|  |
| --- |
| **VISVESVARAYA TECHNOLOGICAL UNIVERSITY**    **“JnanaSangama”, Belgaum -590014, Karnataka.**    **LAB RECORD**    **Bio Inspired Systems (23CS5BSBIS)**  ***Submitted by***  **ROHAN VATS (1BM23CS273)**  ***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 **ROHAN VATS (1BM23CS273),** who is a bonafide student of **B.M.S. College of Engineering.** It is in partial fulfillment for the award of **Bachelor of Engineering in Computer Science and Engineering** of the Visvesvaraya Technological University, Belgaum. The Lab report has been approved as it satisfies the academic requirements of the above-mentioned subject and the work prescribed for the said degree.

|  |  |
| --- | --- |
| Swati Sridharan  Assistant Professor  Department of CSE, BMSCE | Dr. Kavitha Sooda  Professor & HOD  Department of CSE, BMSCE |

# Index

|  |  |  |  |
| --- | --- | --- | --- |
| **Sl.**  **No.** | **Date** | **Experiment Title** | **Page No.** |
| 1 | 18/8/2025 | Genetic Algorithm | 4 |
| 2 | 25/8/2025 | Optimization via gene expression | 6 |
| 3 | 1/9/2025 | Particle Swarm Optimization | 11 |
| 4 | 8/9/2025 | Ant Colony Optimization | 14 |
| 5 | 15/9/2025 | Cuckoo search algorithm | 16 |
| 6 | 29/9/2025 | Grey wolf optimizer | 18 |
| 7 | 13/10/2025 | Parallel cellular algorithm | 22 |

Github Link:

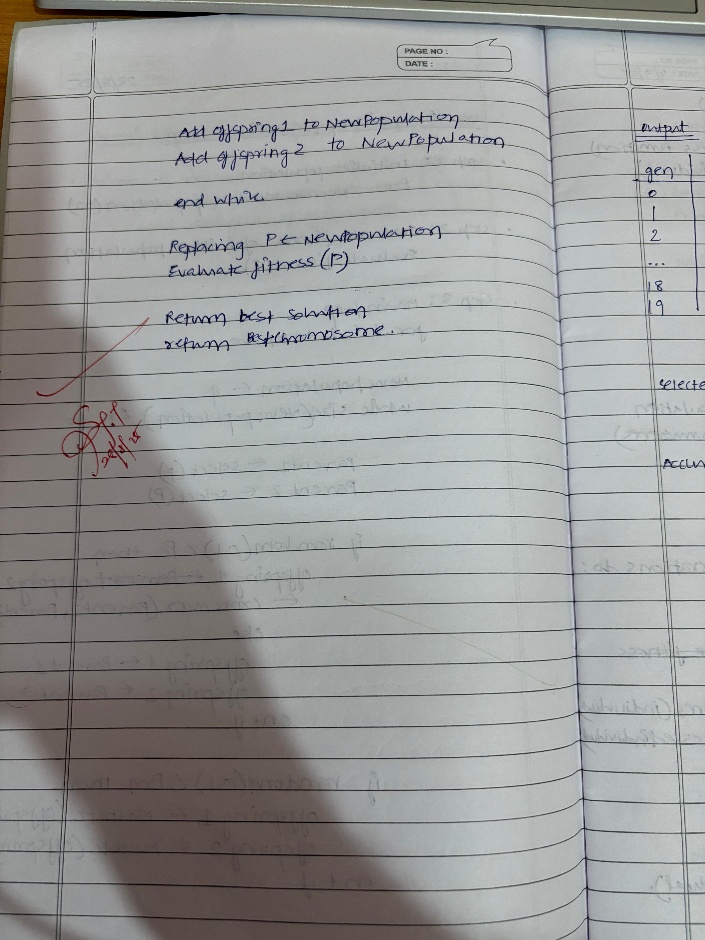
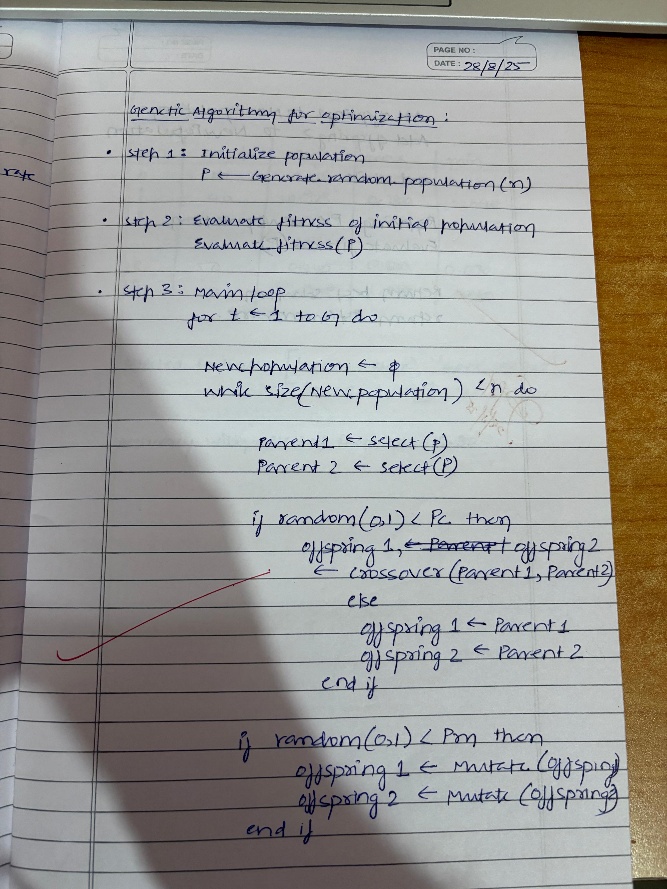
https://github.com/rvats15/BISLAB

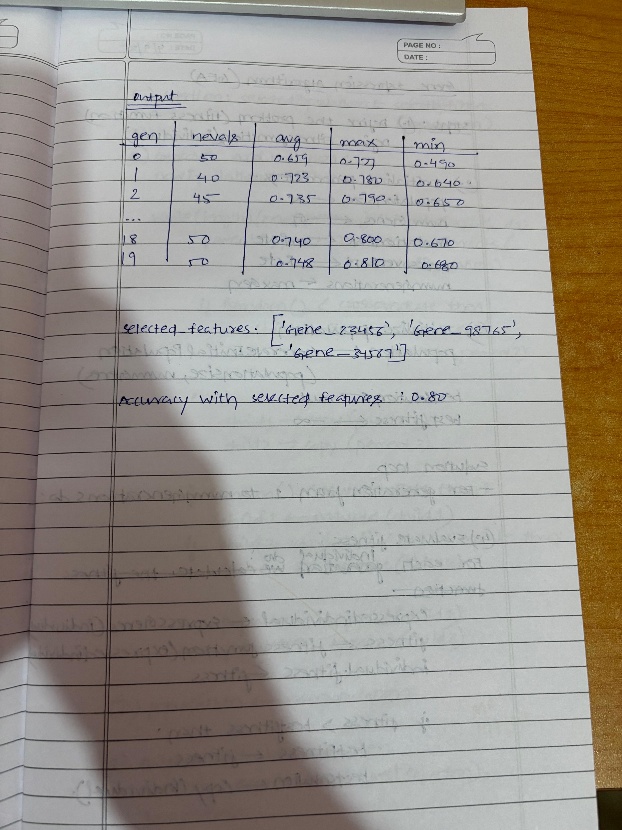
## Program 1 : Genetic Algorithm

**Problem statement:**

Genetic Algorithms (GA) are inspired by the process of natural selection and genetics, where the fittest individuals are selected for reproduction to produce the next generation. GAs are widely used for solving optimization and search problems.

