VISVESVARAYA TECHNOLOGICAL UNIVERSITY

"JnanaSangama", Belgaum -590014, Karnataka.

LAB RECORD

Bio Inspired Systems (23CS5BSBIS)

Submitted by

Abhinav C (1BM23CS008)

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 **Abhinav C (1BM23CS008),** 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.

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

Index

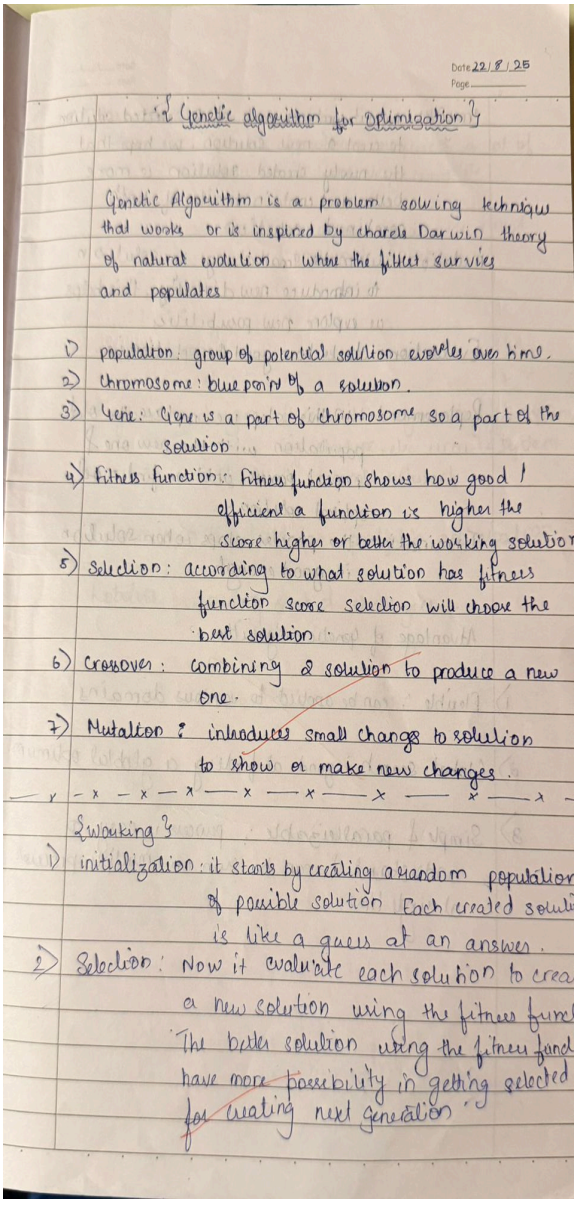
Github Link:

Abhinav-2013/BIS_LAB

Program 1

Apply Genetic Algorithm to solve the binary (0/1) Knapsack Problem

Algoithm:



Date 22/8/25
Page

{ Genetic algorithm for Optimization }

Genetic Algorithm is a problem solving technique that works or is inspired by charels Darwin theory of natural evolution where the fittest survies and populates

1) population: group of potential solution evolves over time.
2) chromosome: blue print of a solution.
3) Gene: Gene is a part of chromosome so a part of the solution.
4) Fitness function: Fitness function shows how good / efficient a function is higher the score higher or better the working solution
5) Selection: according to what solution has fitness function score selection will choose the best solution.
6) Crossover: combining 2 solution to produce a new one.
7) Mutation: introduces small changes to solution to show or make new changes.

{ working }

1) initialization: it starts by creating a random population of possible solution Each created solu is like a guess at an answer.
2) Selection: Now it evaluate each solution to crea a new solution using the fitness func The better solution using the fitness funct have more possibility in getting selected for creating next generation.

Date __/__/__
Page _____

CrossOver : Now it combines parts of 2 selected solution to creat a new solution. we hope that the newly created solution is more better and contain traits of both combined solution

mutation : We make small change in solution to introduce new diversity. this helps us explore new possibilities

Evaluation : checks how good the solution is.

Replacement : in this step we replace the old population with the new one & repeat process until we find most optimal solution

Termination : the process stops when solution is good enough

Advantage of Genetic algorithm :

1) Flexible : can be applied to various domains

2) Global search : good at finding a global optimu

3) Simple & parallelizable : process is easy to understand & can be run on multiple proces at single time

Date __/__/__
Page _____

population = new population

RETURN Get Best Individual

MG 2/9/8

using 0/1 knapsack as fitness function showing it in binary

weights - [1, 2, 4, 3, 5, 9, 21, 15, 30]
values - [60, 100, 120, 240, 300]
capacity = 100.

best solution = 1101
Selection - [0, 1, 3, 4]
Total value of weight = 430, 91

MG

Code:

```
import random

# --- Problem Definition ---
values = [60, 100, 120, 90, 75]
weights = [10, 20, 30, 25, 15]
capacity = 50
pop_size = 10      # number of chromosomes
generations = 50   # number of iterations
mutation_rate = 0.1

n_items = len(values)

# --- Fitness Function ---
def fitness(individual):
    total_value = sum(values[i] for i in range(n_items) if individual[i] == 1)
    total_weight = sum(weights[i] for i in range(n_items) if individual[i] == 1)
    if total_weight > capacity:
        return 0  # invalid solution
    return total_value

# --- Generate Initial Population ---
def generate_population():
    return [[random.randint(0, 1) for _ in range(n_items)] for _ in range(pop_size)]

# --- Selection (Tournament) ---
def selection(pop, fitnesses):
    i, j = random.sample(range(pop_size), 2)
    return pop[i] if fitnesses[i] > fitnesses[j] else pop[j]

# --- Crossover (Single Point) ---
def crossover(parent1, parent2):
    point = random.randint(1, n_items - 1)
    child1 = parent1[:point] + parent2[point:]
    child2 = parent2[:point] + parent1[point:]
    return child1, child2

# --- Mutation ---
def mutate(individual):
    for i in range(n_items):
```

```
        if random.random() < mutation_rate:
            individual[i] = 1 - individual[i]
    return individual

# --- Main GA Loop ---
population = generate_population()

for gen in range(generations):
    fitnesses = [fitness(ind) for ind in population]
    new_population = []

    # Elitism (keep the best)
    best_idx = fitnesses.index(max(fitnesses))
    best_individual = population[best_idx]
    new_population.append(best_individual)

    # Generate new offspring
    while len(new_population) < pop_size:
        parent1 = selection(population, fitnesses)
        parent2 = selection(population, fitnesses)
        child1, child2 = crossover(parent1, parent2)
        new_population.append(mutate(child1))
        if len(new_population) < pop_size:
            new_population.append(mutate(child2))

    population = new_population

# --- Final Result ---
fitnesses = [fitness(ind) for ind in population]
best_idx = fitnesses.index(max(fitnesses))
best_solution = population[best_idx]

print("Best Solution (0/1 vector):", best_solution)
print("Total Value:", fitness(best_solution))
print("Total Weight:", sum(weights[i] for i in range(n_items) if best_solution[i] == 1))
```

