **Navanidhi D J (1BM23CS204)**  ***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 **Navanidhi D J (1BM23CS204),** 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.

|  |  |
| --- | --- |
| Sneha S Bagalkot  Assistant Professor  Department of CSE, BMSCE | Dr. Kavitha Sooda  Professor & HOD  Department of CSE, BMSCE |

**Index**

|  |  |  |  |
| --- | --- | --- | --- |
| **Sl.**  **No.** | **Date** | **Experiment Title** | **Page No.** |
| 1 | 18/8/25 | Genetic Algorithm for optimization for neural network weight optimization. | 4-7 |
| 2 | 1/9/25 | Particle Swarm Optimization for Function Optimization for K-means clutering | 8-10 |
| 3 | 8/9/25 | Ant Colony Optimization for the Traveling Salesman Problem | 11-15 |
| 4 | 22/9/25 | Cuckoo search algorithm : Power system and energy. | 16-19 |
| 5 | 29/9/25 | Grey wolf optimization (Application: network routing optimization) | 20-23 |
| 6 | 13/10/25 | Parallel Cellular Algorithm for modelling the spread of diseases. | 24-28 |
| 7 | 25/8/25 | Optimization via Gene Expression Algorithms for TSP: | 29-34 |

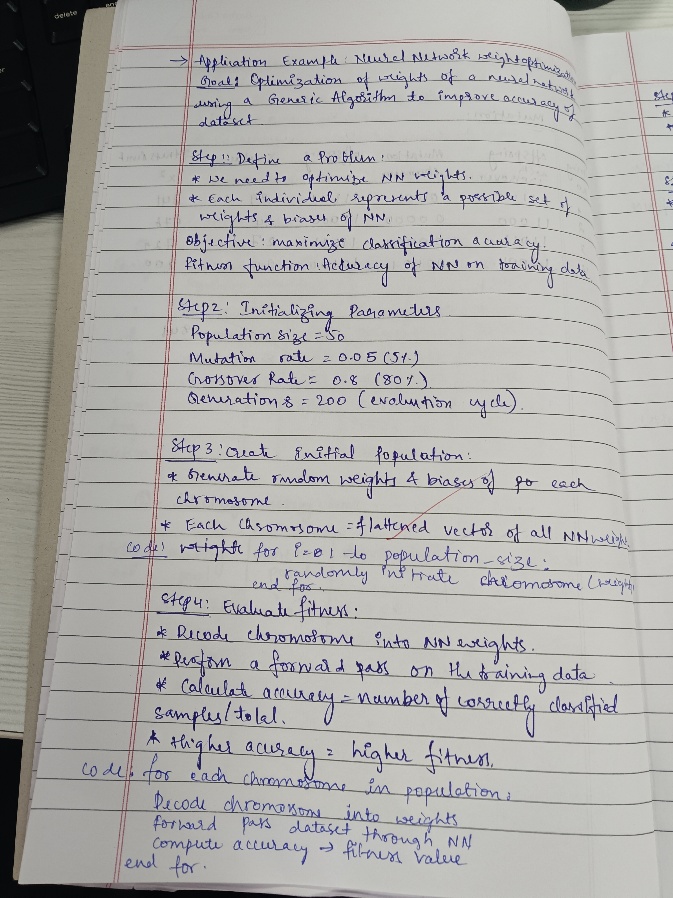
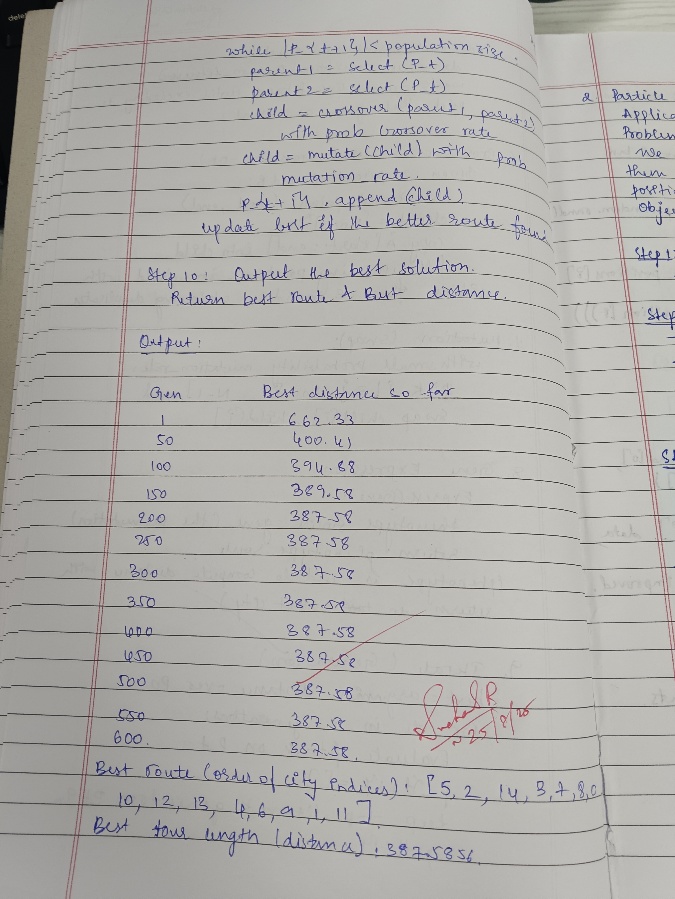
Github Link:

https://github.com/NavanidhiDJ/Bio\_inspired\_systems

**Program 1**

Genetic Algorithm for optimization for neural network weight optimization.

Algorithm:

Code:

import numpy as np

# -----------------------------

# 1. Define Problem: Neural Network (XOR classification)

# -----------------------------

# Dataset (XOR problem)

X = np.array([[0,0],

[0,1],

[1,0],

[1,1]])

y = np.array([[0],

[1],

[1],

[0]])

# Neural network architecture: 2 -> 4 -> 1

input\_size = 2

hidden\_size = 4

output\_size = 1

# -----------------------------

# Helper functions

# -----------------------------

def sigmoid(x):

return 1 / (1 + np.exp(-x))

def forward\_pass(weights, X):

"""

Forward pass of NN with given weights.

weights = flattened array containing all weights and biases

"""

# Unpack weights

w1\_size = input\_size \* hidden\_size

b1\_size = hidden\_size

w2\_size = hidden\_size \* output\_size

b2\_size = output\_size

w1 = weights[:w1\_size].reshape(input\_size, hidden\_size)

b1 = weights[w1\_size:w1\_size+b1\_size].reshape(1, hidden\_size)

w2 = weights[w1\_size+b1\_size:w1\_size+b1\_size+w2\_size].reshape(hidden\_size, output\_size)

b2 = weights[-b2\_size:].reshape(1, output\_size)

# Forward pass

hidden = sigmoid(np.dot(X, w1) + b1)

output = sigmoid(np.dot(hidden, w2) + b2)

return output

def fitness\_function(weights):

""" Fitness = accuracy of predictions """

predictions = forward\_pass(weights, X)

predicted\_labels = (predictions > 0.5).astype(int)

accuracy = np.mean(predicted\_labels == y)

return accuracy

# -----------------------------

# 2. Initialize GA Parameters

# -----------------------------

pop\_size = 50 # population size

n\_generations = 200 # number of generations

mutation\_rate = 0.1

crossover\_rate = 0.8

# Total number of weights in NN

n\_weights = input\_size\*hidden\_size + hidden\_size + hidden\_size\*output\_size + output\_size

# -----------------------------

# 3. Create Initial Population

# -----------------------------

population = np.random.uniform(-1, 1, (pop\_size, n\_weights))

# -----------------------------

# GA Operators

# -----------------------------

def selection(population, fitness):

""" Tournament selection """

idx1, idx2 = np.random.randint(0, len(population), 2)

return population[idx1] if fitness[idx1] > fitness[idx2] else population[idx2]

def crossover(parent1, parent2):

""" Single-point crossover """

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

point = np.random.randint(1, n\_weights-1)

child1 = np.concatenate([parent1[:point], parent2[point:]])

child2 = np.concatenate([parent2[:point], parent1[point:]])

return child1, child2

else:

return parent1.copy(), parent2.copy()

def mutate(individual):

""" Random mutation """

for i in range(len(individual)):

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

individual[i] += np.random.normal(0, 0.5) # small perturbation

return individual

# -----------------------------

# 4–8. GA Main Loop

# -----------------------------

best\_fitness\_history = []

for generation in range(n\_generations):

# Evaluate fitness

fitness = np.array([fitness\_function(ind) for ind in population])

# Track best solution

best\_idx = np.argmax(fitness)

best\_fitness = fitness[best\_idx]

best\_fitness\_history.append(best\_fitness)

if generation % 20 == 0:

print(f"Gen {generation}: Best Fitness = {best\_fitness:.4f}")

# New population

new\_population = []

while len(new\_population) < pop\_size:

parent1 = selection(population, fitness)

parent2 = selection(population, fitness)

child1, child2 = crossover(parent1, parent2)

new\_population.append(mutate(child1))

new\_population.append(mutate(child2))

population = np.array(new\_population[:pop\_size])

# -----------------------------

# 9. Output Best Solution

# -----------------------------

fitness = np.array([fitness\_function(ind) for ind in population])

best\_idx = np.argmax(fitness)

best\_weights = population[best\_idx]

print("\nBest solution found:")

print("Accuracy:", fitness[best\_idx])

# Test the best weights on XOR

preds = forward\_pass(best\_weights, X)