**Algorithm:**





**Code:**

import random def fitness(x): return x\*\*2 def int\_to\_bin(x):

return format(x, '05b') def bin\_to\_int(b): return int(b, 2) def tournament\_selection(pop, k=3): selected = random.sample(pop, k) selected.sort(key=lambda x: fitness(x), reverse=True) return selected[0] def crossover(p1, p2):

b1, b2 = int\_to\_bin(p1), int\_to\_bin(p2) point = random.randint(1, 4) child1 = bin\_to\_int(b1[:point] + b2[point:]) child2 = bin\_to\_int(b2[:point] + b1[point:]) return child1, child2 def mutate(x, mutation\_rate=0.1): if random.random() < mutation\_rate:

b = list(int\_to\_bin(x)) pos = random.randint(0, 4) b[pos] = '1' if b[pos] == '0' else '0' return bin\_to\_int("".join(b)) return x

def genetic\_algorithm(initial\_population=None, pop\_size=6, generations=20, crossover\_rate=0.8, mutation\_rate=0.1): if initial\_population:

population = initial\_population[:pop\_size] # take only needed size else:

population = [random.randint(0, 31) for \_ in range(pop\_size)] for gen in range(generations):

population.sort(key=lambda x: fitness(x), reverse=True) best = population[0] print(f"Gen {gen}: Best x={best}, f(x)={fitness(best)}") new\_pop = [best] while len(new\_pop) < pop\_size:

parent1 = tournament\_selection(population) parent2 = tournament\_selection(population) if random.random() < crossover\_rate:

child1, child2 = crossover(parent1, parent2) else:

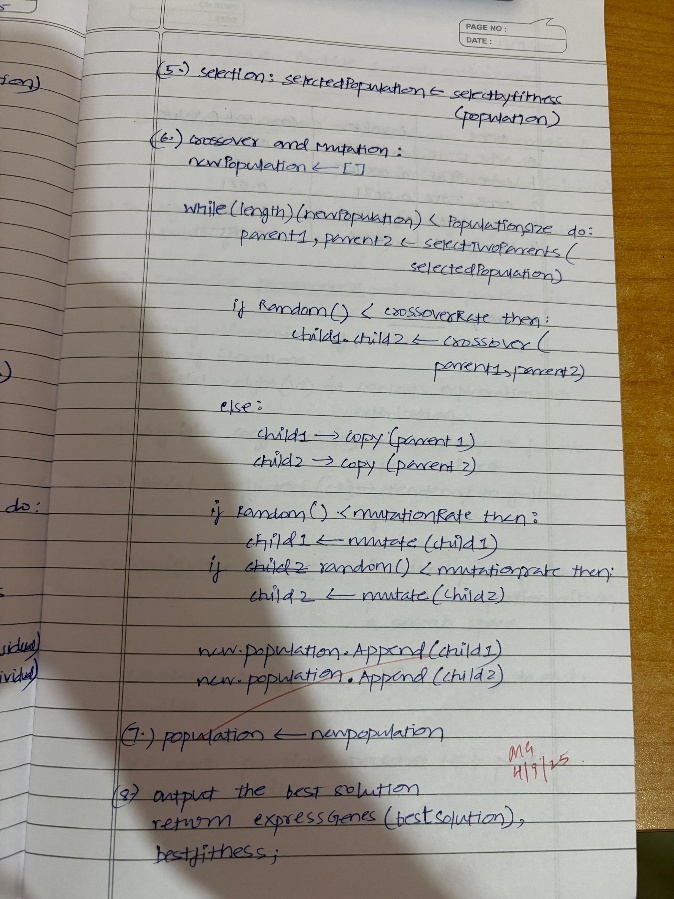
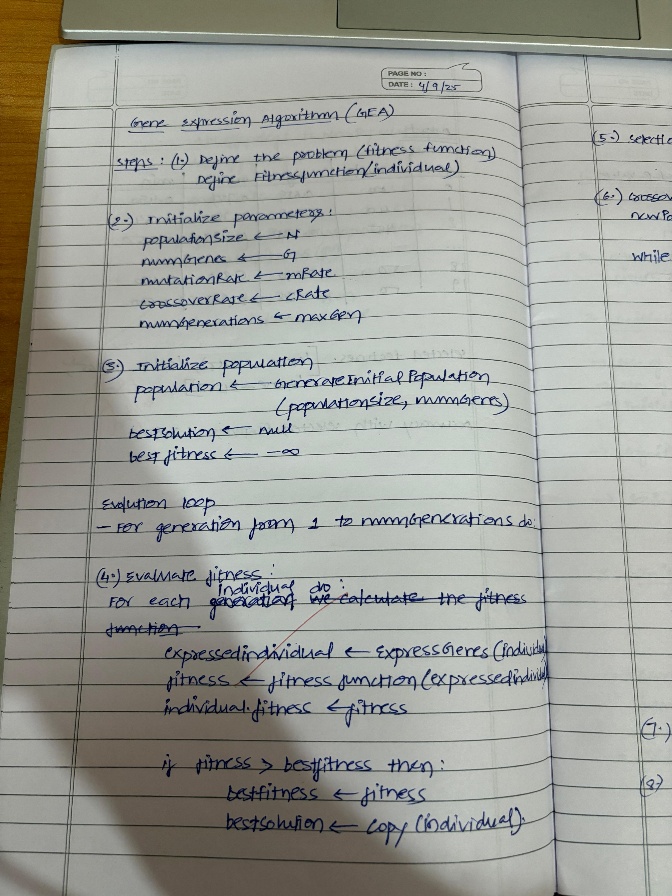
child1, child2 = parent1, parent2 child1 = mutate(child1, mutation\_rate) child2 = mutate(child2, mutation\_rate) new\_pop.extend([child1, child2]) population = new\_pop[:pop\_size] population.sort(key=lambda x: fitness(x), reverse=True) best = population[0] print(f"\nBest Solution: x={best}, f(x)={fitness(best)}") custom\_population = [3, 7, 15, 20, 25, 30] genetic\_algorithm(initial\_population=custom\_population, generations=5)

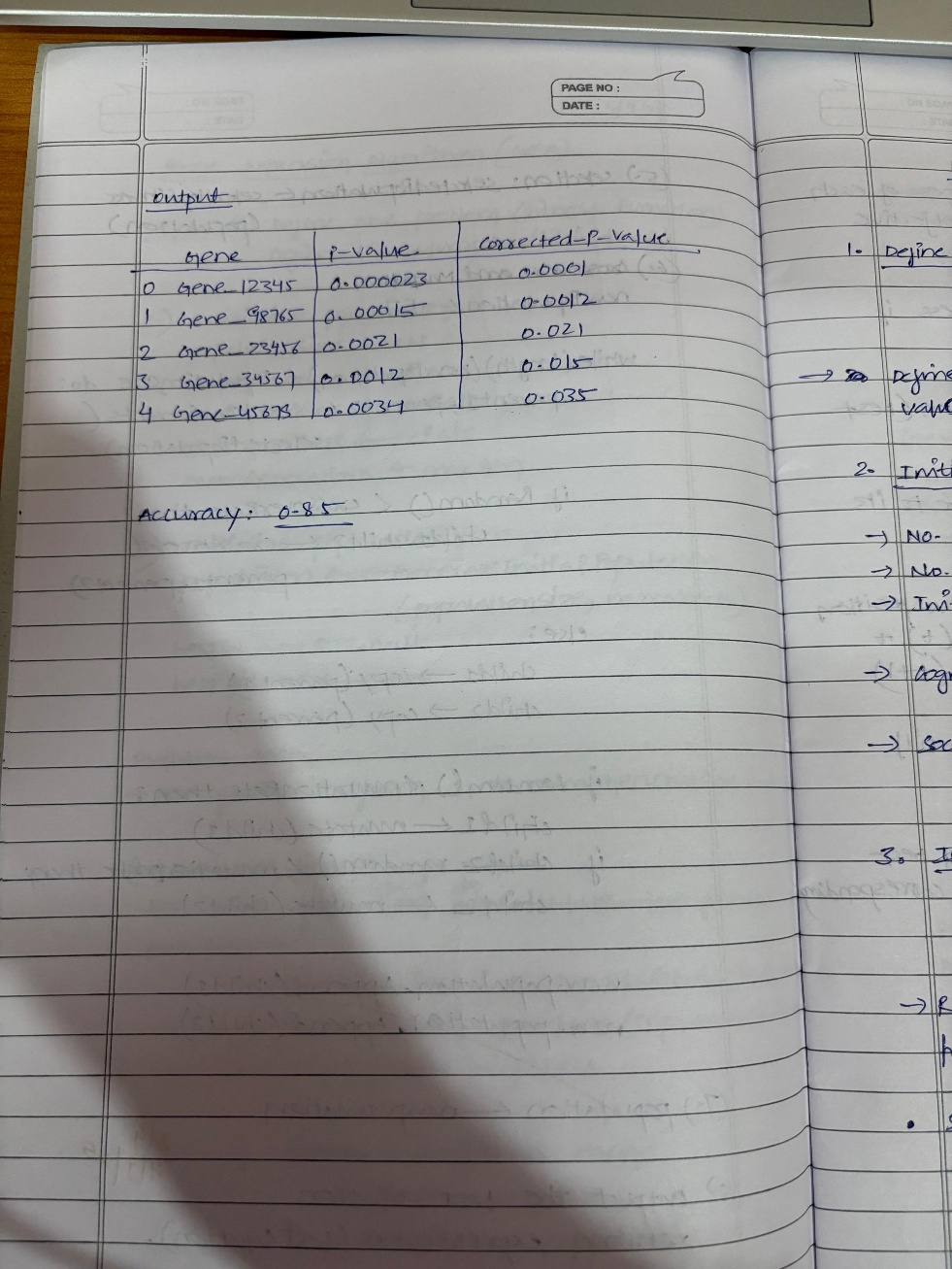
## Program 2 : Optimization via Gene expression

**Problem statement:**

Gene Expression Algorithms (GEA) are inspired by the biological process of gene expression in living organisms. This process involves the translation of genetic information encoded in DNA into functional proteins. In GEA, solutions to optimization problems are encoded in a manner similar to genetic sequences. The algorithm evolves these solutions through selection, crossover, mutation, and gene expression to find optimal or near-optimal solutions. GEA is effective for solving complex optimization problems in various domains, including engineering, data analysis, and machine learning.