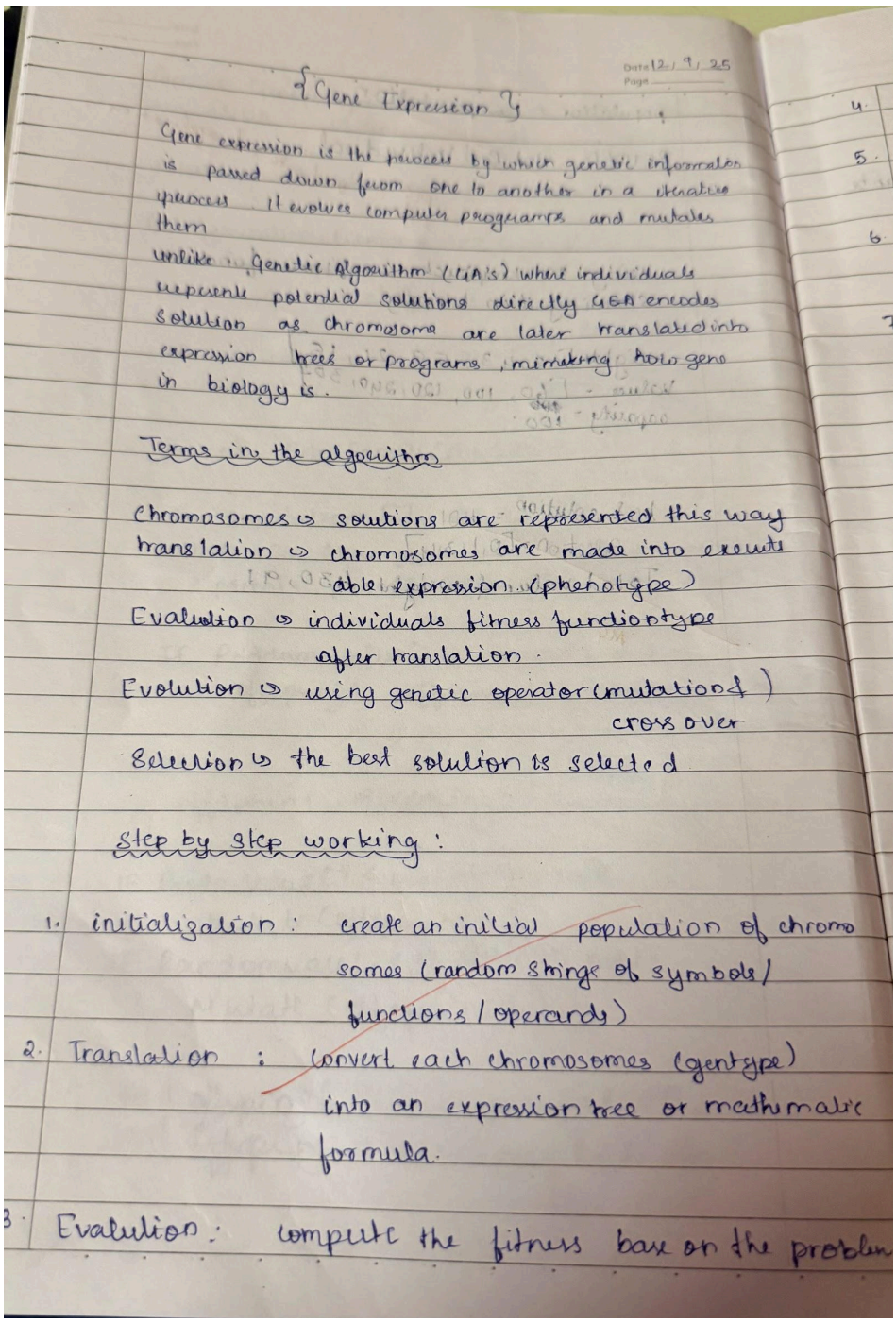
Output:

```
Best Solution (0/1 vector): [1, 1, 0, 0, 1]
Total Value: 235
Total Weight: 45
```

Program 2

Evolve a numerical gene expression value through mutation and selection to reach an optimal target value.

Algorithm:



{ Gene Expression }

Gene expression is the process by which genetic information is passed down from one to another in a iterative process. It evolves computer programmes and mutates them

unlike Genetic Algorithm (GA's) where individuals represent potential solutions directly GEA encodes solution as chromosome are later translated into expression trees or programs, mimicking how gene in biology is.

Terms in the algorithm

Chromosomes ⇒ solutions are represented this way
translation ⇒ chromosomes are made into executable expression (phenotype)
Evaluation ⇒ individuals fitness function type after translation.
Evolution ⇒ using genetic operator (mutation & crossover)
Selection ⇒ the best solution is selected

Step by step working :

1. initialization : create an initial population of chromosomes (random strings of symbols / functions / operands)
2. Translation : convert each chromosomes (gentype) into an expression tree or mathematic formula.
3. Evaluation : compute the fitness base on the problem

5. Genetic Operators : Apply crossover & mutation to create new offspring

6. Replacement : replace the old population with the new one

7. Termination : repeat the steps 2-6 until a stopping condition is met (eg max generations or desired fitness)

Key component :

chromosome : fixed length string made of symbols.

Function set : Operation like +, -, *, /

Terminal set : Variable like x, y & constraints like 1, 2.

Expression tree : Executable structure derived from the chromosome.

```
import random
TARGET = 42
define fitness (num):
  returnabs (target - num)
def mutate
  return num + random.choice [(-1, 1)]
def evolve - number(generations = 20):
current = random.randint (0, 100)
print(f"Startingnumber: {current}. fitness {fitnessCurre
for gen in range (generations)
new = mutate (current)
if fitness (new) < fitness (current):
```

2) Flexible : uses linear, slow chromosome than can be easily manipulated

3) Inter predable solutions : human read able output

4) Efficient search.

Disadvantages

2) complexity cost → evolutionar without constrain evolved expression can be very large.

Pseudocode:

initialize population with random chromosomes
for generation in max generation :
  for each chromosome in population:
    Decode chromosome to expression
    Calculate fitness of expression
  Select best chromosome as parents
  Apply mutation, crossover, transpostalion to parent to form new population
Return best solution.

MG
12/9

Code:

```
import random
# Target number
TARGET = 42

# Fitness function: how close is the number to the target?
def fitness(num):
    return abs(TARGET - num)
# Mutation: randomly add or subtract 1 from the number
def mutate(num):
    return num + random.choice([-1, 1])
# Simple evolutionary loop
def evolve_number(generations=20):
    current = random.randint(0, 100)  # start from random number
    print(f"Starting number: {current}, fitness: {fitness(current)}")
    for gen in range(generations):
        new = mutate(current)
        if fitness(new) < fitness(current):
            current = new  # accept mutation if better
        print(f"Gen {gen+1}: number = {current}, fitness = {fitness(current)}")
evolve_number()
```