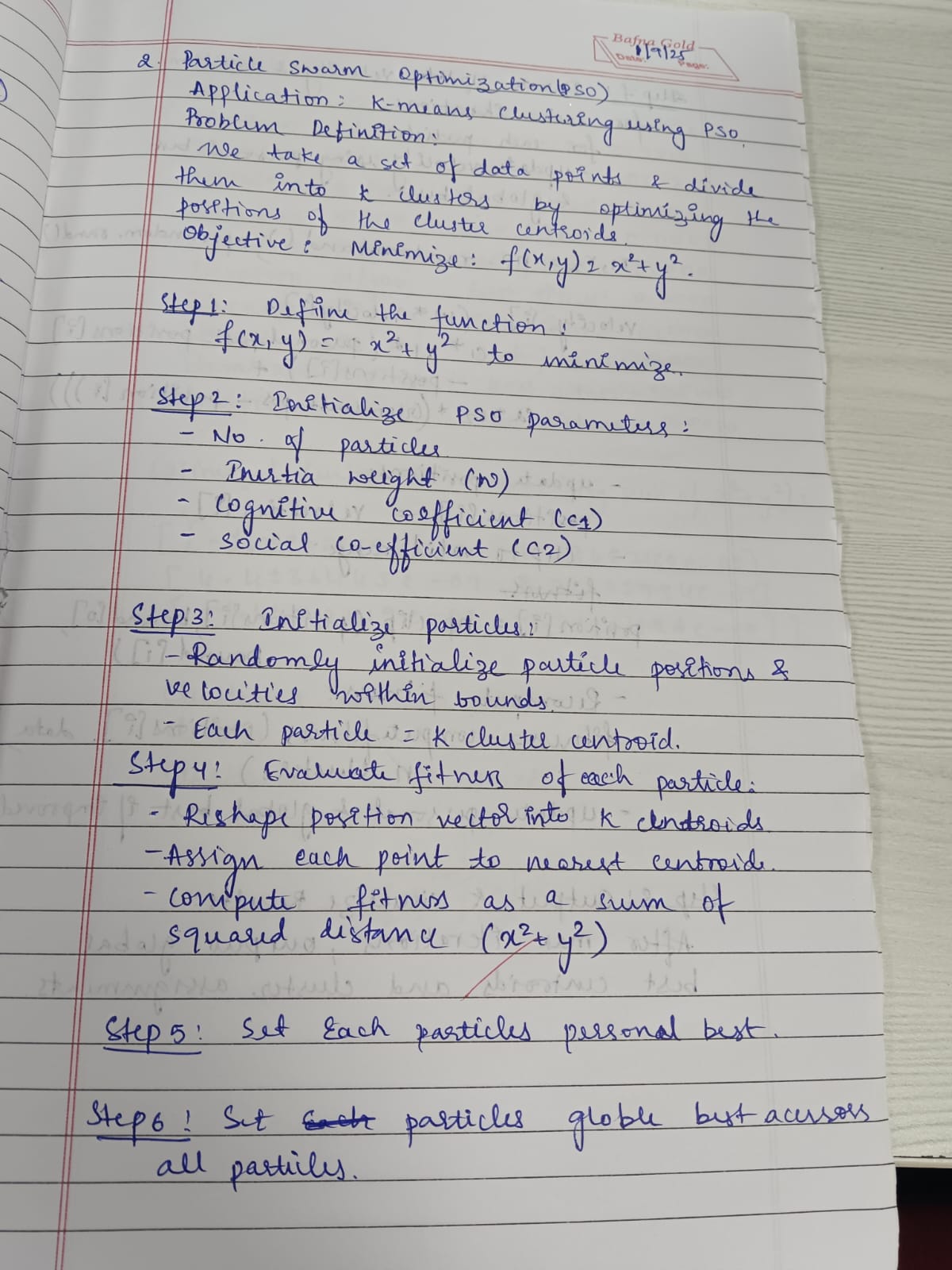
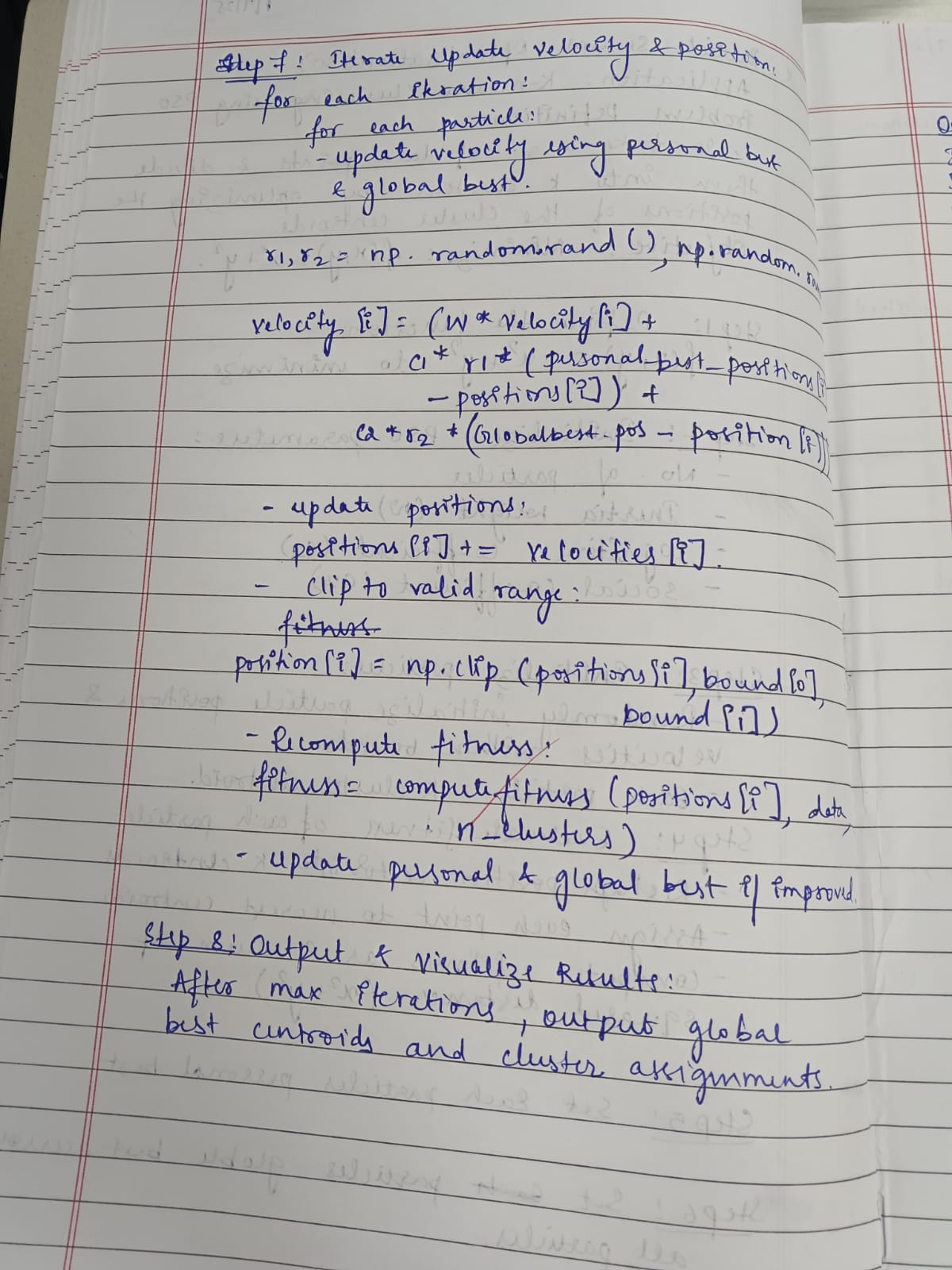
print("Predictions:", (preds > 0.5).astype(int).flatten())

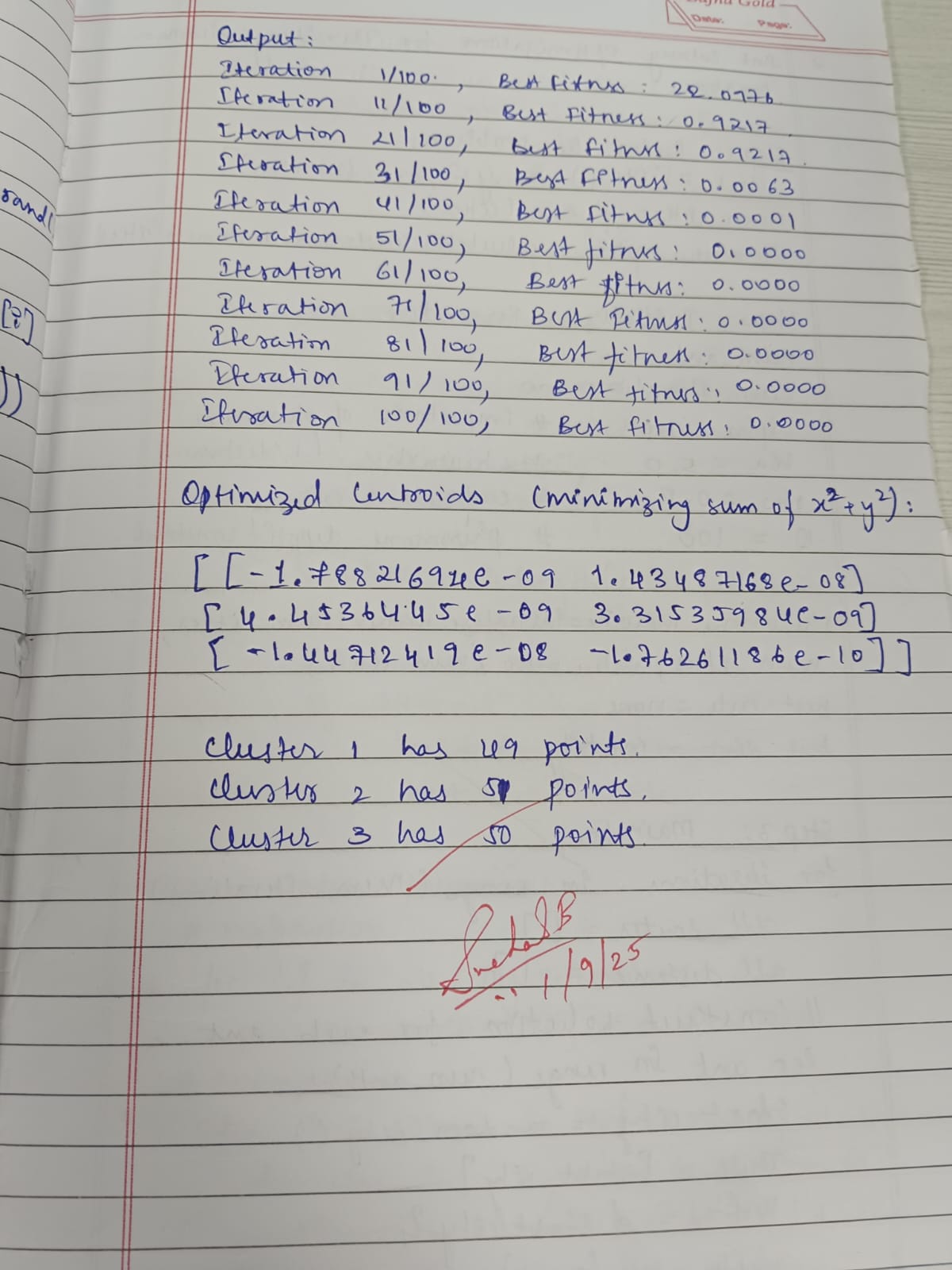
print("Expected :", y.flatten())

**Program 2**

Particle Swarm Optimization for Function Optimization for K-means clutering:

Algorithm:



Code:

import numpy as np

# Sample 2D data points (random clusters)

np.random.seed(0)

data = np.vstack([

np.random.randn(50, 2) + np.array([5, 5]),

np.random.randn(50, 2) + np.array([-5, -5]),

np.random.randn(50, 2) + np.array([5, -5])

])

K = 3 # Number of clusters

num\_particles = 30

max\_iter = 100

dim = 2 # x, y

class Particle:

def \_\_init\_\_(self):

# Each particle represents K centroids (K x dim)

self.position = np.random.uniform(low=np.min(data, axis=0),

high=np.max(data, axis=0),

size=(K, dim))

self.velocity = np.zeros((K, dim))

self.best\_position = self.position.copy()

self.best\_fitness = float('inf')

def update\_velocity(self, global\_best\_position, w=0.5, c1=1.5, c2=1.5):

r1 = np.random.rand(K, dim)

r2 = np.random.rand(K, dim)

cognitive = c1 \* r1 \* (self.best\_position - self.position)

social = c2 \* r2 \* (global\_best\_position - self.position)

self.velocity = w \* self.velocity + cognitive + social

def update\_position(self):

self.position += self.velocity

# Fitness function: sum of x^2 + y^2 of centroids

def fitness(centroids):

return np.sum(centroids[:, 0]\*\*2 + centroids[:, 1]\*\*2)

# PSO main function

def pso():

swarm = [Particle() for \_ in range(num\_particles)]

global\_best\_position = None

global\_best\_fitness = float('inf')

for iteration in range(max\_iter):

for particle in swarm:

fit = fitness(particle.position)

if fit < particle.best\_fitness:

particle.best\_fitness = fit

particle.best\_position = particle.position.copy()

if fit < global\_best\_fitness:

global\_best\_fitness = fit

global\_best\_position = particle.position.copy()

for particle in swarm:

particle.update\_velocity(global\_best\_position)

particle.update\_position()

if iteration % 10 == 0 or iteration == max\_iter - 1:

print(f"Iteration {iteration+1}/{max\_iter}, Best Fitness: {global\_best\_fitness:.4f}")

return global\_best\_position

# Run PSO

best\_centroids = pso()

print("\nOptimized centroids (minimizing sum of x^2 + y^2):\n", best\_centroids)

# Assign points to closest centroid

def assign\_clusters(centroids):

clusters = [[] for \_ in range(K)]

for point in data:

distances = np.linalg.norm(point - centroids, axis=1)

cluster\_idx = np.argmin(distances)

clusters[cluster\_idx].append(point)

return clusters

clusters = assign\_clusters(best\_centroids)