**Algorithm:**





**Code:**

import random import math cities = [

(0, 0), (1, 5), (5, 2), (6, 6), (8, 3),

(2, 1), (7, 7), (3, 3), (4, 4), (9, 0)

]

def distance(a, b):

return math.sqrt((a[0]-b[0])\*\*2 + (a[1]-b[1])\*\*2) def total\_distance(tour):

dist = 0

for i in range(len(tour)): city\_a = cities[tour[i]] city\_b = cities[tour[(i+1) % len(tour)]] dist += distance(city\_a, city\_b)

return dist def create\_individual(n): gene = list(range(n)) random.shuffle(gene) return gene def mutate(individual, rate=0.1):

ind = individual[:] for i in range(len(ind)): if random.random() < rate: j = random.randint(0, len(ind)-1) ind[i], ind[j] = ind[j], ind[i] return ind def crossover(parent1, parent2):

size = len(parent1) a, b = sorted([random.randint(0, size-1) for \_ in range(2)]) child = [None]\*size child[a:b+1] = parent1[a:b+1] p2\_index = 0 for i in range(size): if child[i] is None: while parent2[p2\_index] in child:

p2\_index += 1 child[i] = parent2[p2\_index] return child def genetic\_algorithm(generations=100, pop\_size=100, mutation\_rate=0.1): num\_cities = len(cities) population = [create\_individual(num\_cities) for \_ in range(pop\_size)] best = None best\_dist = float('inf') for gen in range(generations):

scored = [(ind, total\_distance(ind)) for ind in population] scored.sort(key=lambda x: x[1]) if scored[0][1] < best\_dist: best = scored[0][0] best\_dist = scored[0][1] new\_pop = [best] while len(new\_pop) < pop\_size:

p1 = random.choice(scored[:50])[0] p2 = random.choice(scored[:50])[0] child = crossover(p1, p2) child = mutate(child, mutation\_rate) new\_pop.append(child) population = new\_pop if gen % 20 == 0:

print(f"Gen {gen}: Best distance = {best\_dist:.2f}")

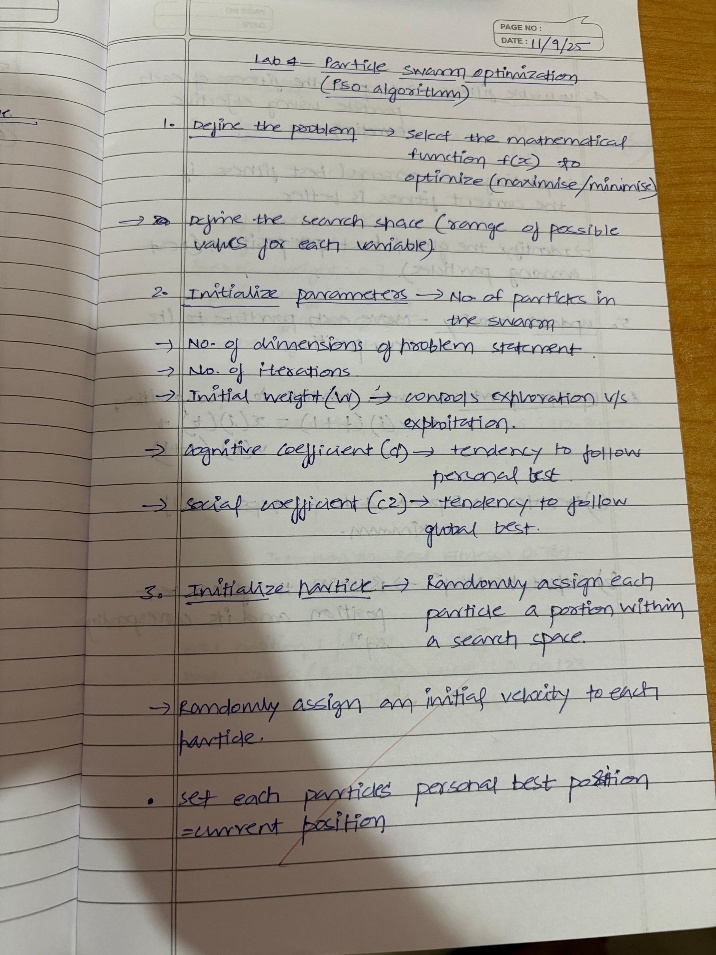
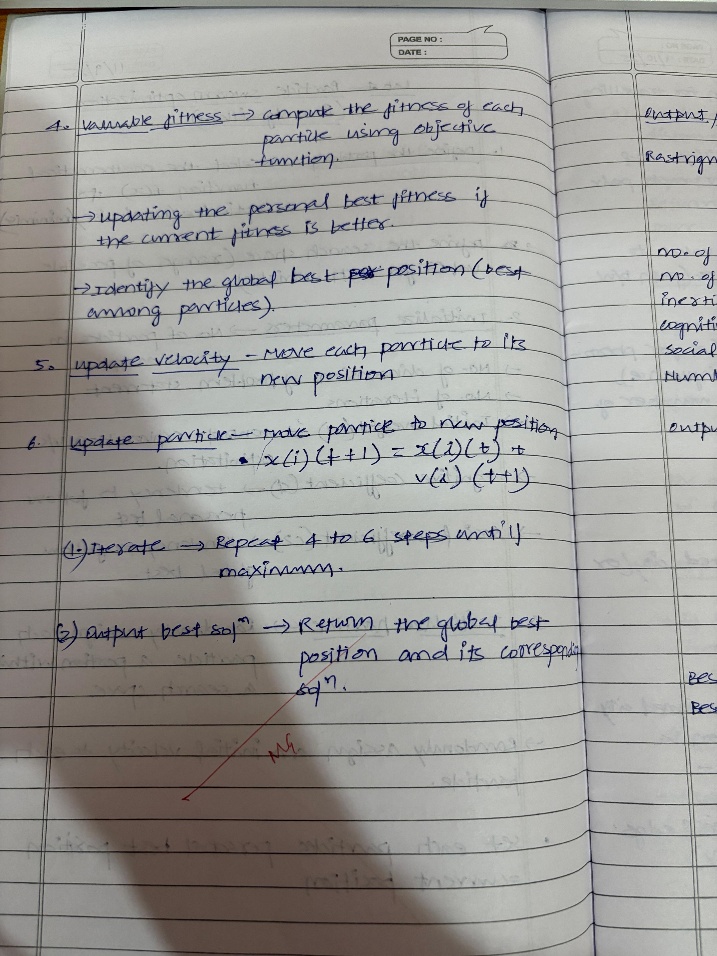
return best, best\_dist best\_tour, best\_dist = genetic\_algorithm() print("\nBest tour found:") print(best\_tour) print(f"Total distance: {best\_dist:.2f}")

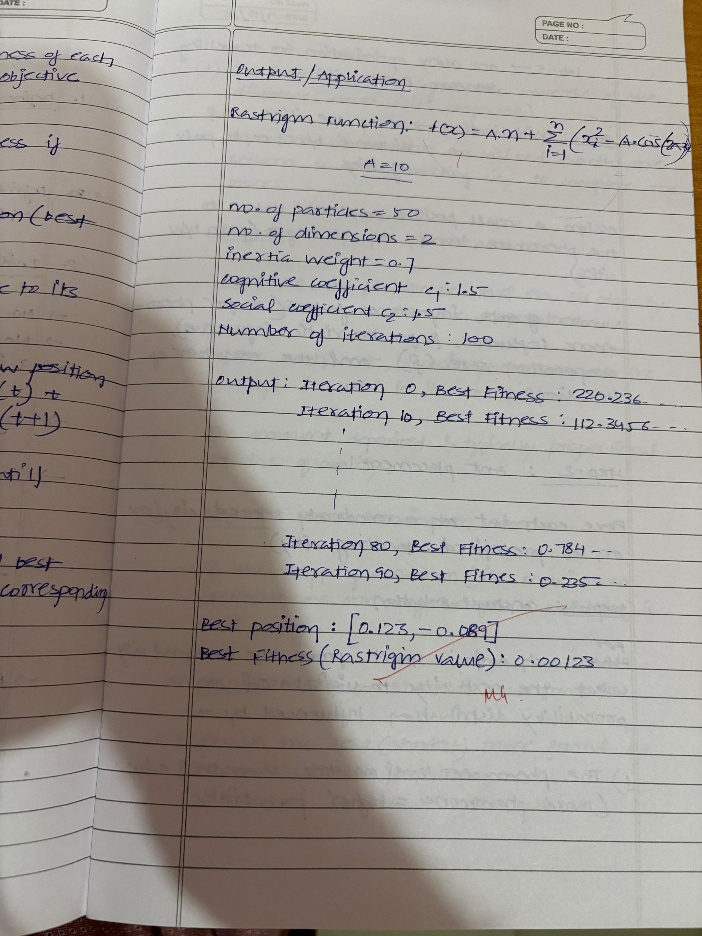
## Program 3 : Particle swarm Optimization

**Problem statement:**

Particle Swarm Optimization (PSO) is inspired by the social behavior of birds flocking or fish schooling. PSO is used to find optimal solutions by iteratively improving a candidate solution with regard to a given measure of quality.