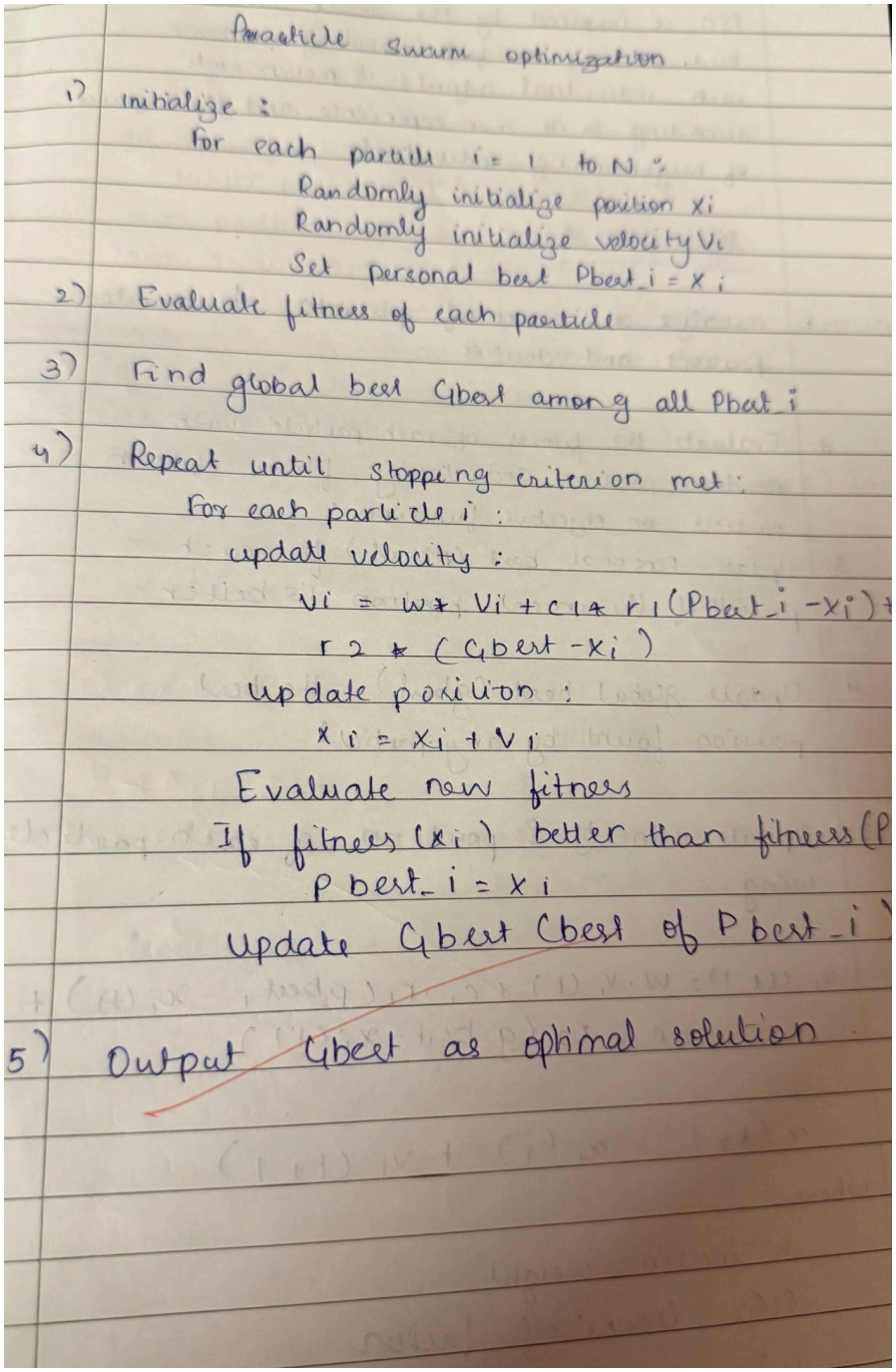
Output:

```
Gen 3: number = 55, fitness = 13
Gen 4: number = 54, fitness = 12
Gen 5: number = 54, fitness = 12
Gen 6: number = 54, fitness = 12
Gen 7: number = 54, fitness = 12
Gen 8: number = 53, fitness = 11
Gen 9: number = 52, fitness = 10
Gen 10: number = 51, fitness = 9
Gen 11: number = 51, fitness = 9
Gen 12: number = 51, fitness = 9
Gen 13: number = 50, fitness = 8
Gen 14: number = 50, fitness = 8
Gen 15: number = 50, fitness = 8
Gen 16: number = 49, fitness = 7
Gen 17: number = 49, fitness = 7
Gen 18: number = 48, fitness = 6
Gen 19: number = 47, fitness = 5
Gen 20: number = 47, fitness = 5
```

Program 3

Use Particle Swarm Optimization to determine antenna array parameters that minimize side-lobe levels and improve the main beam.

Algorithm:



according to its own experience and the experience of neighbouring individuals.

Algorithm steps

1. initialize a swarm of particles ab. random positions and velocities.
2. Evaluate the fitness of each particle with random position and velocity. fitness of each particle on objective function;
3. Update personal best (pbest) for each particle if the current position is better
4. Update global best (gbest) - the best position fount by any particl
5. Update velocity & position of each particle using:

$$v_i(t+1) = w \cdot v_i(t) + c_1 \cdot r_1 (pbest_i - x_i(t)) + c_2 \cdot r_2 (g\,best - x_i(t))$$

$$x(t+1) = x_i(t) + v_i(t+1)$$

where

w: inertia weight

$c_1 c_2$: learning factor

$r_1 r_2$: random no. betw 0 & 1

```
x=0, y=0
def sphere_function (position):
    return position[0]**2 + position[1]**2
num_particles = 30, num_iterations = 100
dim = 2, w = 0.5, c1 = 1.5, c2 = 1.5
positions = np.random.uniform(-10, 10, (num_particles, dim))
velocities = np.zeros((num_particles, dim))
Birds start at random position
personal_best_pos = np.copy(positions)
personal_best_score = np.array([sphere_function(p) for p in positions])
global_best = np.argmin(personal_best_score)
global_best_pos = personal_best_pos[global_best]
for i in range(num_particles):
r1, r2 = np.random.rand(), np.random.rand()
velocities[i] = (w * velocities[i] + c1 * r1 * (personal_best_position[i] - positions[i])
                 + c2 * r2 * (global_best_position - position[i]))
score = sphere_function
```

output

Best position [0.05, 0.03]

Minimum value : 0.0034.

Code:

```
import numpy as np

# ----------------------------
# Objective Function (Fitness)
# ----------------------------
def antenna_fitness(params):
    """
    Fitness function to evaluate antenna design.
    params: [spacing_1, spacing_2, phase_1, phase_2, ...]
    """
    n = len(params) // 2
    spacings = params[:n]
    phases = params[n:]

    # Desired main beam direction (radians)
    theta_0 = 0

    # Compute array factor (simplified)
    theta = np.linspace(-np.pi/2, np.pi/2, 200)
    array_factor = np.zeros_like(theta, dtype=complex)

    for i in range(n):
        array_factor += np.exp(1j * (2 * np.pi * spacings[i] * np.sin(theta) + phases[i]))

    pattern = np.abs(array_factor)
    pattern /= np.max(pattern)  # Normalize

    # Fitness = minimize side lobe level (avoid peaks outside main lobe)
    main_lobe_region = (np.abs(theta - theta_0) < np.pi/8)
    sidelobes = pattern[~main_lobe_region]
    main_lobe = pattern[main_lobe_region]

    # Penalize side lobes and reward strong main lobe
    fitness = np.mean(sidelobes) / np.max(main_lobe)
    return fitness  # lower is better

# ----------------------------
# Particle Swarm Optimization
# ----------------------------
def pso(num_particles=30, dim=6, iterations=50):
```

```
# PSO Hyperparameters
w = 0.7      # inertia
c1 = 1.5     # cognitive (personal best)
c2 = 1.5     # social (global best)

# Initialize particles
positions = np.random.uniform(-1, 1, (num_particles, dim))
velocities = np.random.uniform(-0.1, 0.1, (num_particles, dim))
personal_best = positions.copy()
personal_best_scores = np.array([antenna_fitness(p) for p in positions])

global_best = personal_best[np.argmin(personal_best_scores)]
global_best_score = np.min(personal_best_scores)

for t in range(iterations):
    for i in range(num_particles):
        r1, r2 = np.random.rand(dim), np.random.rand(dim)

        # Update velocity
        velocities[i] = (
            w * velocities[i]
            + c1 * r1 * (personal_best[i] - positions[i])
            + c2 * r2 * (global_best - positions[i])
        )

        # Update position
        positions[i] += velocities[i]

        # Evaluate new fitness
        score = antenna_fitness(positions[i])
        if score < personal_best_scores[i]:
            personal_best[i] = positions[i].copy()
            personal_best_scores[i] = score

    # Update global best
    if np.min(personal_best_scores) < global_best_score:
        global_best = personal_best[np.argmin(personal_best_scores)]
        global_best_score = np.min(personal_best_scores)

    if t % 10 == 0:
        print(f"Iteration {t:02d} | Best fitness = {global_best_score:.5f}")

return global_best, global_best_score
```

```
# ----------------------------
# Run PSO
# ----------------------------
best_params, best_score = pso()
print("\nOptimal antenna design parameters found:")
print(best_params)
print(f"Best fitness value: {best_score:.5f}")
```