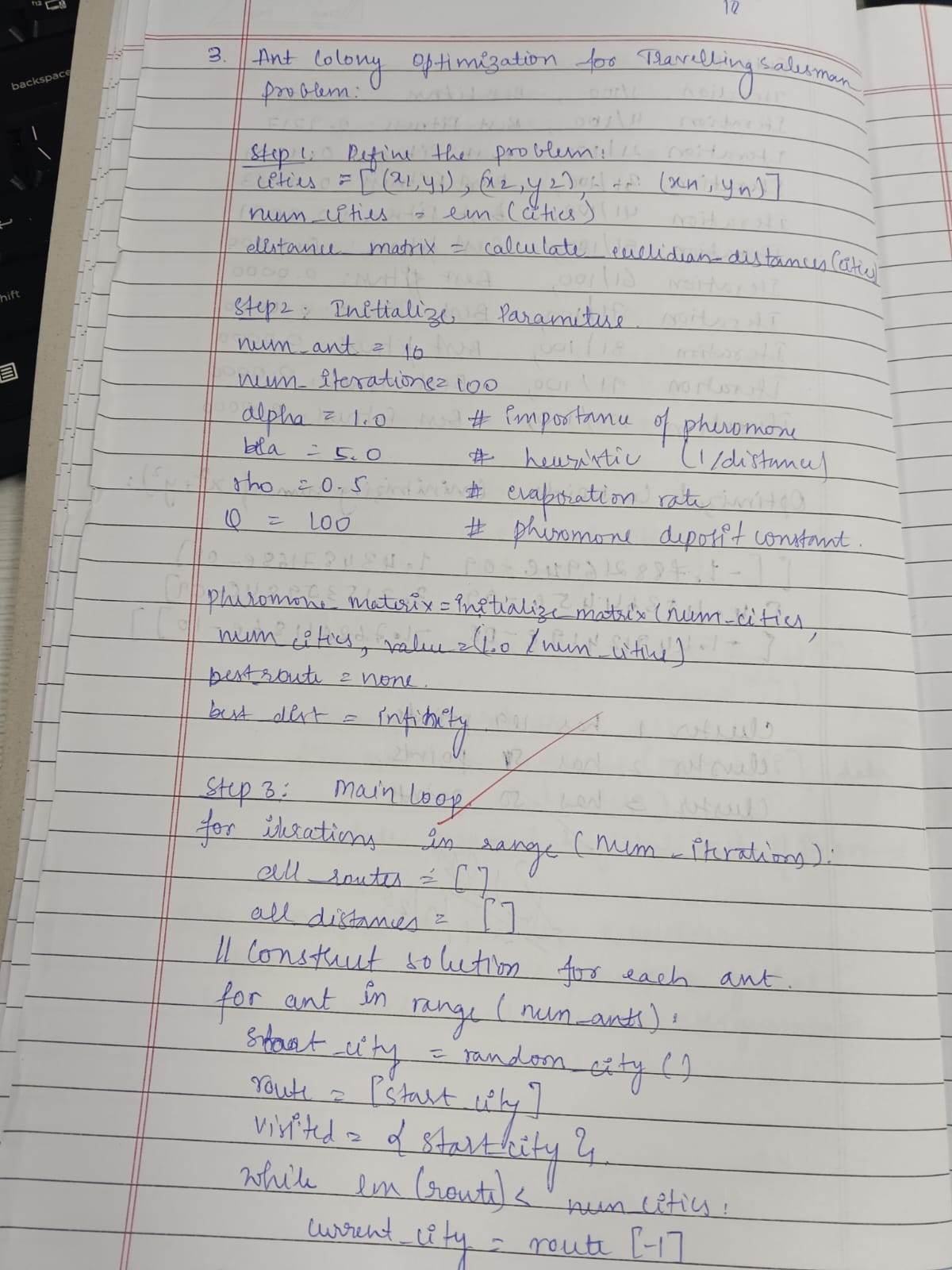
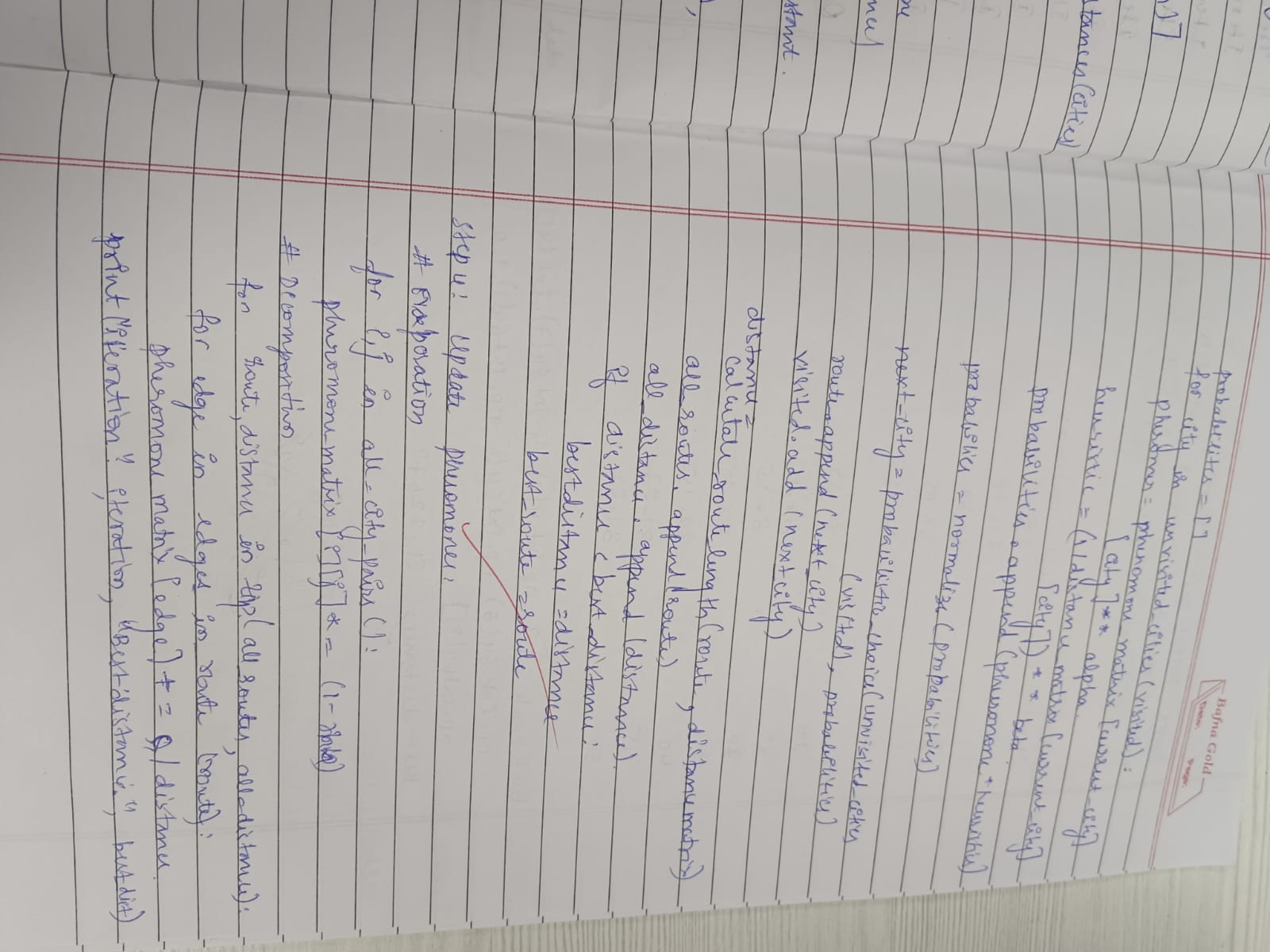
for i, cluster in enumerate(clusters):

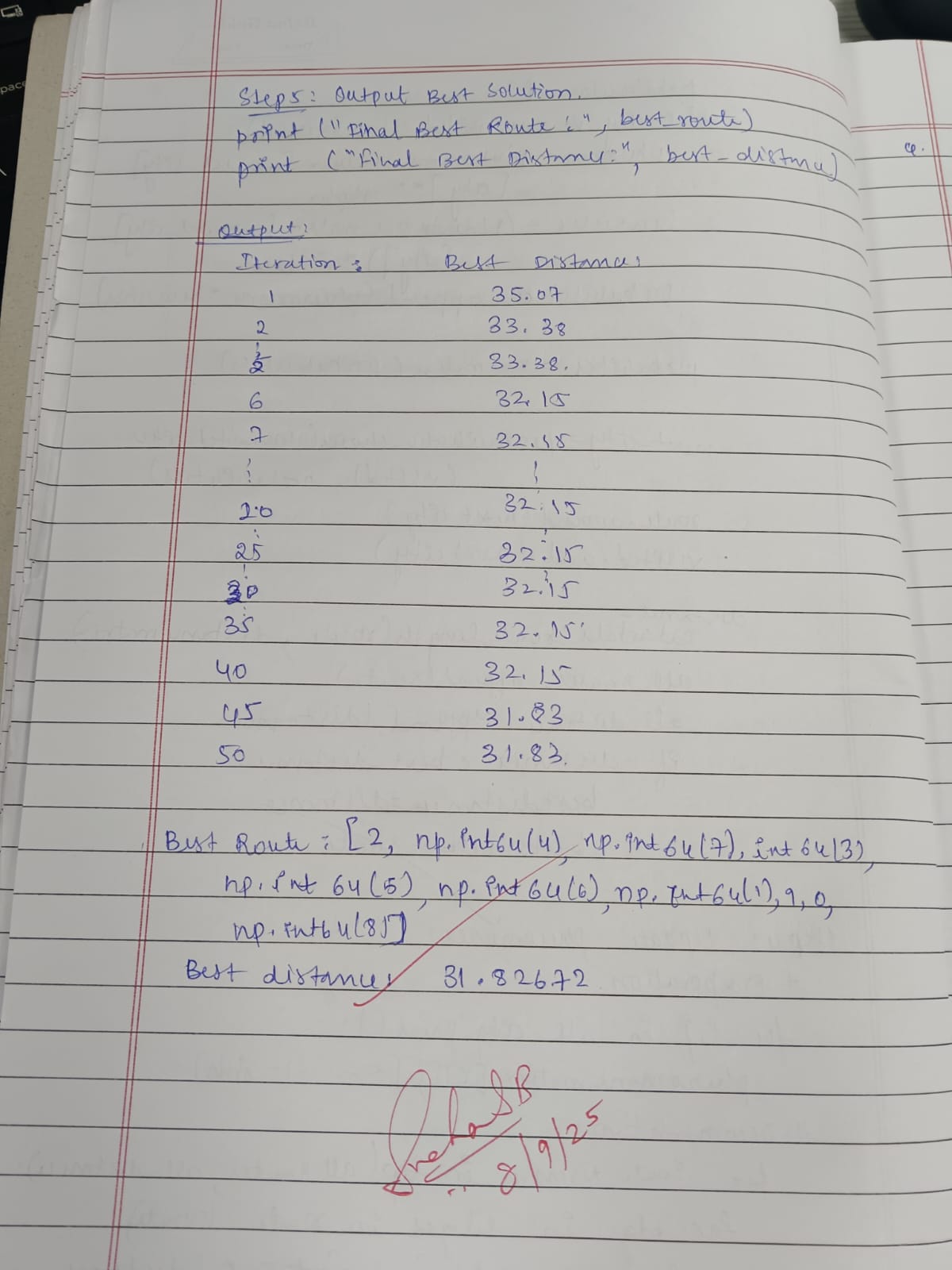
print(f"Cluster {i+1} has {len(cluster)} points.")

**Program 3**

Ant Colony Optimization for the Traveling Salesman Problem:

Algorithm:



Code:

"""

Ant Colony Optimization (ACO) for the Travelling Salesman Problem (TSP)

Usage:

- Requires: numpy, matplotlib

- Run: python3 ant\_colony\_tsp.py

This script generates a random set of cities, runs ACO, and plots

the best tour found.

Parameters can be tuned at the top of the file.