**Algorithm:**



**Code:**

import numpy as np x\_data = np.array([1, 2, 3, 4, 5]) y\_data = np.array([3, 5, 7, 9, 11]) def objective\_function(theta):

theta\_0, theta\_1 = theta predictions = theta\_0 + theta\_1 \* x\_data errors = y\_data - predictions return np.sum(errors\*\*2) num\_particles = 30 num\_iterations = 10

w = 0.7 c1 = 1.5 c2 = 2.1 bounds = [(-10, 10), (-10, 10)] positions = np.array([np.random.uniform(low, high, num\_particles) for low, high in bounds]).T velocities = np.random.uniform(-1, 1, (num\_particles, 2)) personal\_best\_positions = np.copy(positions) personal\_best\_values = np.array([objective\_function(p) for p in personal\_best\_positions]) best\_particle\_index = np.argmin(personal\_best\_values) global\_best\_position = personal\_best\_positions[best\_particle\_index] global\_best\_value = personal\_best\_values[best\_particle\_index] for iteration in range(num\_iterations): for i in range(num\_particles):

fitness = objective\_function(positions[i]) if fitness < personal\_best\_values[i]: personal\_best\_values[i] = fitness personal\_best\_positions[i] = positions[i] if fitness < global\_best\_value: global\_best\_value = fitness global\_best\_position = positions[i] for i in range(num\_particles): r1 = np.random.rand(2) r2 = np.random.rand(2) cognitive = c1 \* r1 \* (personal\_best\_positions[i] - positions[i]) social = c2 \* r2 \* (global\_best\_position - positions[i]) velocities[i] = w \* velocities[i] + cognitive + social positions[i] += velocities[i] for dim in range(2):

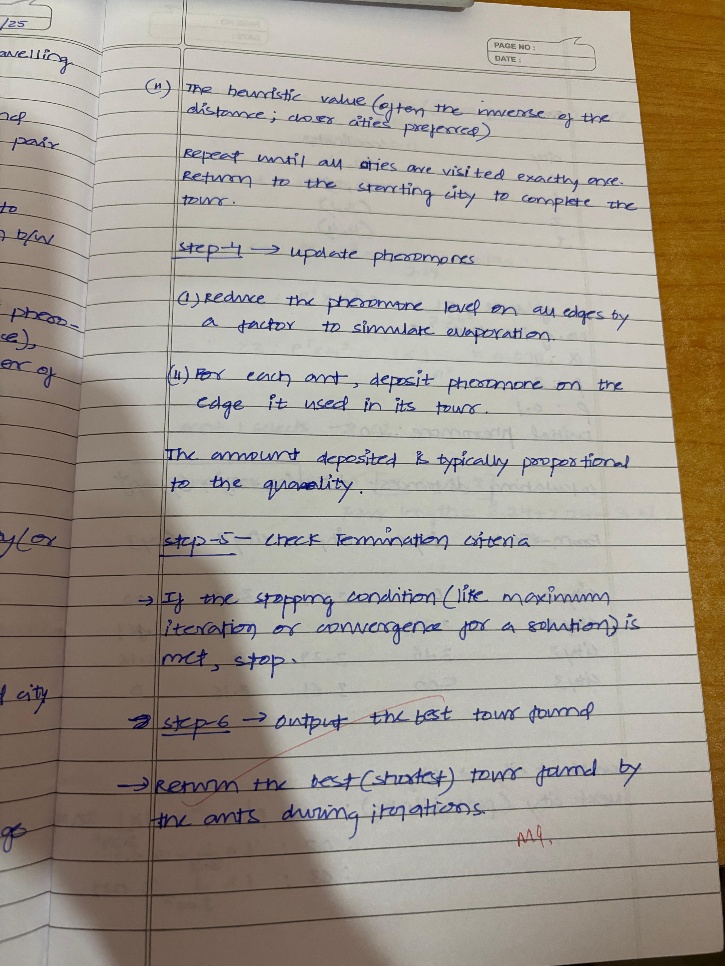
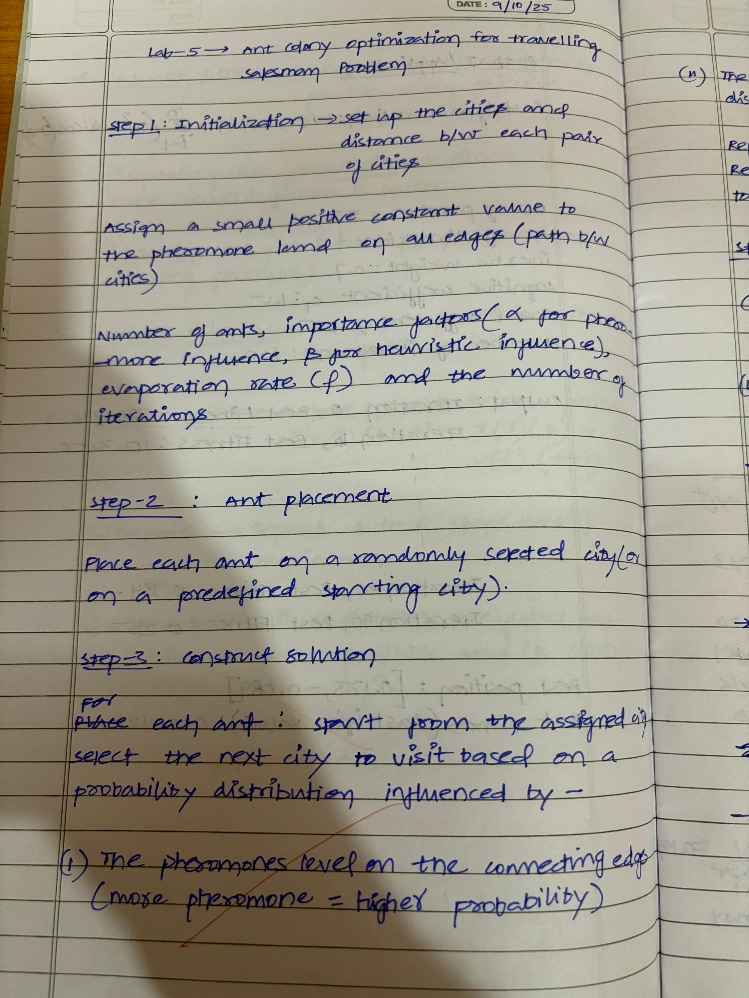
positions[i, dim] = np.clip(positions[i, dim], bounds[dim][0], bounds[dim][1]) print(f"Iteration {iteration+1}/{num\_iterations}, Best SSE: {global\_best\_value:.5f}") print("\nBest parameters found:") print("theta\_0 =", global\_best\_position[0]) print("theta\_1 =", global\_best\_position[1]) print("Minimum sum of squared errors:", global\_best\_value)