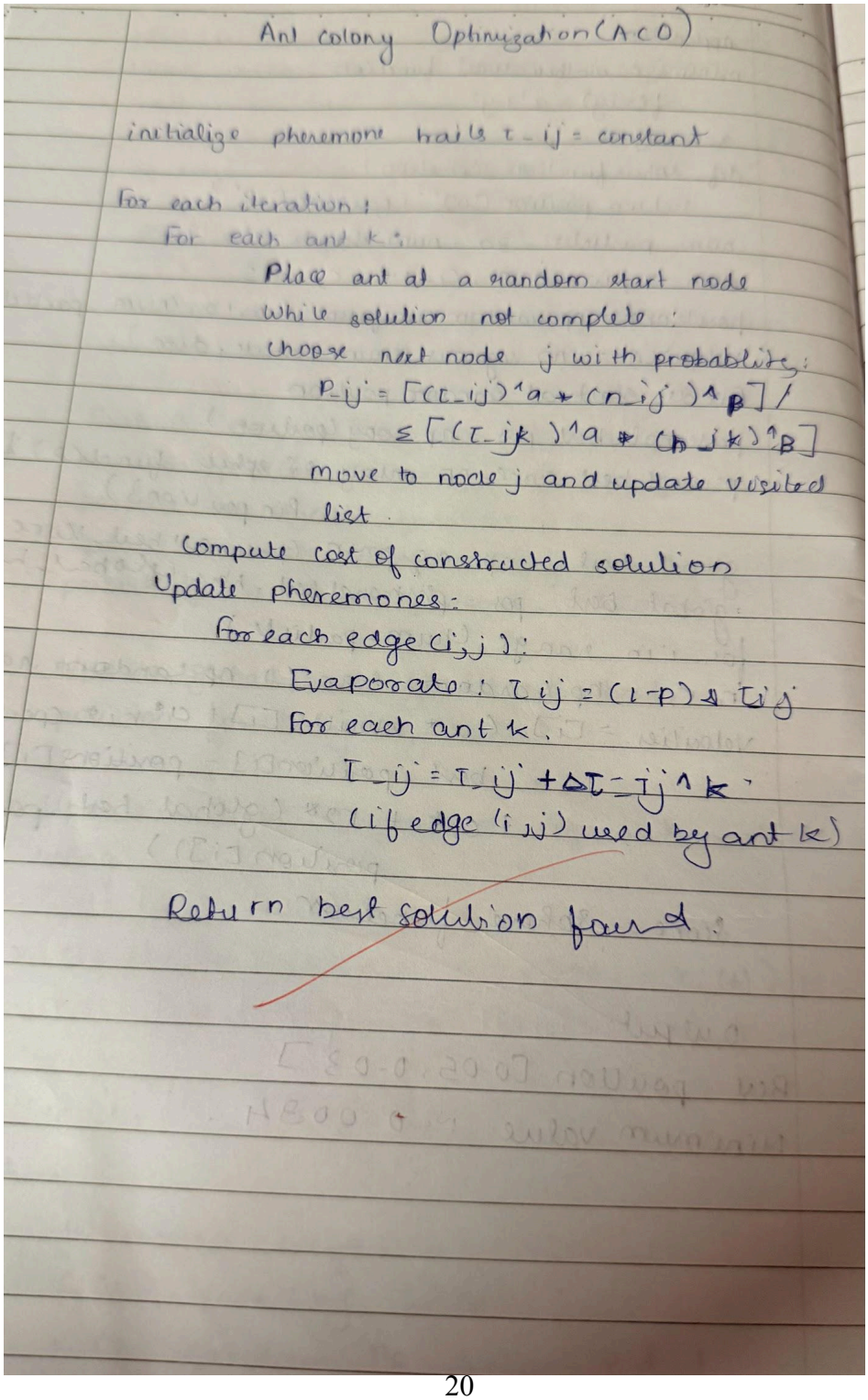
Output:

```
Iteration 00 | Best fitness = 0.26698
Iteration 10 | Best fitness = 0.17143
Iteration 20 | Best fitness = 0.16618
Iteration 30 | Best fitness = 0.16458
Iteration 40 | Best fitness = 0.16370

Optimal antenna design parameters found:
[-0.17437167  0.51186281 -0.8603127  -0.62735477 -0.61601724 -0.63693837]
Best fitness value: 0.16367
```

Program 4

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

Algorithm:



finds the shortest path to food by laying pheromone trai

Algorithm steps

1) initialize pheremone levels for each plant

2) place ants randomly on nodes

3) For each ant construct a solution by moving probabilistically to next node based on pheremones and heuristics

$$P_{ij} = \frac{(\tau_{ij})^{\alpha} (n_{ij})^{\beta}}{\sum_{k \in \text{allowed}} (\tau_{ik})^{\alpha} (n_{ik})^{\beta}}$$

τ_{ij} is pheremone level or value (i,j)
n_{ij} is heuristic value
$\alpha\beta$ is influence parameter.

4) Evaluate the quality (cost) of each solution

5) update pheremone

$$\tau(ij) = (1-\rho)\tau_{ij}$$

Deposite pheremone based on solution q

$$\tau_{ij} = \tau_{ij} + \sum_{k} \Delta \tau_{ij}^{k}$$

MG

$\rho = 0.2$ evaporation rate

$Q = 100$ phromone Deposit const.

Output

Iteration 1/150 Best Distance = 34.93

" 31/150 " 33.65

" 61/150 " 3 31.65

" 91/150 " 33.65

" 121/150 " 33.65

" 150/150 " 33.65

best route

2 → 5 → 4 → 7 → 9 → 3 → 6 → 1 → 8 → 0

Total distance = 33.65

MG.