"""

import math

import random

import numpy as np

import matplotlib.pyplot as plt

def euclidean\_distance(a, b):

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

class AntColonyTSP:

def \_\_init\_\_(

self,

cities,

num\_ants=20,

num\_iterations=200,

alpha=1.0,

beta=5.0,

rho=0.5,

q=100.0,

initial\_pheromone=1.0,

):

self.cities = cities

self.n = len(cities)

self.num\_ants = num\_ants

self.num\_iterations = num\_iterations

self.alpha = alpha

self.beta = beta

self.rho = rho

self.q = q

# distance matrix

self.dist = np.zeros((self.n, self.n), dtype=float)

for i in range(self.n):

for j in range(self.n):

if i == j:

self.dist[i, j] = 1e-9

else:

self.dist[i, j] = euclidean\_distance(cities[i], cities[j])

# pheromone matrix

self.pheromone = np.full((self.n, self.n), initial\_pheromone, dtype=float)

# heuristic matrix (eta = 1/d)

self.eta = 1.0 / self.dist

# best solution

self.best\_tour = None

self.best\_length = float("inf")

def \_select\_next\_city(self, current, visited):

allowed = [i for i in range(self.n) if i not in visited]

if not allowed:

return None

pheromone = np.take(self.pheromone[current], allowed)

heuristic = np.take(self.eta[current], allowed)

numerator = (pheromone \*\* self.alpha) \* (heuristic \*\* self.beta)

if numerator.sum() == 0:

# fallback: choose uniformly

return random.choice(allowed)

probs = numerator / numerator.sum()

next\_city = random.choices(allowed, weights=probs, k=1)[0]

return next\_city

def \_construct\_solutions(self):

ants\_tours = []

ants\_lengths = []

for \_ in range(self.num\_ants):

start = random.randrange(self.n)

tour = [start]

while len(tour) < self.n:

current = tour[-1]

next\_city = self.\_select\_next\_city(current, tour)

if next\_city is None:

break

tour.append(next\_city)

# complete tour (return to start)

length = 0.0

for i in range(len(tour)):

j = (i + 1) % len(tour)

length += self.dist[tour[i], tour[j]]

ants\_tours.append(tour)

ants\_lengths.append(length)

return ants\_tours, ants\_lengths

def \_update\_pheromones(self, ants\_tours, ants\_lengths):

# evaporate

self.pheromone \*= (1.0 - self.rho)

# deposit

for tour, length in zip(ants\_tours, ants\_lengths):

contribution = self.q / (length + 1e-12)

for i in range(len(tour)):

a = tour[i]

b = tour[(i + 1) % len(tour)]

self.pheromone[a, b] += contribution

self.pheromone[b, a] += contribution

def run(self, verbose=True):

for it in range(1, self.num\_iterations + 1):

ants\_tours, ants\_lengths = self.\_construct\_solutions()

# update best

for tour, length in zip(ants\_tours, ants\_lengths):

if length < self.best\_length:

self.best\_length = length

self.best\_tour = tour.copy()

# pheromone update

self.\_update\_pheromones(ants\_tours, ants\_lengths)

if verbose and (it % max(1, self.num\_iterations // 10) == 0):

print(f"Iteration {it}/{self.num\_iterations} - best length: {self.best\_length:.4f}")

return self.best\_tour, self.best\_length

# --- Example usage ---

if \_\_name\_\_ == "\_\_main\_\_":

random.seed(42)

np.random.seed(42)

# generate random cities (change to your own coordinates if needed)

num\_cities = 25

cities = [(random.random() \* 100, random.random() \* 100) for \_ in range(num\_cities)]

aco = AntColonyTSP(

cities,

num\_ants=50,

num\_iterations=300,

alpha=1.0,

beta=5.0,

rho=0.4,

q=100.0,

initial\_pheromone=1.0,

)

best\_tour, best\_length = aco.run(verbose=True)

print("\nBest length:", best\_length)

print("Best tour:", best\_tour)

# plot result

tour\_coords = [cities[i] for i in best\_tour] + [cities[best\_tour[0]]]

xs = [c[0] for c in tour\_coords]

ys = [c[1] for c in tour\_coords]

plt.figure(figsize=(8, 6))

plt.plot(xs, ys, marker="o")

for i, (x, y) in enumerate(cities):

plt.text(x + 0.8, y + 0.8, str(i), fontsize=9)

plt.title(f"ACO TSP - best length: {best\_length:.4f}")

plt.xlabel("X")

plt.ylabel("Y")

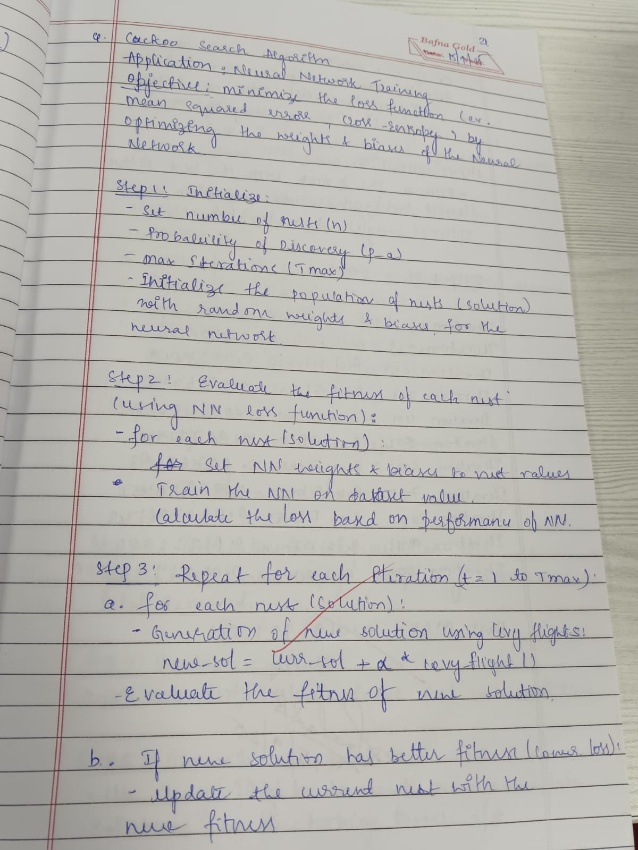
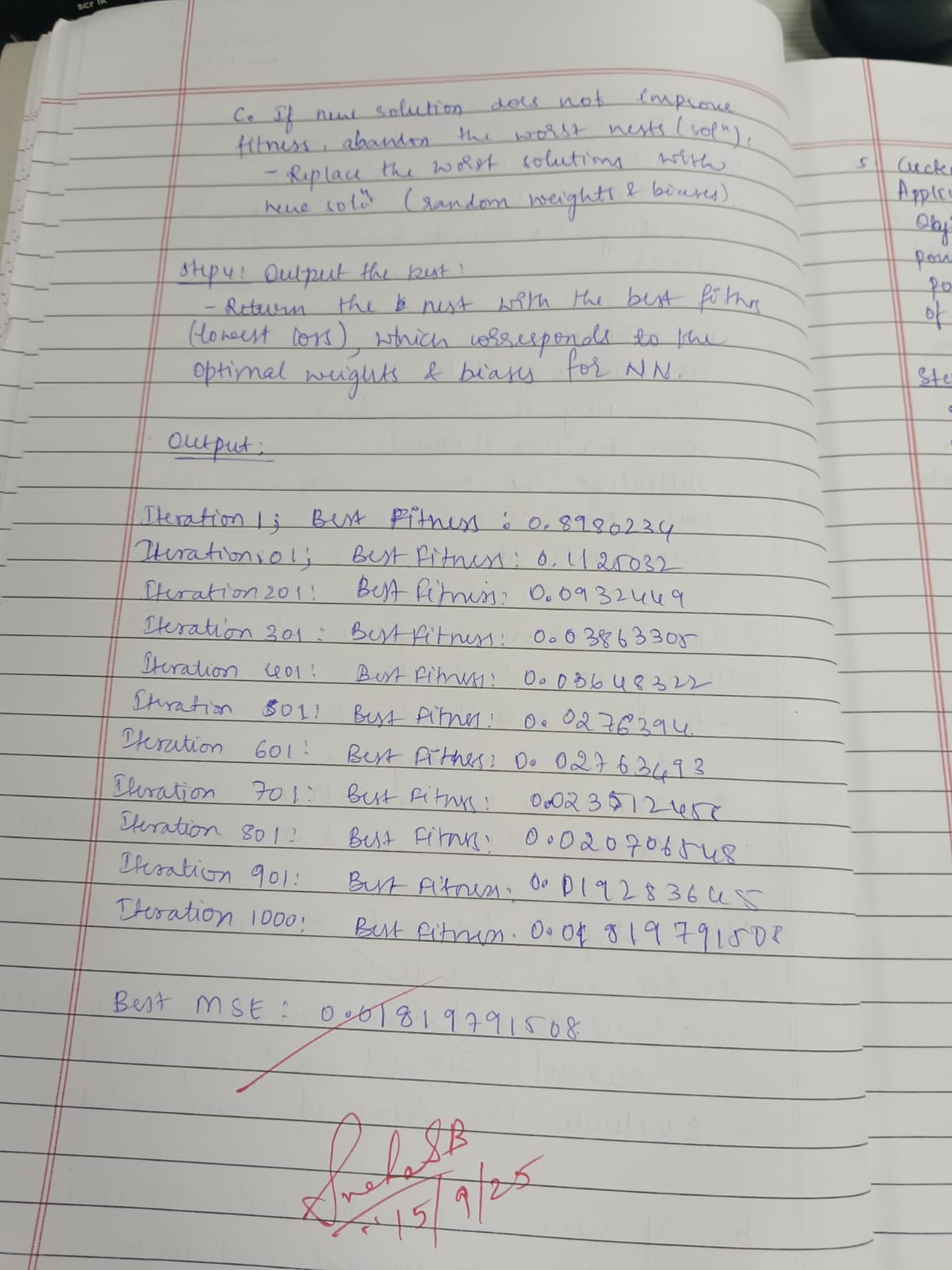
plt.grid(True)

plt.show()

**Program 4**

Cuckoo search algorithm : Power system and energy

Algorithm:

Code:

import numpy as np

import math

# Generator data: cost coefficients a, b, c, min and max power limits

generators = [

{'a': 500, 'b': 5.3, 'c': 0.004, 'Pmin': 150, 'Pmax': 600},

{'a': 400, 'b': 5.5, 'c': 0.006, 'Pmin': 100, 'Pmax': 400},

{'a': 200, 'b': 5.8, 'c': 0.009, 'Pmin': 50, 'Pmax': 200}

]

total\_demand = 850 # MW

num\_generators = len(generators)

num\_nests = 30

max\_iter = 100

pa = 0.25 # discovery rate

def fuel\_cost(P, gen):

return gen['a'] + gen['b'] \* P + gen['c'] \* P\*\*2

def is\_feasible(power\_outputs):

# Check limits

for i, P in enumerate(power\_outputs):

if P < generators[i]['Pmin'] or P > generators[i]['Pmax']:

return False

# Check power balance

if abs(np.sum(power\_outputs) - total\_demand) > 1e-3:

return False

return True

def penalty(power\_outputs):

# Penalty for constraint violations

penalty\_val = 0

# Power limits penalty

for i, P in enumerate(power\_outputs):

if P < generators[i]['Pmin']:

penalty\_val += 1e5 \* (generators[i]['Pmin'] - P)\*\*2

elif P > generators[i]['Pmax']:

penalty\_val += 1e5 \* (P - generators[i]['Pmax'])\*\*2

# Power balance penalty

diff = np.sum(power\_outputs) - total\_demand

penalty\_val += 1e6 \* (diff)\*\*2

return penalty\_val

def fitness(power\_outputs):

cost = 0

for i, P in enumerate(power\_outputs):

cost += fuel\_cost(P, generators[i])

cost += penalty(power\_outputs)

return cost

def levy\_flight(Lambda=1.5, size=1):

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

(math.gamma((1 + Lambda) / 2) \* Lambda \* 2 \*\* ((Lambda - 1) / 2))) \*\* (1 / Lambda)

u = np.random.normal(0, sigma, size=size)

v = np.random.normal(0, 1, size=size)

step = u / (np.abs(v) \*\* (1 / Lambda))

return step

def generate\_feasible\_solution():

# Randomly allocate power to generators respecting limits and total demand

P = np.zeros(num\_generators)

remaining = total\_demand

for i in range(num\_generators - 1):

Pmin = generators[i]['Pmin']

Pmax = min(generators[i]['Pmax'], remaining - sum(g['Pmin'] for g in generators[i+1:]))

P[i] = np.random.uniform(Pmin, Pmax)

remaining -= P[i]

# Last generator takes remaining power

P[-1] = remaining

# If last generator violates limits, adjust randomly (simple fix)

if P[-1] < generators[-1]['Pmin']:

P[-1] = generators[-1]['Pmin']

elif P[-1] > generators[-1]['Pmax']:

P[-1] = generators[-1]['Pmax']

return P

def cuckoo\_search\_eld():

nests = np.array([generate\_feasible\_solution() for \_ in range(num\_nests)])

fitnesses = np.array([fitness(nest) for nest in nests])

best\_index = np.argmin(fitnesses)

best\_nest = nests[best\_index].copy()

best\_fitness = fitnesses[best\_index]

for t in range(max\_iter):

for i in range(num\_nests):

step = levy\_flight(size=num\_generators)

new\_solution = nests[i] + step \* np.random.randn(num\_generators)

# Ensure limits and power balance

new\_solution = np.clip(new\_solution,

[g['Pmin'] for g in generators],

[g['Pmax'] for g in generators])

# Adjust power to match demand (simple proportional scaling)

diff = np.sum(new\_solution) - total\_demand

new\_solution -= diff / num\_generators

new\_fitness = fitness(new\_solution)

if new\_fitness < fitnesses[i]:

nests[i] = new\_solution

fitnesses[i] = new\_fitness

if new\_fitness < best\_fitness:

best\_fitness = new\_fitness

best\_nest = new\_solution.copy()

# Abandon fraction pa of worst nests

abandon\_count = int(pa \* num\_nests)

worst\_indices = np.argsort(fitnesses)[-abandon\_count:]

for idx in worst\_indices:

nests[idx] = generate\_feasible\_solution()

fitnesses[idx] = fitness(nests[idx])

# Update best solution if improved

current\_best\_idx = np.argmin(fitnesses)

if fitnesses[current\_best\_idx] < best\_fitness:

best\_fitness = fitnesses[current\_best\_idx]

best\_nest = nests[current\_best\_idx].copy()

return best\_nest, best\_fitness

# Run the algorithm

best\_solution, best\_cost = cuckoo\_search\_eld()

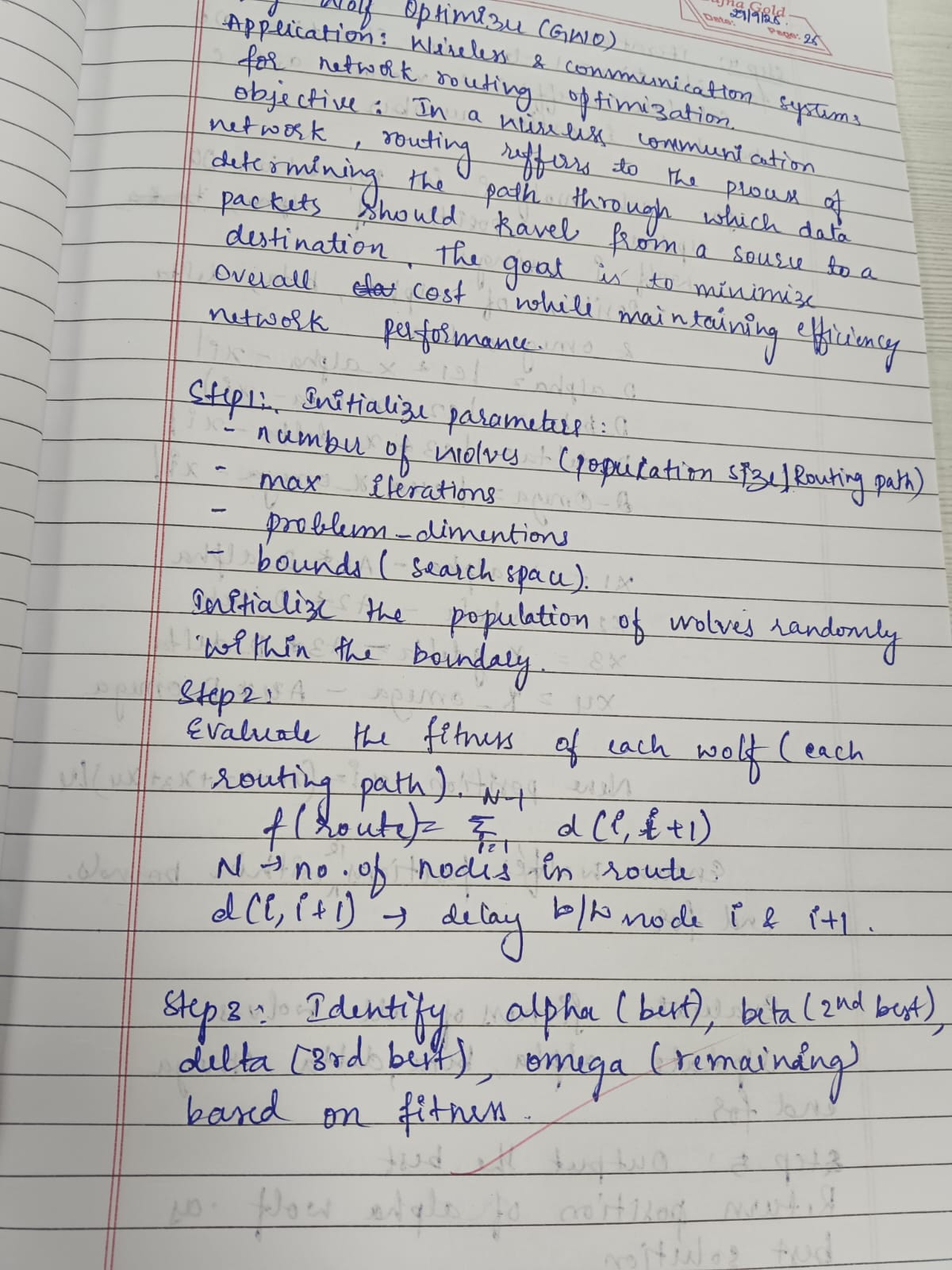
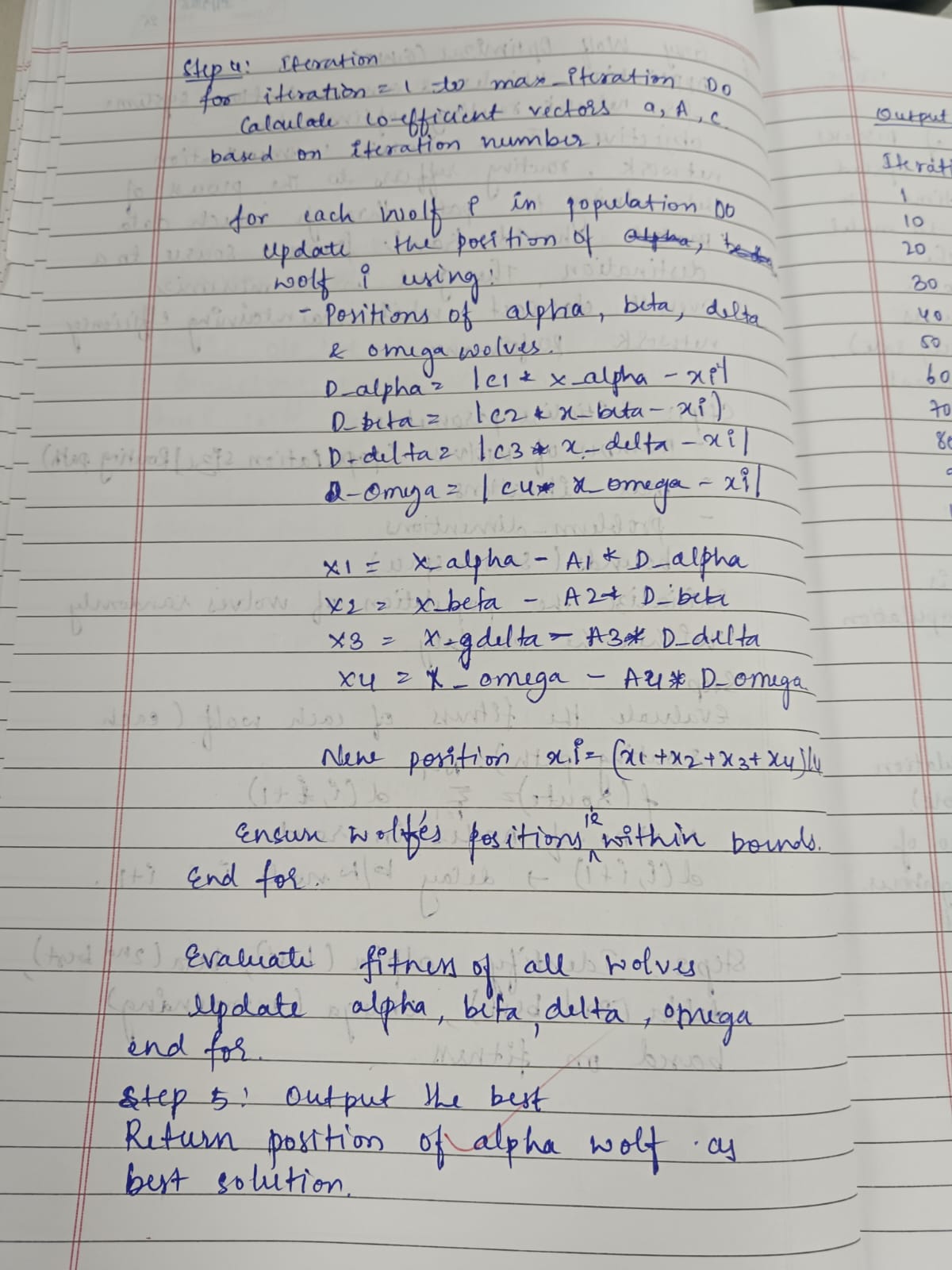
print("Optimal Power Outputs (MW):", best\_solution)

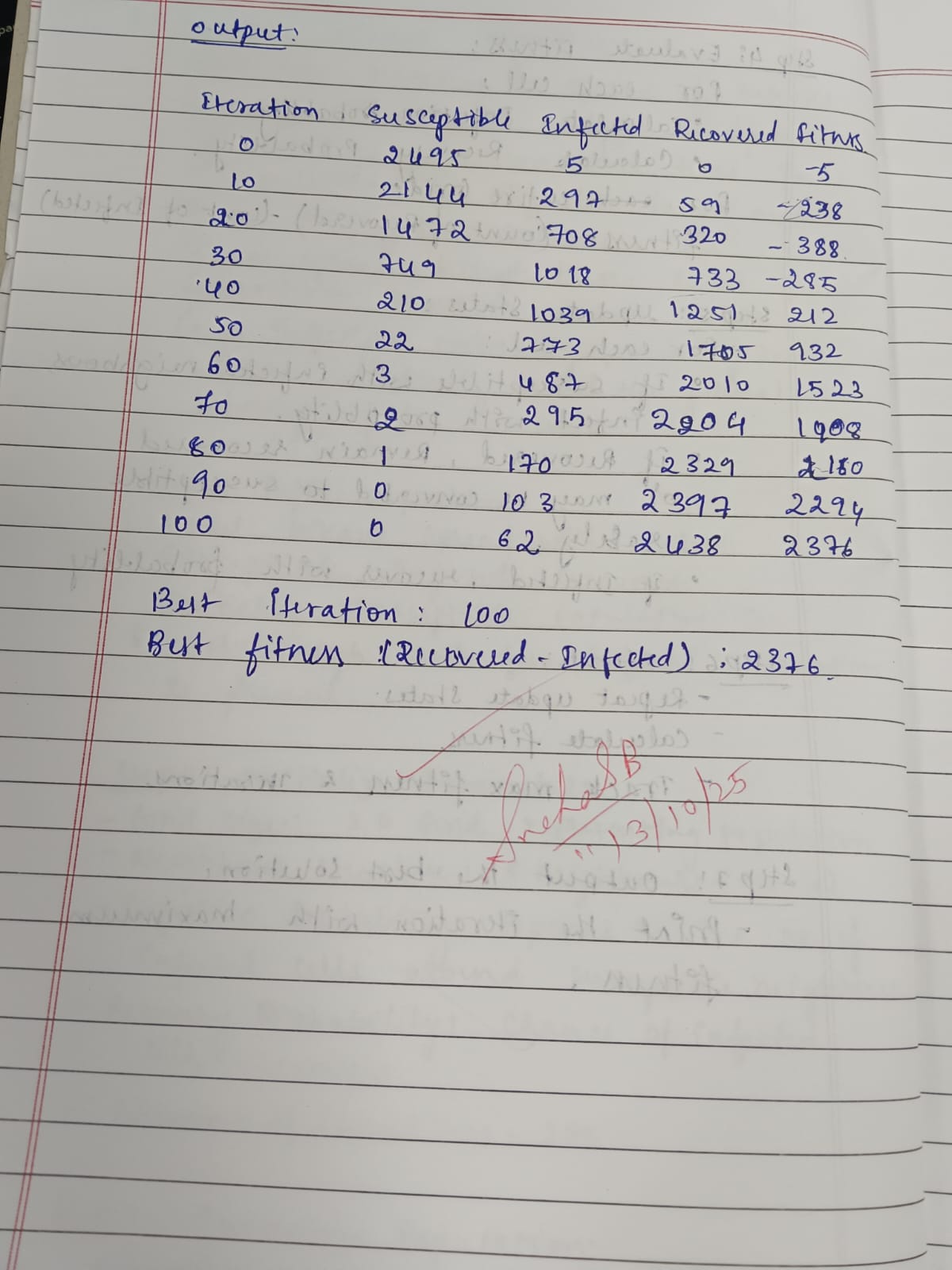
print("Minimum Fuel Cost:", best\_cost)