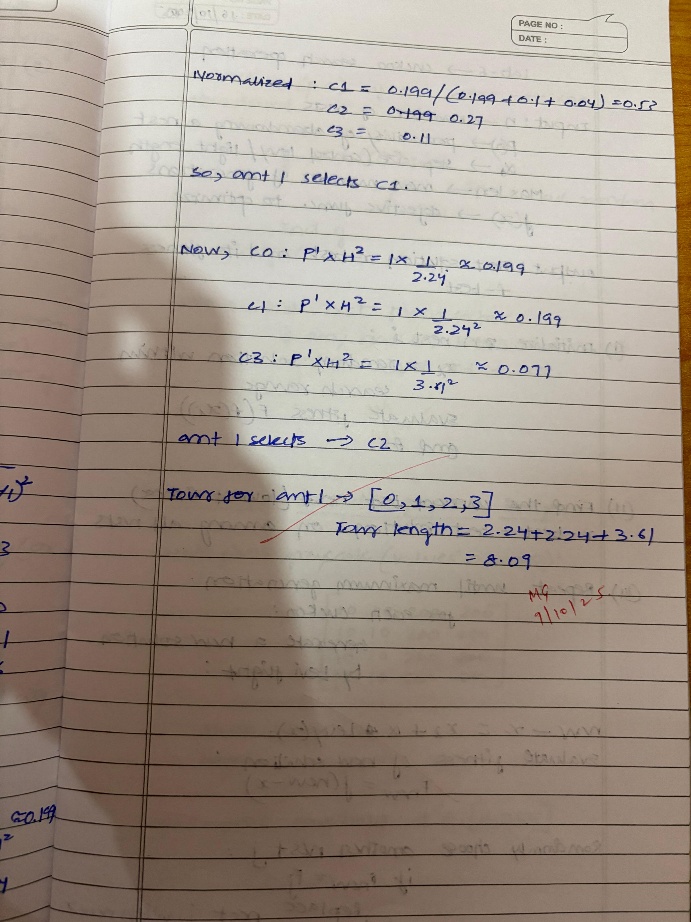
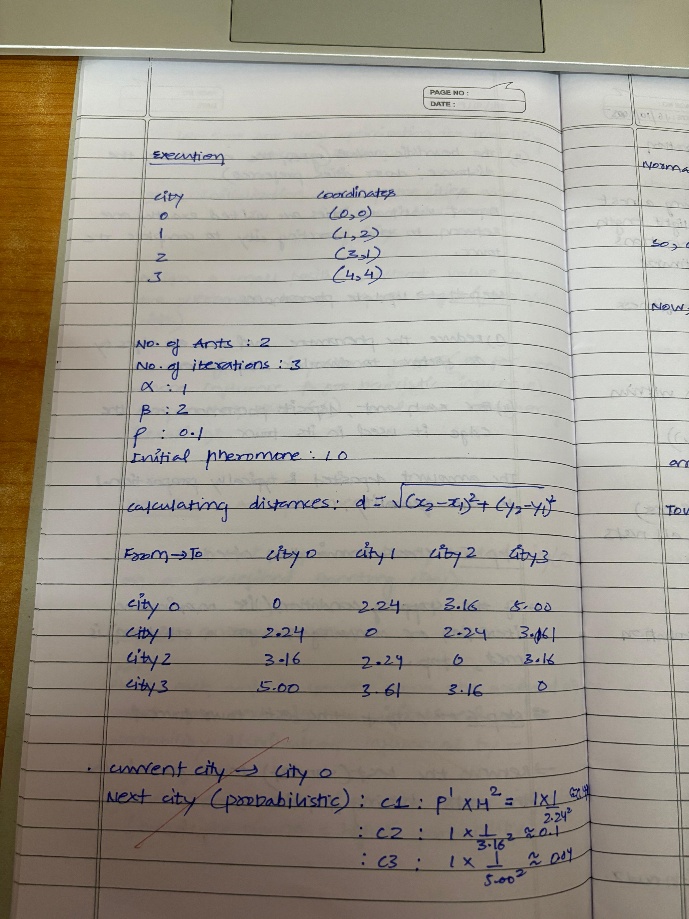
## Program 4 : Ant Colony Optimization

**Problem statement:**

The foraging behavior of ants has inspired the development of optimization algorithms that can solve complex problems such as the Traveling Salesman Problem (TSP). Ant Colony Optimization (ACO) simulates the way ants find the shortest path between food sources and their nest. Implement the ACO algorithm using Python to solve the TSP, where the objective is to find the shortest possible route that visits a list of cities and returns to the origin city.

**Algorithm:**





**Code:**

import numpy as np import random NUM\_CITIES = 10

NUM\_ANTS = 20

NUM\_ITERATIONS = 100

ALPHA = 1.0

BETA = 5.0

EVAPORATION = 0.5

Q = 100 np.random.seed(42)

cities = np.random.rand(NUM\_CITIES, 2) \* 100 dist\_matrix = np.sqrt(((cities[:, np.newaxis, :] - cities[np.newaxis, :, :]) \*\* 2).sum(axis=2)) pheromone = np.ones((NUM\_CITIES, NUM\_CITIES))

best\_distance = float('inf') best\_path = [] for iteration in range(NUM\_ITERATIONS): all\_paths = [] all\_distances = [] for ant in range(NUM\_ANTS):

path = [random.randint(0, NUM\_CITIES - 1)] while len(path) < NUM\_CITIES:

current\_city = path[-1] probabilities = [] for next\_city in range(NUM\_CITIES): if next\_city not in path:

tau = pheromone[current\_city][next\_city] \*\* ALPHA eta = (1 / dist\_matrix[current\_city][next\_city]) \*\* BETA probabilities.append(tau \* eta) else:

probabilities.append(0) probabilities = np.array(probabilities) probabilities /= probabilities.sum() next\_city = np.random.choice(range(NUM\_CITIES), p=probabilities) path.append(next\_city)

path.append(path[0]) # Return to starting city distance = sum(dist\_matrix[path[i]][path[i + 1]] for i in range(NUM\_CITIES)) all\_paths.append(path) all\_distances.append(distance) if distance < best\_distance: best\_distance = distance best\_path = path

pheromone \*= (1 - EVAPORATION) for i in range(NUM\_ANTS): for j in range(NUM\_CITIES): from\_city = all\_paths[i][j] to\_city = all\_paths[i][j + 1] pheromone[from\_city][to\_city] += Q / all\_distances[i] pheromone[to\_city][from\_city] += Q / all\_distances[i] if iteration % 10 == 0 or iteration == NUM\_ITERATIONS - 1:

print(f"Iteration {iteration}: Best Distance = {best\_distance:.2f}") print("\nBest Path Found:") print(" -> ".join(map(str, best\_path)))

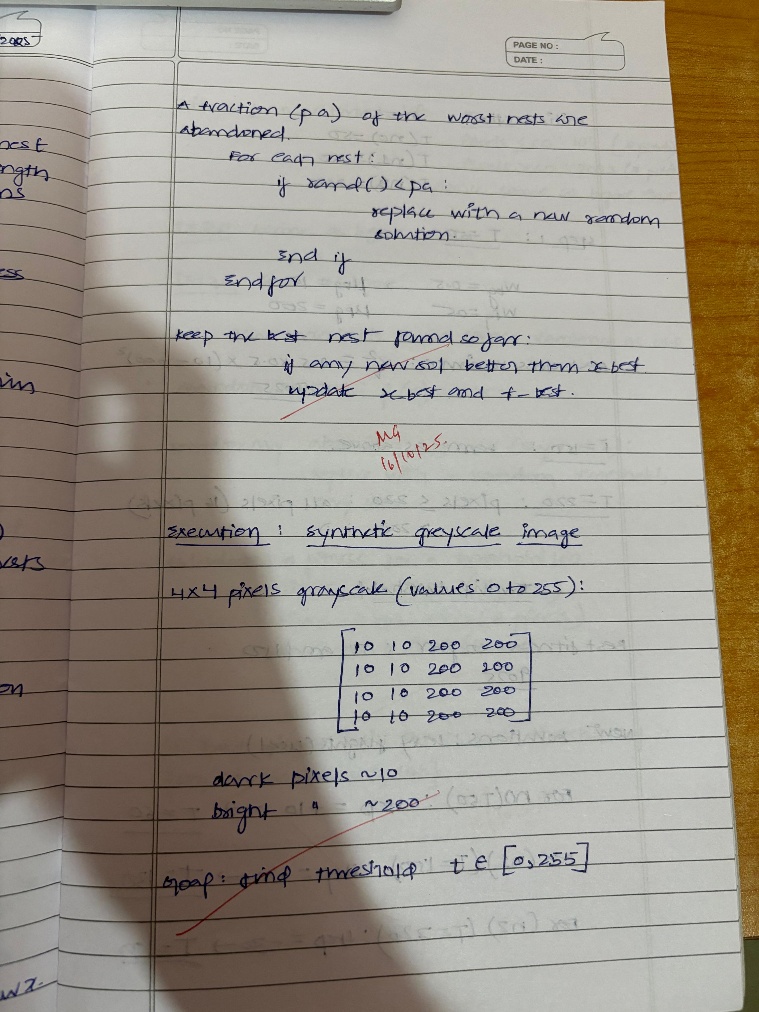
print(f"Total Distance: {best\_distance:.2f}")

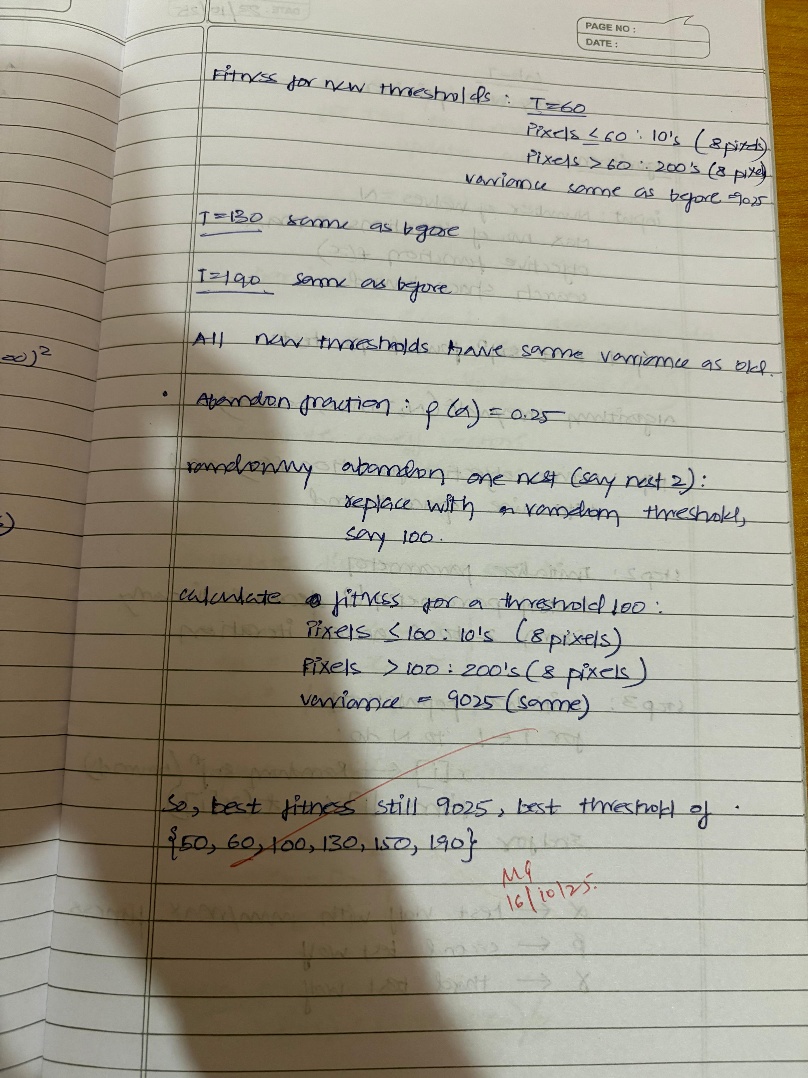
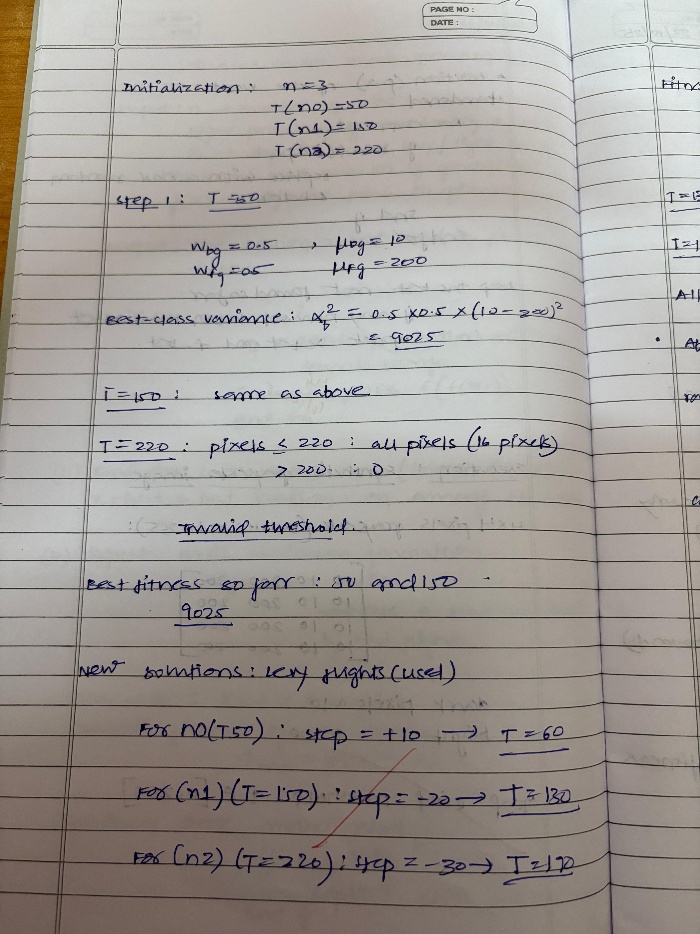
## Program 5 : Cuckoo search Optimization

**Problem statement:**

Cuckoo Search (CS) is a nature-inspired optimization algorithm based on the brood parasitism of some cuckoo species. This behavior involves laying eggs in the nests of other birds, leading to the optimization of survival strategies. CS uses Lévy flights to generate new solutions, promoting global search capabilities and avoiding local minima. The algorithm is widely used for solving continuous optimization problems and has applications in various domains, including engineering design, machine learning, and data mining.

**Algorithm:**





**Code:**

import random import math weights = [10, 20, 30, 40, 15, 25, 35] values = [60, 100, 120, 240, 80, 150, 200] capacity = 100 # Max weight capacity of the truck n\_items = len(weights) n\_nests = 15 max\_iter = 50 pa = 0.25 def fitness(solution):

total\_weight = sum(w for w, s in zip(weights, solution) if s == 1) total\_value = sum(v for v, s in zip(values, solution) if s == 1) if total\_weight > capacity:

return 0 # Penalize overweight solutions else:

return total\_value def generate\_nest():

return [random.randint(0, 1) for \_ in range(n\_items)] def levy\_flight(Lambda=1.5):

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

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

u = random.gauss(0, sigma\_u) v = random.gauss(0, 1) step = u / (abs(v) \*\* (1 / Lambda))

return step def get\_cuckoo(nest, best\_nest):

new\_nest = [] for xi, bi in zip(nest, best\_nest):

step = levy\_flight() val = xi + step \* (xi - bi) s = 1 / (1 + math.exp(-val)) new\_val = 1 if s > 0.5 else 0 new\_nest.append(new\_val) return new\_nest def cuckoo\_search():

nests = [generate\_nest() for \_ in range(n\_nests)] fitness\_values = [fitness(nest) for nest in nests] best\_index = fitness\_values.index(max(fitness\_values)) best\_nest = nests[best\_index][:] best\_fitness = fitness\_values[best\_index] for \_ in range(max\_iter): for i in range(n\_nests):

new\_nest = get\_cuckoo(nests[i], best\_nest) new\_fitness = fitness(new\_nest) if new\_fitness > fitness\_values[i]: nests[i] = new\_nest fitness\_values[i] = new\_fitness for i in range(n\_nests): if random.random() < pa: nests[i] = generate\_nest() fitness\_values[i] = fitness(nests[i])

current\_best\_index = fitness\_values.index(max(fitness\_values)) current\_best\_fitness = fitness\_values[current\_best\_index] if current\_best\_fitness > best\_fitness: best\_fitness = current\_best\_fitness best\_nest = nests[current\_best\_index][:] return best\_nest, best\_fitness if \_\_name\_\_ == "\_\_main\_\_":

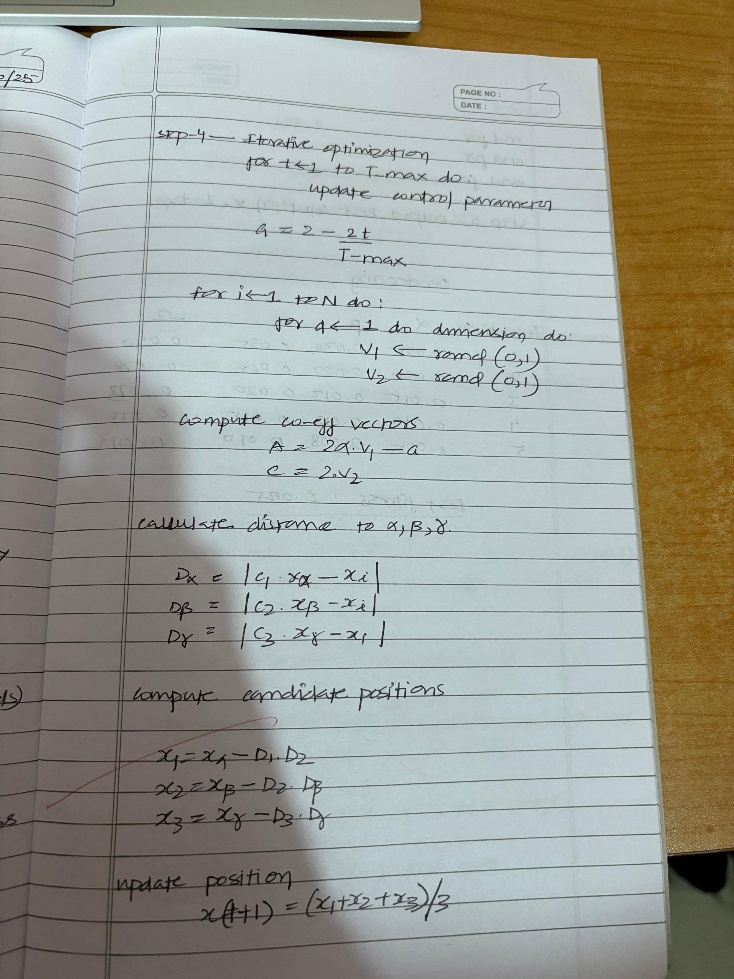
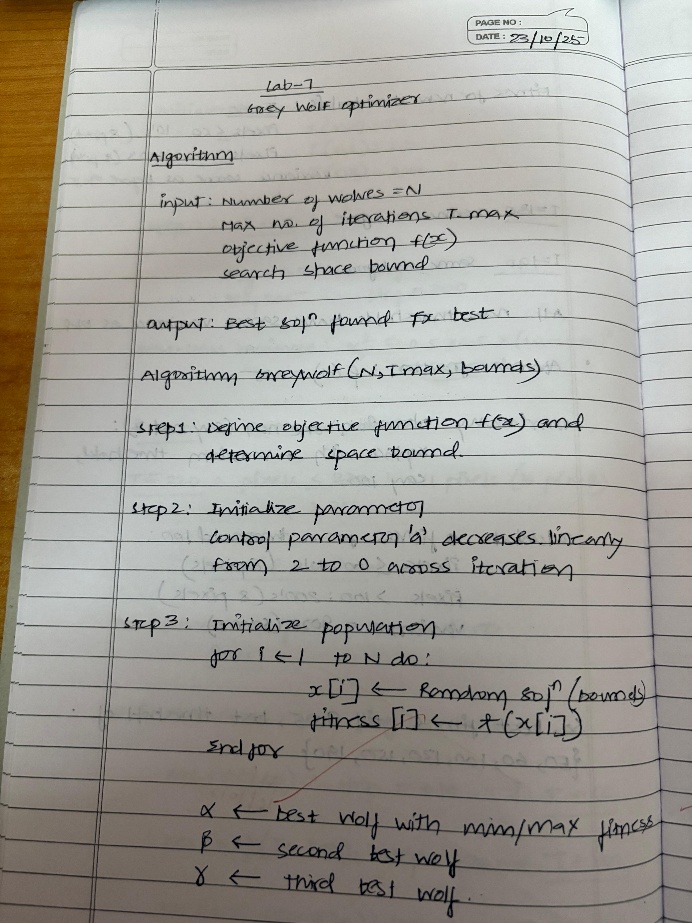
best\_solution, best\_value = cuckoo\_search() total\_weight = sum(w for w, s in zip(weights, best\_solution) if s == 1) print(f"Best packing solution (1 = selected): {best\_solution}") print(f"Total value of supplies packed: {best\_value}") print(f"Total weight: {total\_weight}")

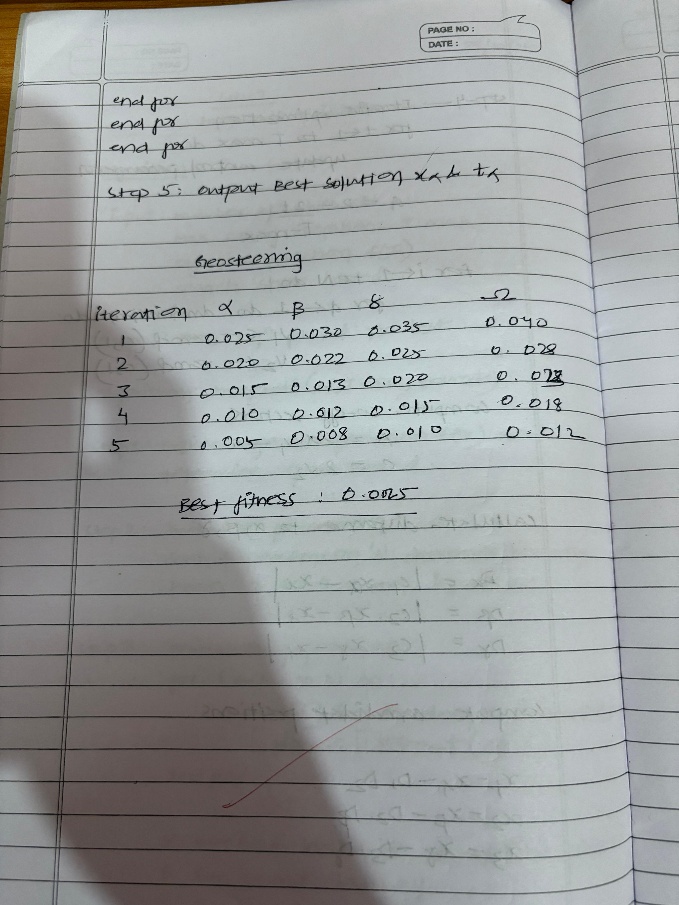
## Program 6 : Grey Wolf Optimization

**Problem statement:**

The Grey Wolf Optimizer (GWO) algorithm is a swarm intelligence algorithm inspired by the social hierarchy and hunting behavior of grey wolves. It mimics the leadership structure of alpha, beta, delta, and omega wolves and their collaborative hunting strategies. The GWO algorithm uses these social hierarchies to model the optimization process, where the alpha wolves guide the search process while beta and delta wolves assist in refining the search direction. This algorithm is effective for continuous optimization problems and has applications in engineering, data analysis, and machine learning.

**Algorithm:**





**Code:**

import numpy as np def gwo(obj\_func, dim, search\_space, n\_agents=20, max\_iter=100):

lb, ub = search\_space wolves = np.random.uniform(lb, ub, (n\_agents, dim)) alpha, beta, delta = None, None, None alpha\_score, beta\_score, delta\_score = float("inf"), float("inf"), float("inf") for t in range(max\_iter): for i in range(n\_agents):

fitness = obj\_func(wolves[i]) if fitness < alpha\_score:

delta\_score, delta = beta\_score, beta beta\_score, beta = alpha\_score, alpha alpha\_score, alpha = fitness, wolves[i].copy() elif fitness < beta\_score:

delta\_score, delta = beta\_score, beta beta\_score, beta = fitness, wolves[i].copy() elif fitness < delta\_score:

delta\_score, delta = fitness, wolves[i].copy() a = 2 - 2 \* (t / max\_iter) for i in range(n\_agents): 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] - wolves[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] - wolves[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] - wolves[i][j])

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

wolves[i][j] = np.clip((X1 + X2 + X3) / 3, lb, ub) return alpha, alpha\_score grid\_size = (20, 20) start, goal = np.array([0, 0]), np.array([19, 19]) obstacles = [

(5, 5, 10, 10),

(12, 0, 14, 14),

(3, 15, 15, 17)

]

def is\_collision(point): x, y = point.astype(int) if x < 0 or y < 0 or x >= grid\_size[0] or y >= grid\_size[1]:

return True for ox1, oy1, ox2, oy2 in obstacles: if ox1 <= x <= ox2 and oy1 <= y <= oy2: return True return False

waypoints = waypoints.reshape(-1, 2) path = [start] + [w.astype(int) for w in waypoints] + [goal] total\_dist, penalty = 0, 0 for i in range(len(path) - 1):

dist = np.linalg.norm(path[i + 1] - path[i])

total\_dist += dist if is\_collision(path[i + 1]):

penalty += 100

energy = 0 for i in range(1, len(path) - 1): v1 = path[i] - path[i - 1] v2 = path[i + 1] - path[i] if np.linalg.norm(v1) > 0 and np.linalg.norm(v2) > 0:

cos\_angle = np.dot(v1, v2) / (np.linalg.norm(v1) \* np.linalg.norm(v2)) angle = np.arccos(np.clip(cos\_angle, -1, 1)) energy += angle return total\_dist + energy \* 5 + penalty n\_waypoints = 5 # intermediate waypoints dim = n\_waypoints \* 2 best\_path, best\_score = gwo(path\_cost, dim, (0, grid\_size[0]-1), n\_agents=30, max\_iter=200) best\_waypoints = best\_path.reshape(-1, 2).astype(int)

final\_path = np.vstack([start, best\_waypoints, goal]) clean\_path = [] for p in final\_path:

pt = tuple(map(int, p)) if len(clean\_path) == 0 or pt != clean\_path[-1]:

clean\_path.append(pt) print("Best Path Found:") for p in clean\_path:

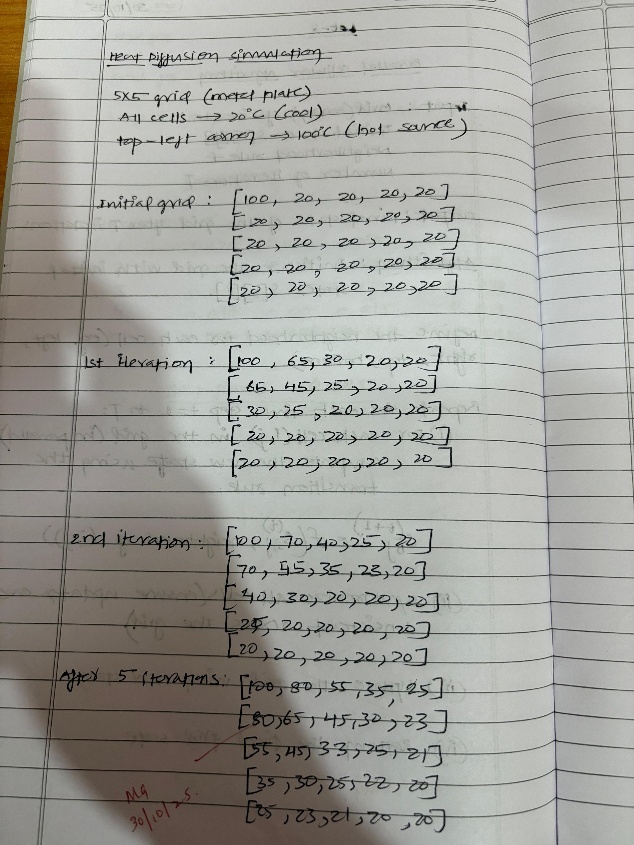
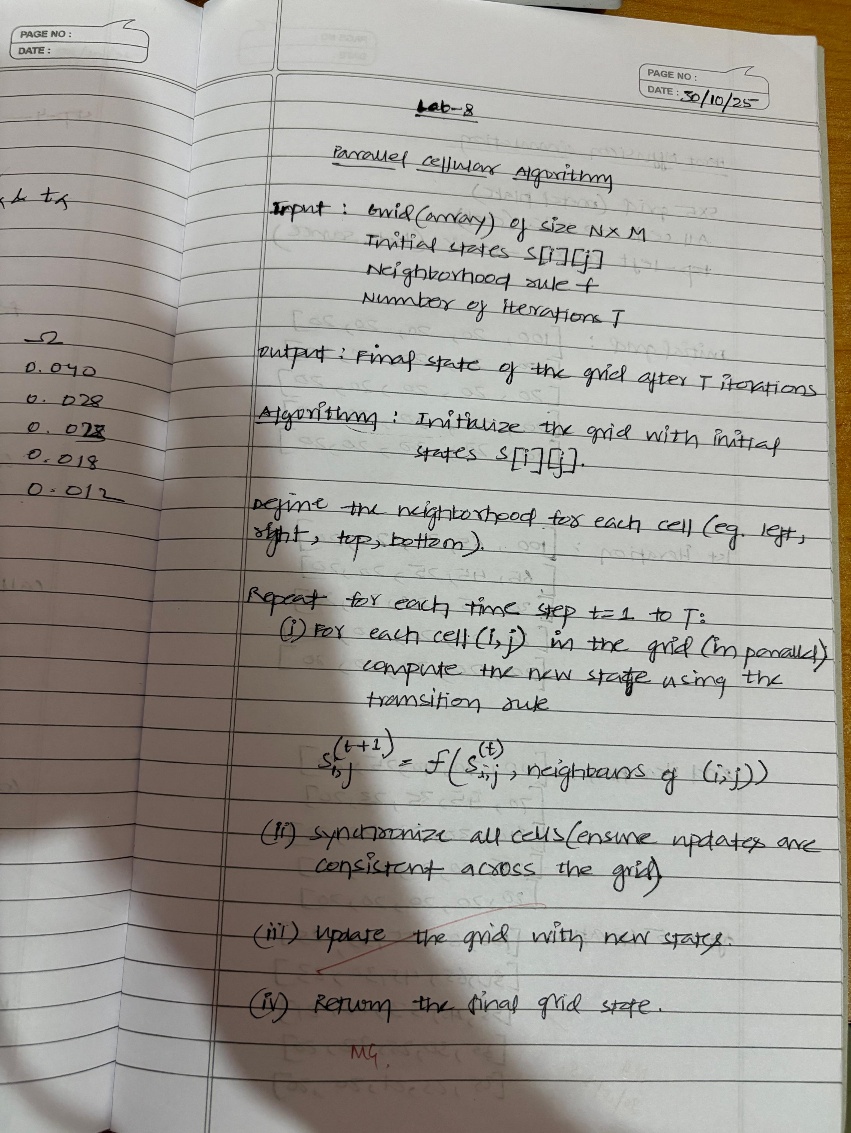
print(p) print("\nPath Cost:", round(best\_score, 2))

## Program 7 : Parallel cellular Optimization

**Problem statement:**

Parallel Cellular Algorithms are inspired by the functioning of biological cells that operate in a highly parallel and distributed manner. These algorithms leverage the principles of cellular automata and parallel computing to solve complex optimization problems efficiently. Each cell represents a potential solution and interacts with its neighbors to update its state based on predefined rules. This interaction models the diffusion of information across the cellular grid, enabling the algorithm to explore the search space effectively. Parallel Cellular Algorithms are particularly suitable for large-scale optimization problems and can be implemented on parallel computing architectures for enhanced performance.

**Algorithm:**



**Code:**

import numpy as np import random from itertools import permutations

distance\_matrix = np.array([

[0, 2, 9, 10],

[2, 0, 6, 4],

[9, 6, 0, 8],

[10, 4, 8, 0]

])

num\_customers = distance\_matrix.shape[0] - 1 population\_size = 9 grid\_dim = (3, 3) num\_vehicles = 2 def generate\_individual(): perm = list(range(1, num\_customers + 1)) random.shuffle(perm)

return perm population = [generate\_individual() for \_ in range(population\_size)] def fitness(individual): split\_points = np.linspace(0, num\_customers, num\_vehicles + 1, dtype=int) total\_distance = 0 for i in range(num\_vehicles):

route = [0] + individual[split\_points[i]:split\_points[i+1]] + [0] # depot at start and end for j in range(len(route) - 1):

total\_distance += distance\_matrix[route[j], route[j+1]] return total\_distance def get\_neighbors(idx):

r, c = divmod(idx, grid\_dim[1]) neighbors = [] for dr in [-1, 0, 1]: for dc in [-1, 0, 1]:

nr, nc = r + dr, c + dc if 0 <= nr < grid\_dim[0] and 0 <= nc < grid\_dim[1]:

n\_idx = nr \* grid\_dim[1] + nc if n\_idx != idx:

neighbors.append(n\_idx)

return neighbors def crossover(parent1, parent2):

size = len(parent1) a, b = sorted(random.sample(range(size), 2)) child = [None] \* size child[a:b] = parent1[a:b] pointer = b for gene in parent2[b:] + parent2[:b]: if gene not in child: if pointer == size:

pointer = 0 child[pointer] = gene pointer += 1 return child def mutate(individual):

a, b = random.sample(range(len(individual)), 2) individual[a], individual[b] = individual[b], individual[a] return individual def pca\_iteration(pop): new\_pop = pop.copy() for idx in range(len(pop)):

neighbors = get\_neighbors(idx) partner\_idx = random.choice(neighbors)

parent1 = pop[idx] parent2 = pop[partner\_idx] child = crossover(parent1, parent2) if random.random() < 0.2: child = mutate(child) if fitness(child) < fitness(pop[idx]):

new\_pop[idx] = child

return new\_pop num\_generations = 25 for gen in range(num\_generations):

population = pca\_iteration(population) best\_fitness = min(fitness(ind) for ind in population) print(f"Generation {gen+1}: Best total distance = {best\_fitness}") best\_individual = min(population, key=fitness) print("\nBest route assignment (split evenly):") split\_points = np.linspace(0, num\_customers, num\_vehicles + 1, dtype=int) for i in range(num\_vehicles):

route = [0] + best\_individual[split\_points[i]:split\_points[i+1]] + [0] print(f"Vehicle {i+1} route: {route}")

print(f"Total distance: {fitness(best\_individual)}")