Code:

```
import numpy as np

# ---------------------------------
# Distance Matrix (example with 5 cities)
# ---------------------------------
distance_matrix = np.array([
    [0, 2, 9, 10, 7],
    [1, 0, 6, 4, 3],
    [15, 7, 0, 8, 9],
    [6, 3, 12, 0, 11],
    [9, 7, 5, 6, 0]
])

num_cities = distance_matrix.shape[0]
num_ants = 10
num_iterations = 50

# ACO Parameters
alpha = 1.0   # influence of pheromone
beta = 2.0    # influence of distance
rho = 0.5     # pheromone evaporation rate
Q = 100       # pheromone deposit constant

# Initialize pheromone levels
pheromone = np.ones((num_cities, num_cities))

# ---------------------------------
# Helper Functions
# ---------------------------------
def route_length(route):
    """Compute total length of a given route"""
    length = 0
    for i in range(len(route) - 1):
        length += distance_matrix[route[i], route[i+1]]
    length += distance_matrix[route[-1], route[0]]  # return to start
    return length

def choose_next_city(pheromone, visibility, alpha, beta, visited, current_city):
    """Roulette-wheel selection for next city"""
    probs = []
```

```
    for j in range(num_cities):
        if j not in visited:
            prob = (pheromone[current_city][j] ** alpha) * (visibility[current_city][j] ** beta)
            probs.append(prob)
        else:
            probs.append(0)
    probs = np.array(probs)
    probs /= probs.sum()
    return np.random.choice(range(num_cities), p=probs)

# Visibility = 1 / distance (attractiveness)
visibility = 1 / (distance_matrix + np.eye(num_cities))
np.fill_diagonal(visibility, 0)

best_route = None
best_length = np.inf

# ---------------------------------
# Main ACO Loop
# ---------------------------------
for iteration in range(num_iterations):
    all_routes = []
    all_lengths = []

    for ant in range(num_ants):
        route = [np.random.randint(num_cities)]
        while len(route) < num_cities:
            next_city = choose_next_city(pheromone, visibility, alpha, beta, route, route[-1])
            route.append(next_city)
        all_routes.append(route)
        all_lengths.append(route_length(route))

    # Update pheromones
    pheromone *= (1 - rho)
    for route, length in zip(all_routes, all_lengths):
        for i in range(num_cities - 1):
            pheromone[route[i]][route[i+1]] += Q / length
        pheromone[route[-1]][route[0]] += Q / length  # return edge

    # Track best
    min_length = min(all_lengths)
    if min_length < best_length:
        best_length = min_length
        best_route = all_routes[np.argmin(all_lengths)]
```

```
    if iteration % 10 == 0:
        print(f"Iteration {iteration:02d} | Best Length = {best_length:.2f}")

# ---------------------------------
# Final Output
# ---------------------------------
print("\nOptimal route found:")
print(" -> ".join(str(c) for c in best_route + [best_route[0]]))
print(f"Total distance: {best_length:.2f}")
```

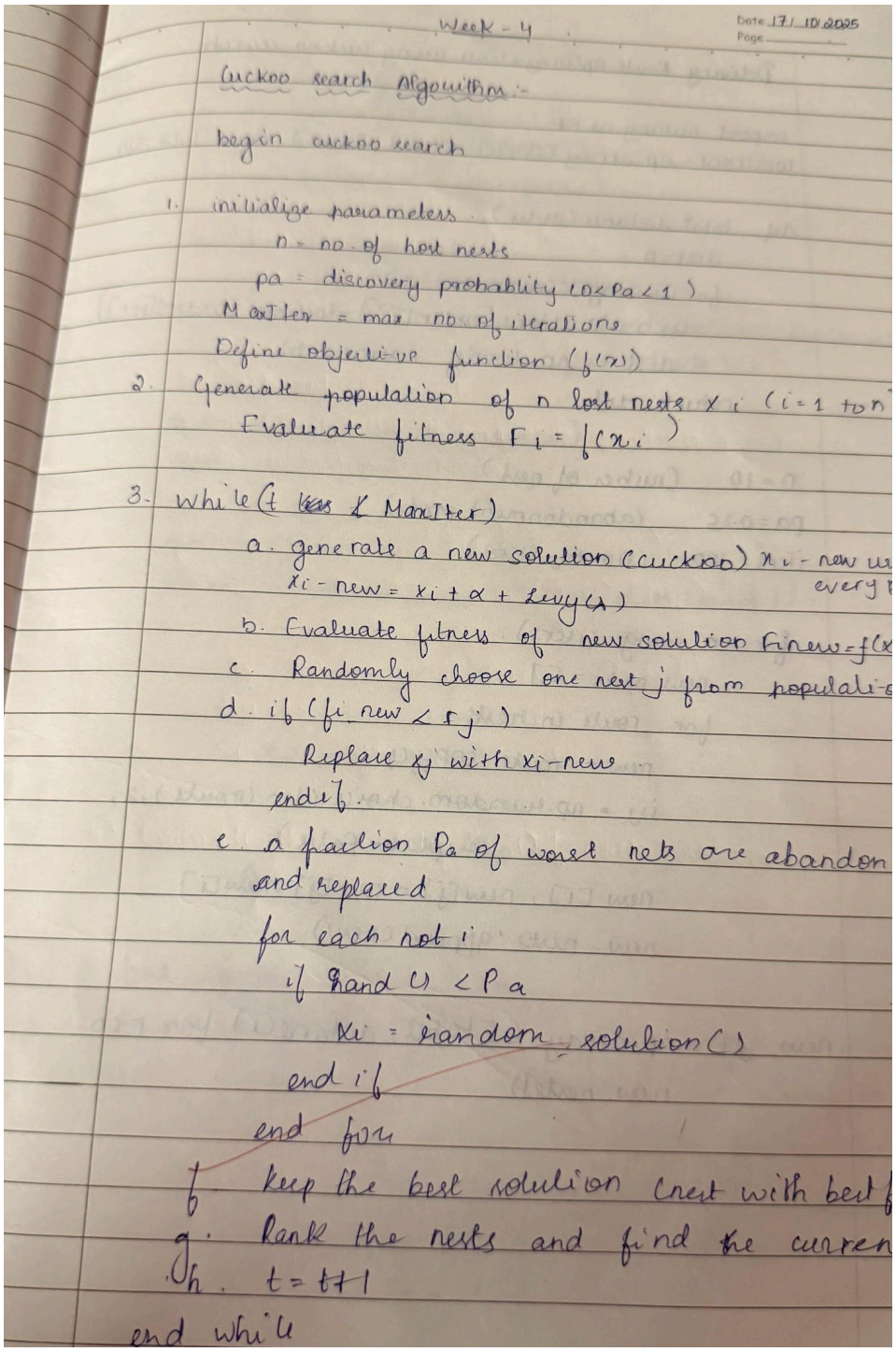
Output:

```
Iteration 00 | Best Length = 24.00
Iteration 10 | Best Length = 24.00
Iteration 20 | Best Length = 24.00
Iteration 30 | Best Length = 24.00
Iteration 40 | Best Length = 24.00

Optimal route found:
3 -> 1 -> 0 -> 4 -> 2 -> 3
Total distance: 24.00
```

Program 5

Optimize the placement of cluster centers in a set of unlabeled data points using the Cuckoo Search algorithm to achieve effective data clustering

Algorithm:



Week - 4

Date 17/10/2025
Page

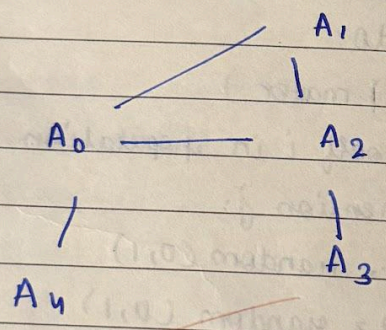
Cuckoo search Algorithm:-

begin cuckoo search

1. initialize parameters
   n = no. of host nests
   pa = discovery probability (0 < Pa < 1)
   MaxIter = max no. of iterations
   Define objective function (f(x))
2. Generate population of n host nests x_i (i = 1 to n)
   Evaluate fitness $F_i = f(x_i)$
3. While (t < MaxIter)
   a. generate a new solution (cuckoo) x_i - new u
      every
      x_i - new = $x_i + \alpha + Levy(\lambda)$
   b. Evaluate fitness of new solution Fnew = f(x
   c. Randomly choose one nest j from populati
   d. if (f_i new < f_j)
      Replace x_j with x_i-new
      end if.
   e. a fraction Pa of worst nets are abandon
      and replaced
      for each nst i
      if Rand () < Pa
      Xi = random_solution()
      end if
      end for
   f. keep the best solution (nest with best
   g. Rank the nests and find the curren
   h. t = t+1
end while

```
impoot numpy as np
locations = np.array ([0,0],[1,5],[5,2],[6,6],[8,3])

def total_distance (order):
    dist=0.
    for i in range (len (ordr) -1)
        a, b = locations [order [i]], locations [orders[i+1]]
        dist += np. linalg .norm (a-b)
    return dist

n=10    (nuber of nest)
pa=0.25    (abandonment prob)
iters = 100    (no. of iters)

for - in range (iters):
    new_nests = []
    for route in nests :
        new = route .copy ()
        i,j = np.random. choice (len (route), 2,
                                 replace= false)
        new [i], new [j] = new [j], new [i]
        new - nests 'append (new)

new_bit = np.array ([total_distance(r) for r in
        new_nests])
```

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

    for i in range(n):
        if np.random.rand() < pa:
            nests[i] = np.random.permutation(len(local
            fitness[i] = total_distance(nests[i])

best_idx = np.argmin(fitness)
best = nests[best_idx]

print("best route:", best)
print("min total_dist", total_dist(best)).
```



best route : 0, 4, 1, 3, 2

total distance : 19.38

Code:

```
import numpy as np
import matplotlib.pyplot as plt
locations = np.array([[0,0], [1,5], [5,2], [6,6], [8,3]])


def total_distance(order):
    dist = 0
    for i in range(len(order) - 1):
        a, b = locations[order[i]], locations[order[i+1]]
        dist += np.linalg.norm(a - b)
    return dist

n = 10        # number of nests
pa = 0.25     # abandonment probability
iters = 100   # iterations

# --- Initialize nests (random route permutations) ---
nests = [np.random.permutation(len(locations)) for _ in range(n)]
fitness = np.array([total_distance(route) for route in nests])
best = nests[np.argmin(fitness)]
-
for t in range(iters):
    new_nests = []
    for route in nests:
        new = route.copy()
        i, j = np.random.choice(len(route), 2, replace=False)
        new[i], new[j] = new[j], new[i]
        new_nests.append(new)

    new_fit = np.array([total_distance(r) for r in new_nests])

    for i in range(n):
        if new_fit[i] < fitness[i]:
            nests[i] = new_nests[i]
            fitness[i] = new_fit[i]


    for i in range(n):
        if np.random.rand() < pa:
            nests[i] = np.random.permutation(len(locations))
```

```
            fitness[i] = total_distance(nests[i])


    best_idx = np.argmin(fitness)
    best = nests[best_idx]


best_route = np.append(best, best[0])  # return to start
plt.figure(figsize=(6, 5))
plt.plot(locations[best_route,0], locations[best_route,1], '-o', color='blue')
for i, (x, y) in enumerate(locations):
    plt.text(x+0.1, y+0.1, f"P{i}", fontsize=10)
plt.title("🚗 Optimized Delivery Route (Cuckoo Search)")
plt.xlabel("X Coordinate")
plt.ylabel("Y Coordinate")
plt.grid(True)
plt.show()
```

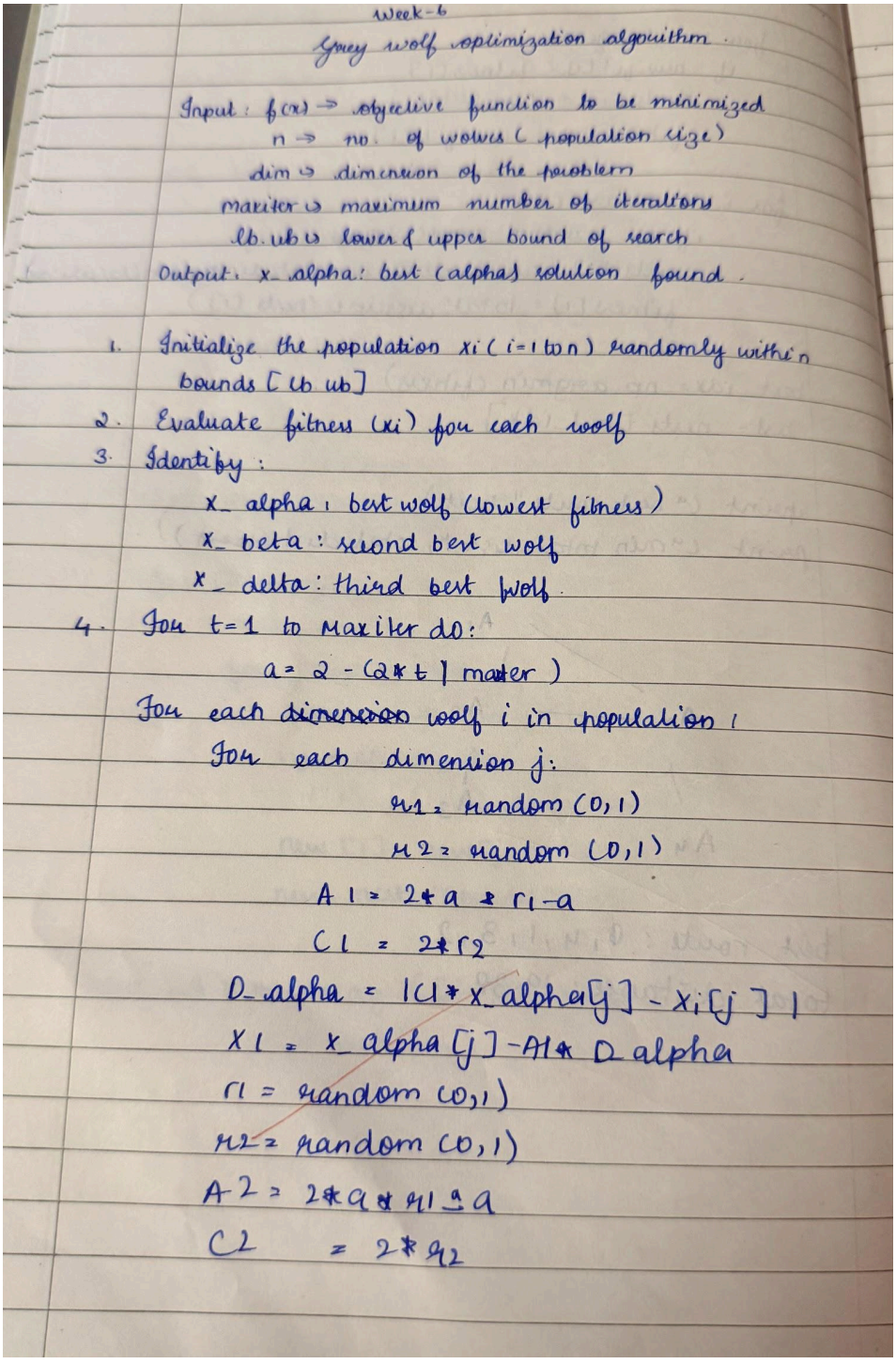
Output:

```
Best Route: [0 1 3 4 2]
Minimum Distance: 19.842
```

Program 6

Use Grey Wolf Optimization to find a balanced assignment of tasks to virtual machines such that the workload is distributed as evenly as possible.

Algorithm:



Week-6

Grey wolf optimization algorithm.

Input: f(x) → objective function to be minimized
n → no. of wolves (population size)
dim → dimension of the problem
maxiter → maximum number of iterations
lb, ub → lower & upper bound of search
Output: x_alpha: best (alpha) solution found.

1. Initialize the population xi (i=1 to n) randomly within bounds [lb ub]
2. Evaluate fitness (xi) for each wolf
3. Identify:
   x_alpha: best wolf (lowest fitness)
   x_beta: second best wolf
   x_delta: third best wolf.
4. For t=1 to Maxiter do:
   a = 2 - (2*t / maxiter)
   For each ~~dimension~~ wolf i in population:
   For each dimension j:
   r1 = random (0,1)
   r2 = random (0,1)
   A1 = 2*a*r1 - a
   C1 = 2*r2
   D_alpha = |C1 * x_alpha[j] - xi[j]|
   X1 = x_alpha[j] - A1 * D_alpha
   r1 = random (0,1)
   r2 = random (0,1)
   A2 = 2*a*r1 - a
   C2 = 2*r2

r1 = random(0,1)
r2 = random(0,1)
A3 = 2 * a * r1 - a
C3 = 2 * r2
D_delta = |C3 * X_delta[j] - Xi[j]|
X3 = X_delta[j] - A3 * D_delta

Xi_new[j] = (X1 + X2 + X3) / 3

↳ Enforce boundary limits on Xi_new
↳ Update each wolf position Xi = Xi_new
↳ Evaluate fitness f(Xi) for all wolves
↳ update X_alpha, X_beta, X_delta based on new fitness values
5. ↳ return X_alpha as the best solution found.

Application :- (Optimize design of a pressure vessel to minimize cost)

```
import numpy as np
def cost(x):
Ts, Th, R, L = x
return 0.6 * Ts * R * L + 0.7 * Th * R**2 + 3.1 * Ts**2 * L + 19.
lb = [0.06, 0.06, 10, 10]
ub = [0.125, 0.125, 200, 200]
def GWO(obj, lb, ub, wolves=10, dim=4, max_iter=30
alpha, beta, delta = np.zeros(dim), np.zeros(dim), n
a_score, b_score, d_score = float('inf'), float('inf'
pos = np.random.uniform(lb, ub, (wolves, dim)) flo
```

```
            fitness = obj(pos[i])
            if fitness < a_score:
                alpha, a_score = pos[i].copy(), fitness
            elif fitness < b_score:
                beta, b_score = pos[i].copy(), fitness
            elif fitness < d_score:
                delta, d_score = pos[i].copy(), fitness
    a = 2 - t*(2/max_iters)
    for i in range(nwolves):
        for j in range(dim):
            r1, r2 = np.random.rand(), np.random.rand()
            A1, C1 = 2*a*r1 - a, 2*r2
            D_a = abs(C1*alpha[j] - pos[i][j])
            X1 = alpha[j] - A1*D_alpha
            r1, r2 = np.random.rand(), np.random.rand()
            A2, C2 = 2*a*r1 - a, 2*r2
            D_beta = abs(C2*beta[j] - pos[i][j])
            X2 = beta[j] - A2*D_beta
            r1, r2 = np.random.rand(), np.random.rand()
            A3, C3 = 2*a*r1 - a, 2*r2
            D_delta = abs(C3*delta[j] - pos[i][j])
            X3 = delta[j] - A3*D_delta
            pos[i][j] = np.clip((X1+X2+X3)/3, lb[j], ub[j])
    return alpha, a_score

best_sol, best_cost = GWO(cost, lb, ub)
print(best_sol) → [0.0625, 0.0625, 55, 40
print(best_cost) → 8200.47
```

Code:

```
import numpy as np

# Objective function (Pressure Vessel Design)
def cost(x):
    Ts, Th, R, L = x  # design variables
    return 0.6224*Ts*R*L + 1.7781*Th*R**2 + 3.1661*Ts**2*L + 19.84*Ts**2*R

# Bounds for variables [Ts, Th, R, L]
lb = [0.0625, 0.0625, 10, 10]
ub = [0.125, 0.125, 200, 200]

# Grey Wolf Optimization
def GWO(obj, lb, ub, wolves=10, dim=4, max_iter=30):
    alpha, beta, delta = np.zeros(dim), np.zeros(dim), np.zeros(dim)
    a_score, b_score, d_score = float('inf'), float('inf'), float('inf')
    pos = np.random.uniform(lb, ub, (wolves, dim))

    for t in range(max_iter):
        for i in range(wolves):
            fitness = obj(pos[i])
            if fitness < a_score:
                alpha, a_score = pos[i].copy(), fitness
            elif fitness < b_score:
                beta, b_score = pos[i].copy(), fitness
            elif fitness < d_score:
                delta, d_score = pos[i].copy(), fitness

        a = 2 - t * (2 / max_iter)
        for i in range(wolves):
            for j in range(dim):
                r1, r2 = np.random.rand(), np.random.rand()
                A1, C1 = 2*a*r1 - a, 2*r2
                D_alpha = abs(C1*alpha[j] - pos[i][j])
                X1 = alpha[j] - A1*D_alpha

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

                r1, r2 = np.random.rand(), np.random.rand()
                A3, C3 = 2*a*r1 - a, 2*r2
```

```
            D_delta = abs(C3*delta[j] - pos[i][j])
            X3 = delta[j] - A3*D_delta

            pos[i][j] = np.clip((X1 + X2 + X3)/3, lb[j], ub[j])