**Program 5**

Grey wolf optimization (Application: network routing optimization)

Algorithm:



Code:

import numpy as np

# Objective function (example):

# Let's simulate a function that depends on multiple routing parameters.

def network\_routing\_cost(x):

# x is a vector representing routing parameters

# For demonstration, use a nonlinear function:

return np.sum(x\*\*2) + 10 \* np.sum(np.sin(x))

# GWO Implementation with Alpha, Beta, Delta, Omega wolves

class GreyWolfOptimizer:

def \_\_init\_\_(self, obj\_func, dim, bounds, num\_wolves=30, max\_iter=100):

self.obj\_func = obj\_func

self.dim = dim

self.lb, self.ub = bounds

self.num\_wolves = num\_wolves

self.max\_iter = max\_iter

# Initialize wolf positions

self.positions = np.random.uniform(self.lb, self.ub, (num\_wolves, dim))

# Initialize alpha, beta, delta, omega

self.alpha\_pos = np.zeros(dim)

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

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

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

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

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

self.omega\_positions = np.zeros((num\_wolves - 3, dim))

self.omega\_scores = np.ones(num\_wolves - 3) \* float('inf')

def update\_wolves(self):

# Evaluate fitness of all wolves and update alpha, beta, delta, omega

fitness\_scores = np.array([self.obj\_func(pos) for pos in self.positions])

# Sort wolves by fitness scores (ascending)

sorted\_indices = np.argsort(fitness\_scores)

sorted\_positions = self.positions[sorted\_indices]

sorted\_scores = fitness\_scores[sorted\_indices]

# Alpha, Beta, Delta

self.alpha\_pos = sorted\_positions[0]

self.alpha\_score = sorted\_scores[0]

self.beta\_pos = sorted\_positions[1]

self.beta\_score = sorted\_scores[1]

self.delta\_pos = sorted\_positions[2]

self.delta\_score = sorted\_scores[2]

# Omega wolves (remaining wolves)

self.omega\_positions = sorted\_positions[3:]

self.omega\_scores = sorted\_scores[3:]

def optimize(self):

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

for i in range(self.num\_wolves):

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

# Update the positions based on alpha, beta, delta, omega wolves

a = 2 - t \* (2 / self.max\_iter) # a decreases linearly from 2 to 0

for j in range(self.dim):

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

A1 = 2 \* a \* r1 - a

C1 = 2 \* r2

D\_alpha = abs(C1 \* self.alpha\_pos[j] - self.positions[i][j])

X1 = self.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 \* self.beta\_pos[j] - self.positions[i][j])

X2 = self.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 \* self.delta\_pos[j] - self.positions[i][j])

X3 = self.delta\_pos[j] - A3 \* D\_delta

# Omega wolf influence

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

A4 = 2 \* a \* r1 - a

C4 = 2 \* r2

omega\_idx = np.random.randint(self.num\_wolves - 3)

D\_omega = abs(C4 \* self.omega\_positions[omega\_idx][j] - self.positions[i][j])

X4 = self.omega\_positions[omega\_idx][j] - A4 \* D\_omega

# Update wolf position

self.positions[i][j] = (X1 + X2 + X3 + X4) / 4

# Boundary check

self.positions[i] = np.clip(self.positions[i], self.lb, self.ub)

# Evaluate fitness of all wolves and update alpha, beta, delta, omega

self.update\_wolves()

# Optionally print progress

if (t+1) % 10 == 0 or t == 0:

print(f"Iteration {t+1}, Best Fitness: {self.alpha\_score:.4f}")

return self.alpha\_pos, self.alpha\_score

# Problem setup

dimension = 5 # For example, 5 routing parameters

bounds = (-10, 10) # Limits on parameters

gwo = GreyWolfOptimizer(network\_routing\_cost, dimension, bounds, num\_wolves=20, max\_iter=100)

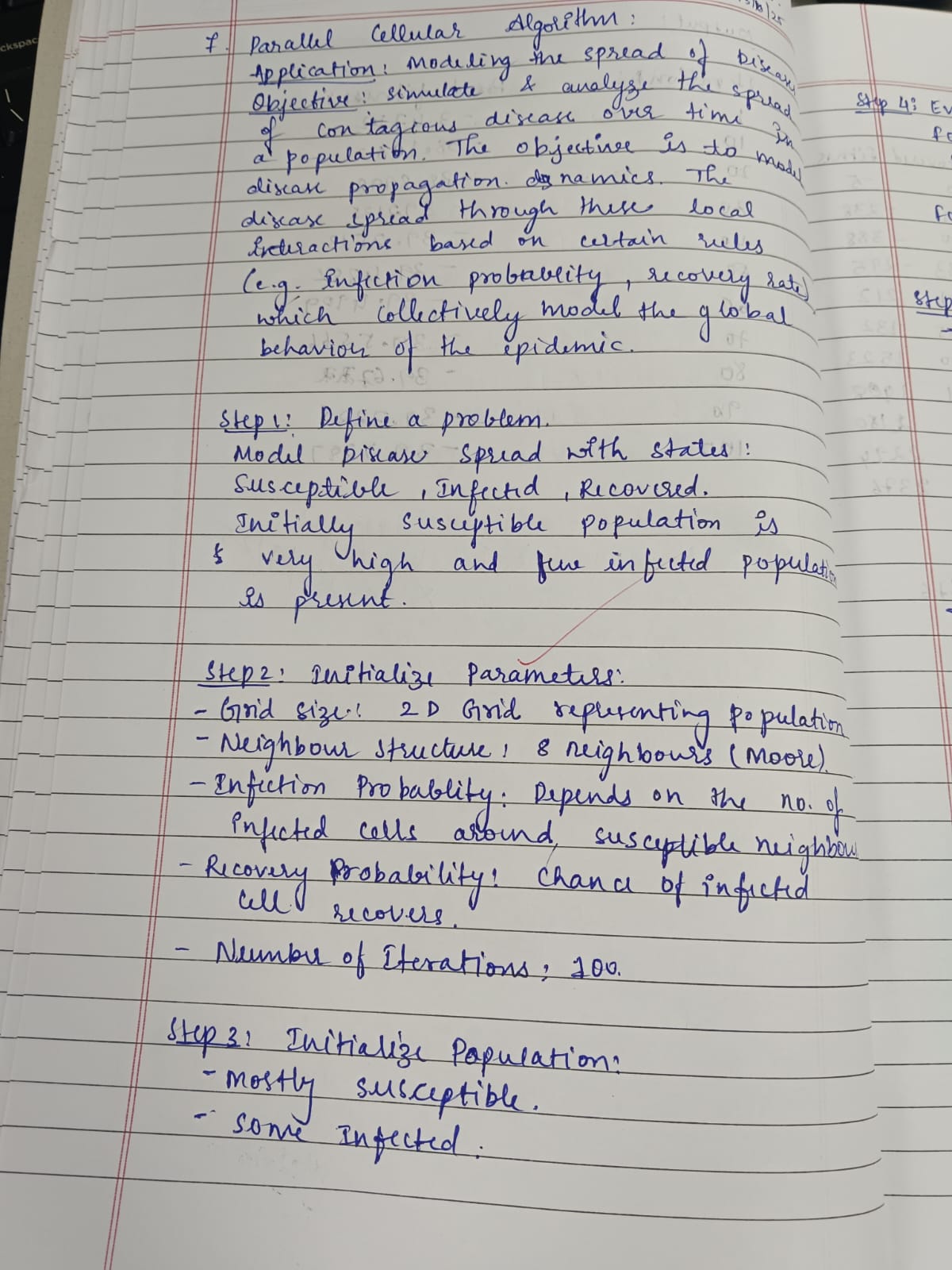
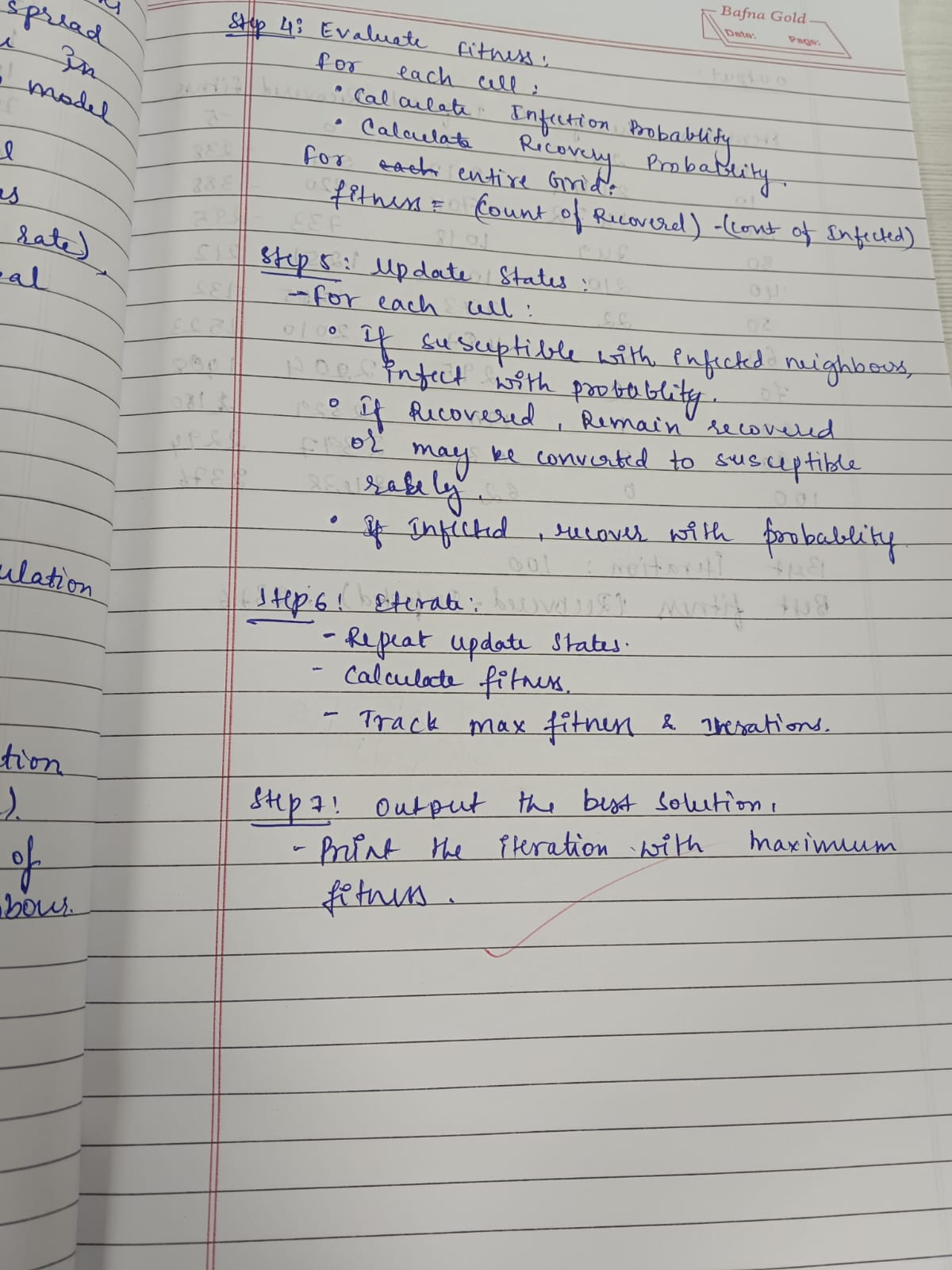
best\_solution, best\_cost = gwo.optimize()

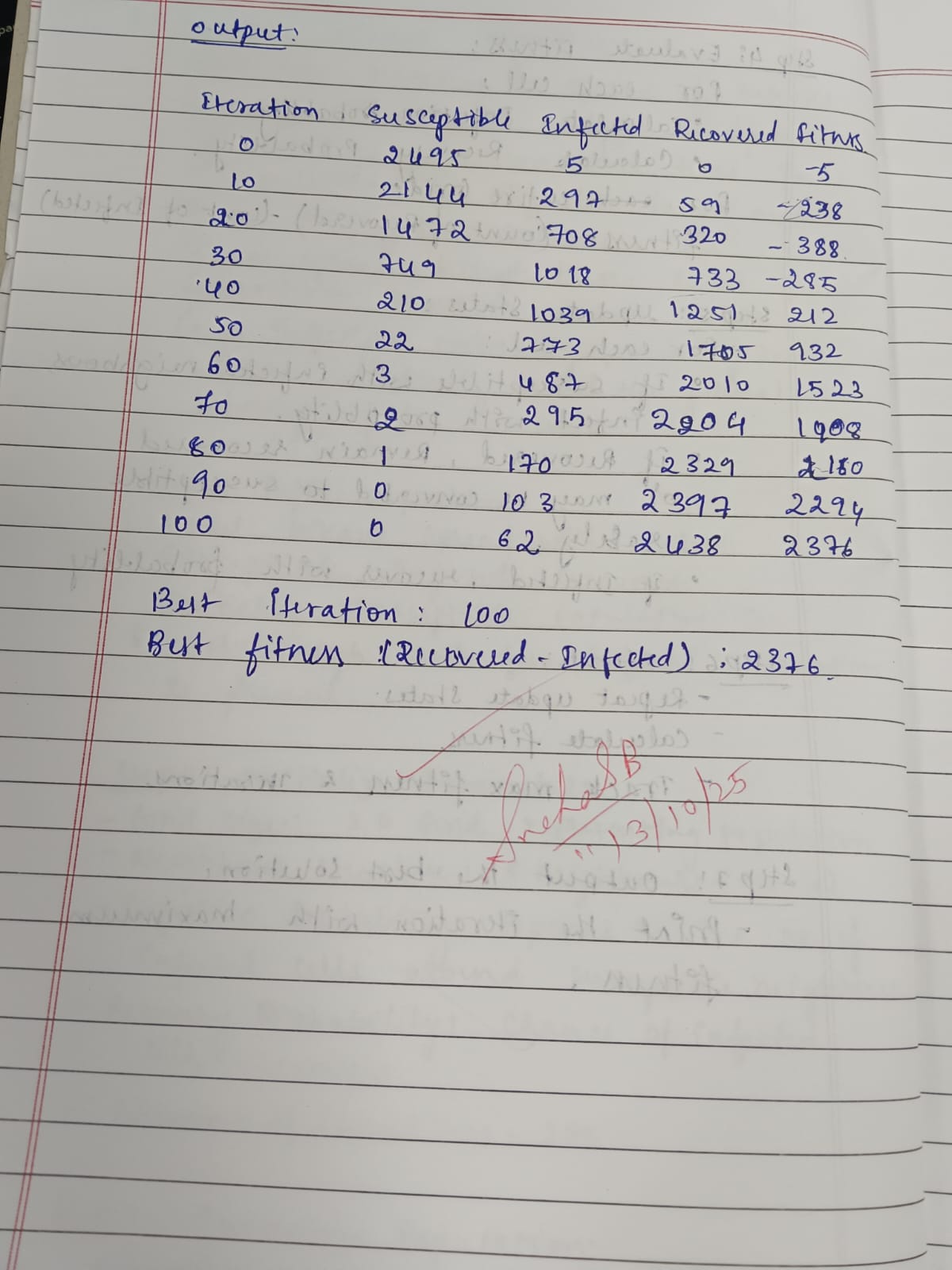
print("\nBest routing parameters found:", best\_solution)

print("Minimum routing cost:", best\_cost)

**Program 6**

Parrelel Cellular Algorithm for modeling the spread of diseases.Algorithm:



Code:

import numpy as np

from multiprocessing import Process, Pipe

import matplotlib.pyplot as plt

from matplotlib import animation

import time

GRID\_SIZE = (200, 200)

P\_INFECT = 0.35

P\_RECOVER = 0.02

NEIGHBORHOOD = 'moore'

PROCESSES = 4

TIMESTEPS = 400

INITIAL\_INFECTED = 10

EXTERNAL\_INFECTION\_RATE = 0.0001

SUSCEPTIBLE = 0

INFECTED = 1

RECOVERED = 2

def init\_grid(rows, cols, initial\_infected=10):

grid = np.zeros((rows, cols), dtype=np.uint8)

rs = np.random.randint(0, rows, size=initial\_infected)

cs = np.random.randint(0, cols, size=initial\_infected)

grid[rs, cs] = INFECTED

return grid

def neighbor\_kernel(neighborhood='moore'):

if neighborhood == 'moore':

return np.array([[1,1,1],[1,0,1],[1,1,1]], dtype=np.uint8)

else:

return np.array([[0,1,0],[1,0,1],[0,1,0]], dtype=np.uint8)

def count\_infected\_neighbors(subgrid, kernel):

r, c = subgrid.shape

out = np.zeros((r-2, c-2), dtype=np.uint8)

for dr in range(3):

for dc in range(3):

if kernel[dr, dc]:

out += (subgrid[dr:r-2+dr, dc:c-2+dc] == INFECTED)

return out

def worker(child\_conn, rows\_slice, cols, kernel, params):

p\_infect, p\_recover, ext\_rate = params

while True:

msg = child\_conn.recv()

if msg == 'STOP':

break

halo = msg

infected\_neighbors = count\_infected\_neighbors(halo, kernel)

inner = halo[1:-1, 1:-1]

next\_inner = inner.copy()

susceptible\_mask = (inner == SUSCEPTIBLE)

k = infected\_neighbors.astype(np.float64)

prob\_infection = 1.0 - np.power((1.0 - p\_infect), k)

randvals = np.random.random(size=prob\_infection.shape)

new\_infections = susceptible\_mask & (randvals < prob\_infection)

ext\_rand = np.random.random(size=prob\_infection.shape)

new\_infections |= susceptible\_mask & (ext\_rand < ext\_rate)

next\_inner[new\_infections] = INFECTED

infected\_mask = (inner == INFECTED)

rec\_rand = np.random.random(size=infected\_mask.shape)

recoveries = infected\_mask & (rec\_rand < p\_recover)

next\_inner[recoveries] = RECOVERED

child\_conn.send(next\_inner)

class ParallelCellularSIR:

def \_\_init\_\_(self, rows, cols, p\_infect, p\_recover, neighborhood='moore', processes=4, external\_rate=0.0):

self.rows = rows

self.cols = cols

self.p\_infect = p\_infect

self.p\_recover = p\_recover

self.neighborhood = neighborhood

self.kernel = neighbor\_kernel(neighborhood)

self.processes = min(processes, rows)

self.external\_rate = external\_rate

self.grid = init\_grid(rows, cols, INITIAL\_INFECTED)

sizes = [rows // self.processes + (1 if i < rows % self.processes else 0) for i in range(self.processes)]

starts = [sum(sizes[:i]) for i in range(self.processes)]

ends = [starts[i] + sizes[i] for i in range(self.processes)]

self.slices = list(zip(starts, ends))

self.parents = []

self.workers = []

for s, e in self.slices:

parent\_conn, child\_conn = Pipe()

p = Process(target=worker, args=(child\_conn, (s, e), cols, self.kernel, (self.p\_infect, self.p\_recover, self.external\_rate)))

p.daemon = True

p.start()

self.parents.append(parent\_conn)

self.workers.append(p)

def step(self):

for i, (s, e) in enumerate(self.slices):

top = max(0, s-1)

bottom = min(self.rows, e+1)

sub = self.grid[top:bottom, :]

if top == 0:

sub = np.vstack((np.zeros((1, self.cols), dtype=np.uint8), sub))

if bottom == self.rows:

sub = np.vstack((sub, np.zeros((1, self.cols), dtype=np.uint8)))

left\_col = self.grid[top:bottom, -1].reshape(-1,1)

right\_col = self.grid[top:bottom, 0].reshape(-1,1)

sub = np.hstack((left\_col, sub, right\_col))

self.parents[i].send(sub)

next\_grid = np.zeros\_like(self.grid)

for i, (s, e) in enumerate(self.slices):

inner = self.parents[i].recv()

next\_grid[s:e, :] = inner

self.grid = next\_grid

def run(self, timesteps=100, verbose=True, animate=False):

if animate:

fig, ax = plt.subplots(figsize=(6,6))

img = ax.imshow(self.grid, vmin=0, vmax=2, interpolation='nearest')

ax.set\_title('Parallel Cellular SIR')

plt.axis('off')

def update(frame):

self.step()

img.set\_data(self.grid)

return (img,)

anim = animation.FuncAnimation(fig, update, frames=timesteps, interval=50, blit=True)

plt.show()

else:

for t in range(timesteps):

self.step()

if verbose and (t % max(1, timesteps//10) == 0):

unique, counts = np.unique(self.grid, return\_counts=True)

d = dict(zip(unique, counts))

s = d.get(SUSCEPTIBLE, 0)

i = d.get(INFECTED, 0)

r = d.get(RECOVERED, 0)

print(f"t={t}: S={s} I={i} R={r}")

def stop\_workers(self):

for p\_conn in self.parents:

p\_conn.send('STOP')

for w in self.workers:

w.join(timeout=1)

if \_\_name\_\_ == '\_\_main\_\_':

np.random.seed(1)

rows, cols = GRID\_SIZE

model = ParallelCellularSIR(rows, cols, P\_INFECT, P\_RECOVER, NEIGHBORHOOD, PROCESSES, EXTERNAL\_INFECTION\_RATE)

start = time.time()

try:

model.run(TIMESTEPS, verbose=True, animate=True)

finally:

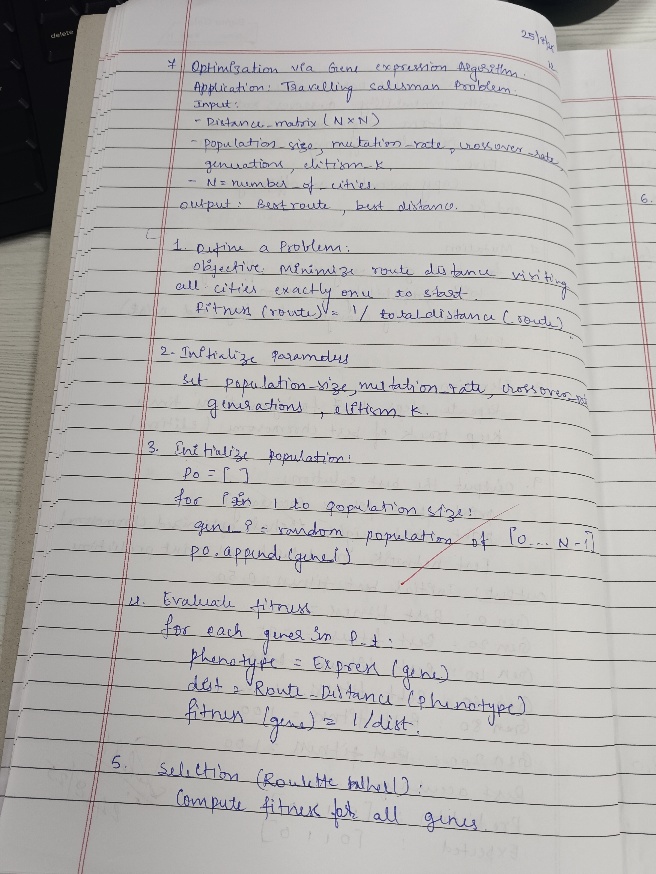
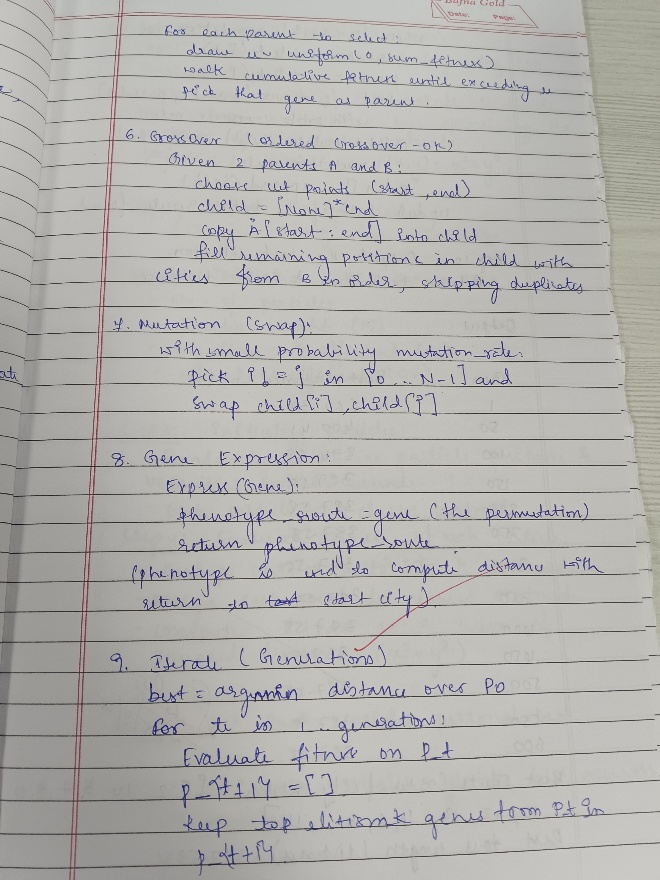
model.stop\_workers()

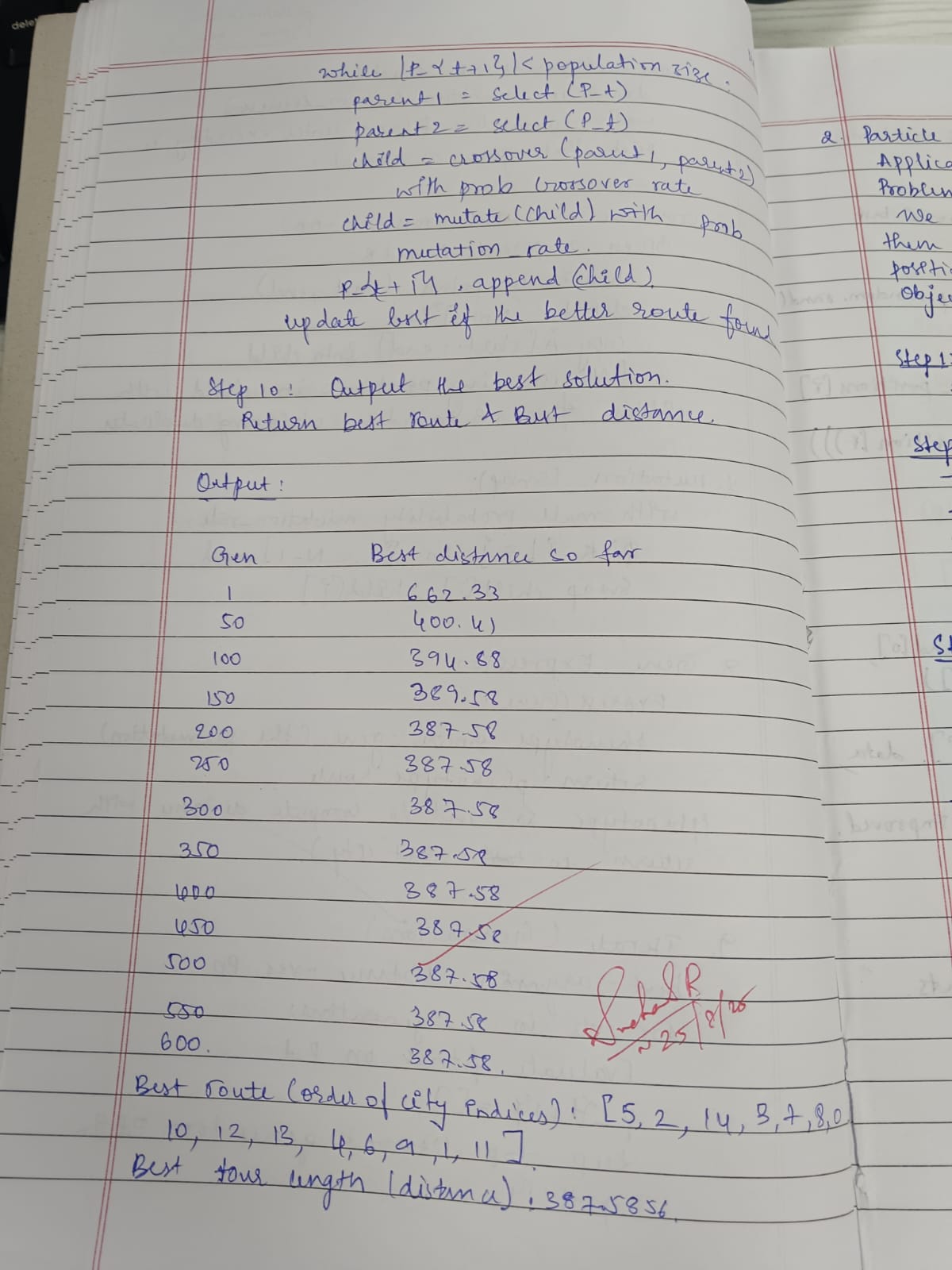
print('Elapsed:', time.time() - start)

**Program 7**

Optimization via Gene Expression Algorithms for TSP:

Algorithm:



Code:

import random

import math

from typing import List, Tuple, Optional

# -----------------------------

# Utility: Distance / Fitness

# -----------------------------

def route\_distance(route: List[int], distance\_matrix: List[List[float]]) -> float:

"""

Total cyclic route distance (returns to starting city).

route = permutation of city indices, e.g., [0,2,1,3]

"""

n = len(route)

total = 0.0

for i in range(n):

a = route[i]

b = route[(i + 1) % n] # wrap to start

total += distance\_matrix[a][b]

return total

def fitness\_of(route: List[int], distance\_matrix: List[List[float]]) -> float:

d = route\_distance(route, distance\_matrix)

# Avoid division by zero (shouldn't happen with positive distances)

return 1.0 / d if d > 0 else float("inf")

# -----------------------------

# Step 8: Gene Expression

# (For TSP, genotype == phenotype: permutation -> route)

# -----------------------------

def express\_gene\_to\_route(gene: List[int]) -> List[int]:

return gene # identity mapping for TSP

# -----------------------------

# Step 5: Selection (Roulette Wheel)

# -----------------------------

def roulette\_wheel\_selection(population: List[List[int]],

fitnesses: List[float]) -> List[int]:

total\_fit = sum(fitnesses)

# Fallback if all fitnesses are zero (shouldn't happen here)

if total\_fit == 0.0:

return random.choice(population)

pick = random.uniform(0, total\_fit)

current = 0.0

for ind, f in zip(population, fitnesses):

current += f

if current >= pick:

return ind

return population[-1]

# -----------------------------

# Step 6: Crossover (Ordered Crossover - OX)

# -----------------------------

def ordered\_crossover(parent1: List[int], parent2: List[int]) -> List[int]:

n = len(parent1)

start, end = sorted(random.sample(range(n), 2))

child = [None] \* n

# Copy slice from parent1

child[start:end] = parent1[start:end]

# Fill remaining from parent2 in order

ptr = end

for city in parent2:

if city not in child:

if ptr == n:

ptr = 0

child[ptr] = city

ptr += 1

return child

# -----------------------------

# Step 7: Mutation (Swap)

# -----------------------------

def swap\_mutation(route: List[int], mutation\_rate: float) -> List[int]:

child = route[:]

if random.random() < mutation\_rate:

i, j = random.sample(range(len(child)), 2)

child[i], child[j] = child[j], child[i]

return child

# -----------------------------

# Step 3: Initialize Population

# -----------------------------

def initialize\_population(num\_cities: int, population\_size: int) -> List[List[int]]:

return [random.sample(range(num\_cities), num\_cities)

for \_ in range(population\_size)]

# -----------------------------

# Step 9: Main GEA loop for TSP

# -----------------------------

def gea\_tsp(distance\_matrix: List[List[float]],

population\_size: int = 100,

generations: int = 500,

mutation\_rate: float = 0.10,

crossover\_rate: float = 0.90,

elitism\_k: int = 2,

seed: Optional[int] = 42) -> Tuple[List[int], float]:

"""

Solve TSP using a Gene Expression Algorithm.

Returns (best\_route, best\_distance).

"""

if seed is not None:

random.seed(seed)

num\_cities = len(distance\_matrix)

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

# Evaluate initial population

fitnesses = [fitness\_of(express\_gene\_to\_route(ind), distance\_matrix)

for ind in population]

# Track best

best\_idx = max(range(population\_size), key=lambda i: fitnesses[i])

best\_route = population[best\_idx][:]

best\_distance = route\_distance(best\_route, distance\_matrix)

for gen in range(1, generations + 1):

# ---- Elitism: take top-k directly into next population ----

ranked = sorted(range(population\_size),

key=lambda i: fitnesses[i],

reverse=True)

elites = [population[i][:] for i in ranked[:elitism\_k]]

# ---- Build next generation ----

next\_pop: List[List[int]] = elites[:]

while len(next\_pop) < population\_size:

# Step 5: Selection

p1 = roulette\_wheel\_selection(population, fitnesses)

p2 = roulette\_wheel\_selection(population, fitnesses)

# Step 6: Crossover

if random.random() < crossover\_rate:

child = ordered\_crossover(p1, p2)

else:

child = p1[:]

# Step 7: Mutation

child = swap\_mutation(child, mutation\_rate)

next\_pop.append(child)

population = next\_pop

# Step 4: Evaluate fitness of new population

fitnesses = [fitness\_of(express\_gene\_to\_route(ind), distance\_matrix)

for ind in population]

# Update global best

gen\_best\_idx = max(range(population\_size), key=lambda i: fitnesses[i])

gen\_best\_route = population[gen\_best\_idx]

gen\_best\_distance = route\_distance(gen\_best\_route, distance\_matrix)

if gen\_best\_distance < best\_distance:

best\_distance = gen\_best\_distance

best\_route = gen\_best\_route[:]

# Optional: progress log

if gen % 50 == 0 or gen == 1 or gen == generations:

print(f"Gen {gen:4d} | Best distance so far: {best\_distance:.4f}")

return best\_route, best\_distance

# -----------------------------

# Helper: Build a distance matrix from 2D coordinates (demo)

# -----------------------------

def euclidean\_distance\_matrix(points\_2d: List[Tuple[float, float]]) -> List[List[float]]:

n = len(points\_2d)

dist = [[0.0]\*n for \_ in range(n)]

for i in range(n):

x1, y1 = points\_2d[i]

for j in range(n):

if i == j:

continue

x2, y2 = points\_2d[j]

dx = x1 - x2

dy = y1 - y2

dist[i][j] = math.hypot(dx, dy)

return dist

# -----------------------------

# Example usage / quick test

# -----------------------------

if \_\_name\_\_ == "\_\_main\_\_":

# Example 1: fixed small distance matrix (5 cities)

# distance\_matrix = [

# [0, 2, 9, 10, 7],

# [1, 0, 6, 4, 3],

# [15, 7, 0, 8, 3],

# [6, 3, 12, 0, 11],

# [9, 7, 5, 6, 0]

# ]

# Example 2: random points in 2D plane (visualizable case)

random.seed(1)

points = [(random.uniform(0, 100), random.uniform(0, 100)) for \_ in range(15)]

distance\_matrix = euclidean\_distance\_matrix(points)

best\_route, best\_distance = gea\_tsp(

distance\_matrix,

population\_size=120,

generations=600,

mutation\_rate=0.12,

crossover\_rate=0.90,

elitism\_k=3,

seed=7

)

print("\nBest route (order of city indices):", best\_route)

print("Best tour length (distance):", round(best\_distance, 4))