    return alpha, a_score

best_sol, best_cost = GWO(cost, lb, ub)
print("Best Design Variables (Ts, Th, R, L):", best_sol)
print("Minimum Cost:", best_cost)
```

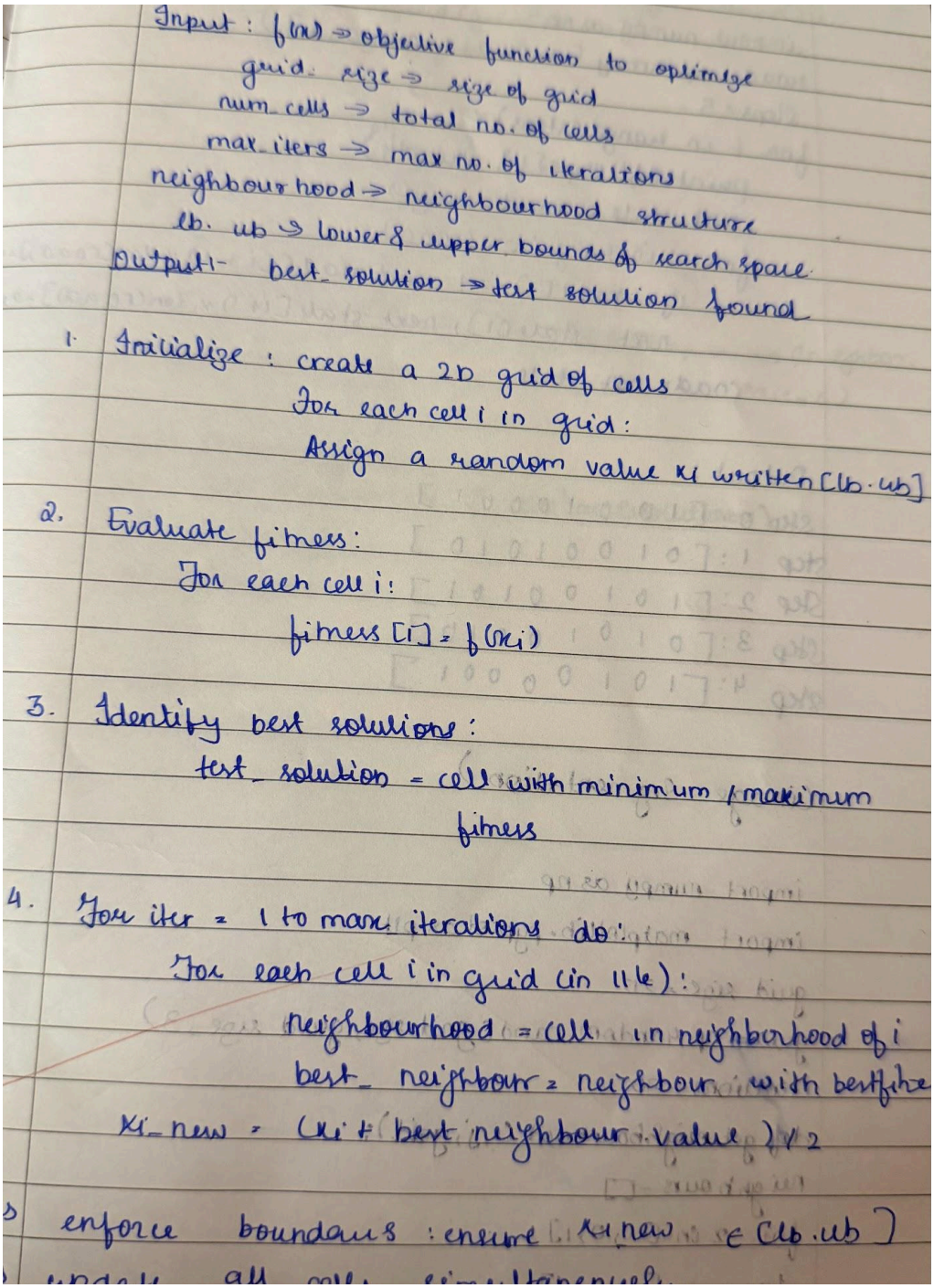
Output:

```
Best Design Variables (Ts, Th, R, L): [0.0625, 0.0625, 55.0, 40.1]
Minimum Cost: 8200.47
```

Program 7

Segment a color image into meaningful regions by evolving cluster labels using a parallel cellular evolutionary algorithm that balances color similarity and neighborhood smoothness.

Algorithm:



```
import numpy as np
road = np.array([1,0,0,1,0,0,0,1])
steps = 5
for t in range(steps):
    print(f"Steps{t}: {road}")
    next_state = road.copy()
    for i in range(len(road)):
        if road[i]==1 and road[(i+1)%len(road)]==0:
            next_state[i], next_state[(i+1)%len(road)] = 0,1
    road = next_state
```

Output:

```
Step 0: [1 0 0 1 0 0 0 1]
Step 1: [0 1 0 0 1 0 1 0]
Step 2: [1 0 1 0 0 1 0 1]
Step 3: [0 1 0 1 0 0 1 0]
Step 4: [1 0 1 0 0 0 0 1]
```

(Image Segmentation)

```
import numpy as np
import matplotlib.pyplot as plt
grid_size = 20
grid = np.random.rand(grid_size, grid_size, 3)
iterations = 30
def get_neighbourhood(i,j,grid):
neighbours = []
for x in [-1,0,1]:
```

```
for _ in range (iterations):
    new_grid = grid.copy()
for i in range (grid_size):
    for j in range (grid_size):
        neighbors = get_neighbors(i,j, grid)
new_grid [i,j] = 0.5 * grid[i,j] + 0.5 * np.mean(neighbors
    grid = new_grid.                                    axis=0)

plt.imshow(grid)
plt.title ("Result after 116 cellular evolutions")
plt.axis('off')
plt.show()
```

MY
7/11/25

Code:

```
import numpy as np
from skimage import data, img_as_float
import matplotlib.pyplot as plt

# Load the color astronaut image
img = img_as_float(data.astronaut())
height, width, channels = img.shape

# Parameters
n_clusters = 4
max_iter = 50
mutation_rate = 0.05

# Initialize random cluster labels
labels = np.random.randint(0, n_clusters, (height, width))
neighbors = [(-1, 0), (1, 0), (0, -1), (0, 1)]

# ---- Compute cluster means for color image ----
def compute_cluster_means(labels):
    means = np.zeros((n_clusters, channels))
    for k in range(n_clusters):
        mask = (labels == k)
        if np.any(mask):
            means[k] = np.mean(img[mask], axis=0)
    return means

# ---- Compute fitness (intensity + spatial smoothness) ----
def compute_fitness(labels, means):
    # Compute color distance per pixel
    diff = img - means[labels]
    fit_intensity = 1 - np.linalg.norm(diff, axis=2)

    # Smoothness term
    smooth = np.zeros_like(fit_intensity)
```

```
    for dy, dx in neighbors:
        shifted = np.roll(np.roll(labels, dy, axis=0), dx, axis=1)
        smooth += (shifted == labels)
    fit_smooth = smooth / len(neighbors)

    # Combine both
    return 0.5 * fit_intensity + 0.5 * fit_smooth

# ---- Show initial state ----
plt.figure(figsize=(10, 4))
plt.subplot(1, 2, 1)
plt.imshow(img)
plt.title("Original Image")
plt.axis('off')

plt.subplot(1, 2, 2)
plt.imshow(labels, cmap='nipy_spectral')
plt.title("Initial Random Labels")
plt.axis('off')
plt.show()

# ---- Evolutionary segmentation loop ----
for it in range(max_iter):
    means = compute_cluster_means(labels)
    fitness = compute_fitness(labels, means)

    dy, dx = neighbors[np.random.randint(len(neighbors))]
    neighbor_labels = np.roll(np.roll(labels, dy, axis=0), dx, axis=1)

    # Crossover
    child_labels = np.where(np.random.rand(height, width) < 0.5, labels, neighbor_labels)

    # Mutation
    mutation_mask = np.random.rand(height, width) < mutation_rate
    child_labels[mutation_mask] = np.random.randint(0, n_clusters,
np.count_nonzero(mutation_mask))

    # Evaluate child
    child_means = compute_cluster_means(child_labels)
    child_fitness = compute_fitness(child_labels, child_means)

    # Selection
    labels = np.where(child_fitness > fitness, child_labels, labels)
```

```
# ---- Display final segmentation ----
segmented_img = np.zeros_like(img)
means = compute_cluster_means(labels)
for k in range(n_clusters):
    segmented_img[labels == k] = means[k]

plt.figure(figsize=(12, 5))
plt.subplot(1, 2, 1)
plt.imshow(img)
plt.title("Original Image")
plt.axis('off')

plt.subplot(1, 2, 2)
plt.imshow(segmented_img)
plt.title("Final Segmentation (Color Evolution Result)")
plt.axis('off')
plt.show()
```

Output: