VISVESVARAYA TECHNOLOGICAL UNIVERSITY

"JnanaSangama", Belgaum -590014, Karnataka.

LAB RECORD

Bio Inspired Systems (23CS5BSBIS)

Submitted by

Sanghamitra R(1BM22CS237)

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
Sep-2024 to Jan-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 **Sanghamitra R(1BM22CS237),** 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.

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

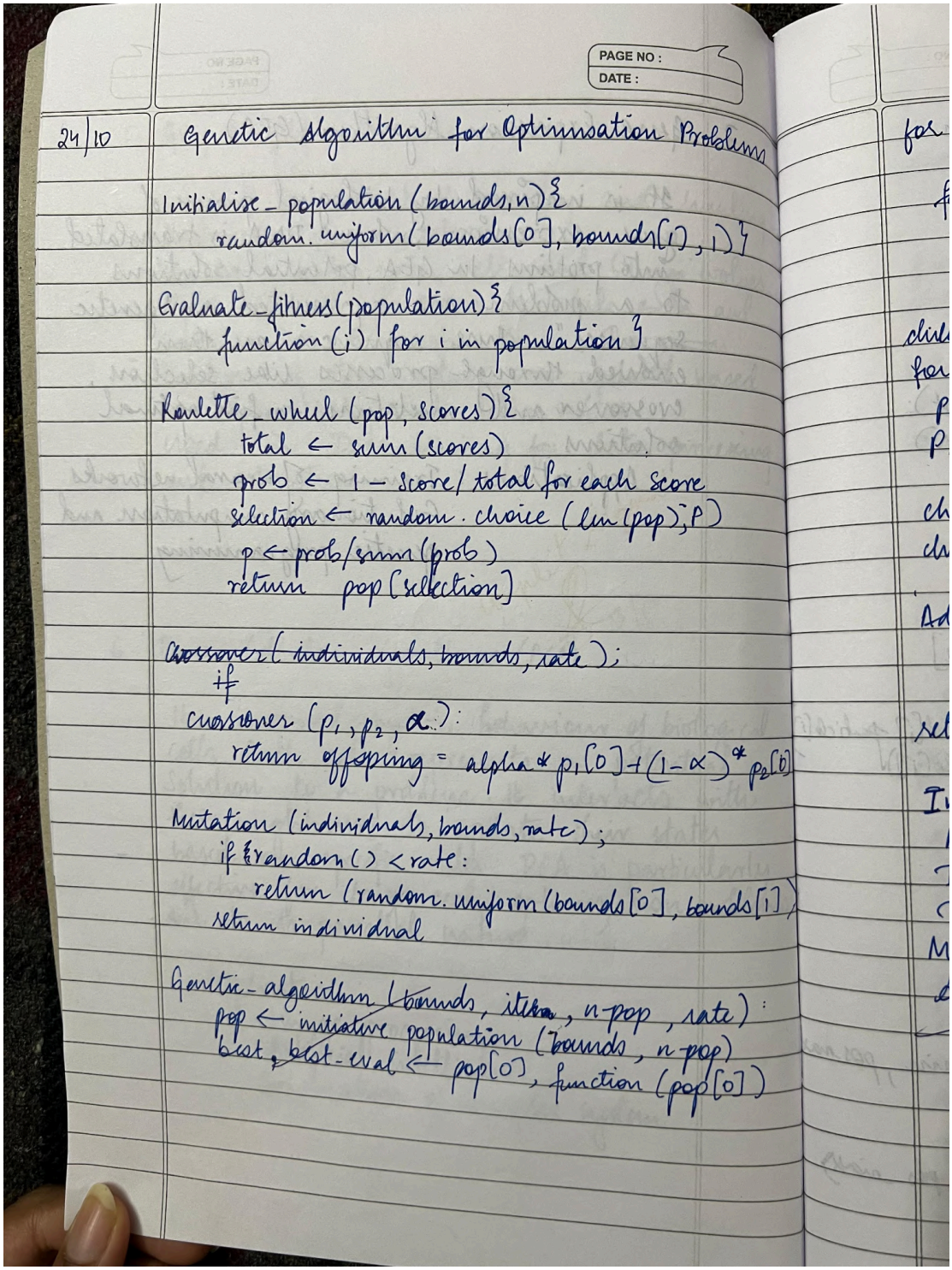
Index

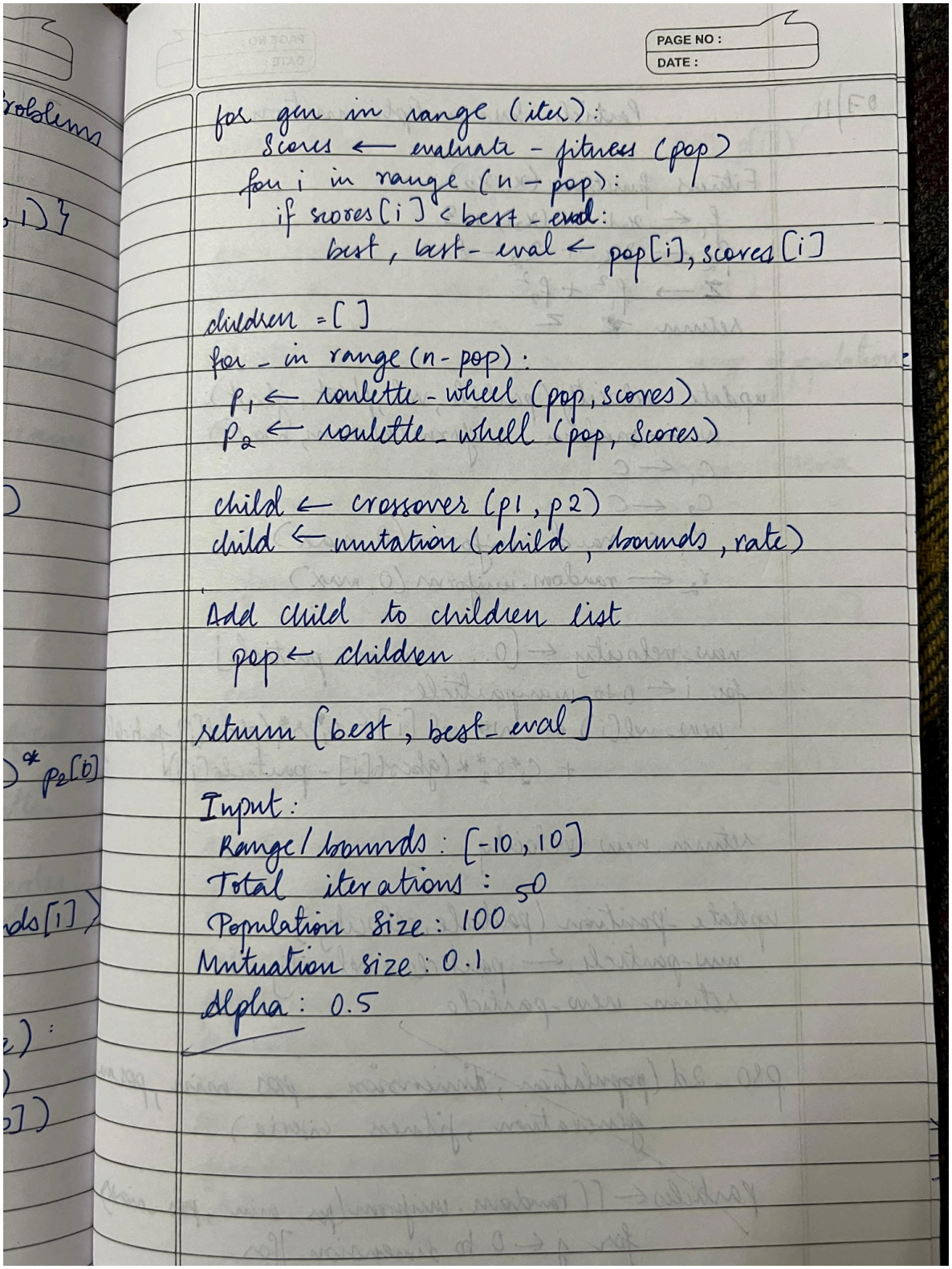
Github Link: https://github.com/sanghamitrarajagopal/BIS_LAB_CS237

Program 1
Genetic Algorithm for Optimization Problems.

Implement a Genetic Algorithm using Python to solve a basic optimization problem, such as finding the maximum value of a mathematical function.

Algorithm:





```
for gen in range (iter):
    scores ← evaluate - fitness (pop)
    for i in range (n - pop):
        if scores[i] < best - eval:
            best, best - eval ← pop[i], scores[i]

children = [ ]
for _ in range (n - pop):
    p1 ← roulette - wheel (pop, scores)
    p2 ← roulette - wheel (pop, scores)

child ← crossover (p1, p2)
child ← mutation (child, bounds, rate)

Add child to children list
pop ← children

return [best, best_eval]
```

Input:
Range / bounds : [-10, 10]
Total iterations : 50
Population size : 100
Mutuation size : 0.1
Alpha : 0.5

CODE:

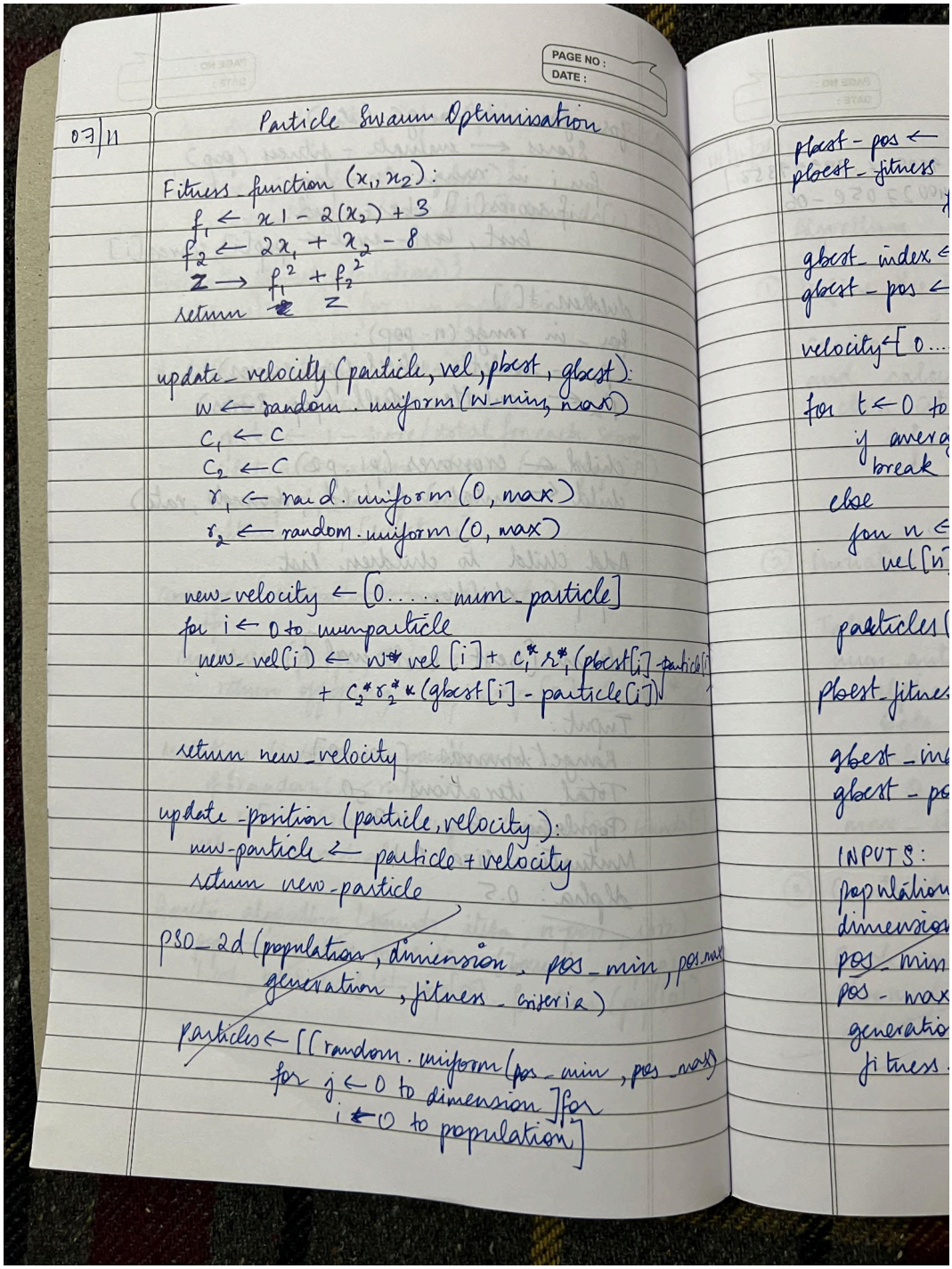
```
import random

# Define the fitness function
def fitness_function(x):
    return x**2  # Example function: f(x) = x^2

# Generate initial population
def generate_population(size, x_min, x_max):
    return [random.uniform(x_min, x_max) for _ in range(size)]

# Selection process
def select_parents(population, fitnesses):
    total_fitness = sum(fitnesses)
    selection_probs = [f / total_fitness for f in fitnesses]
    parents = random.choices(population, weights=selection_probs, k=2)
    return parents

# Crossover process
def crossover(parent1, parent2):
    alpha = random.random()
    child = alpha * parent1 + (1 - alpha) * parent2
    return child

# Mutation process
def mutate(child, mutation_rate, x_min, x_max):
    if random.random() < mutation_rate:
        child = random.uniform(x_min, x_max)
    return child

# Genetic Algorithm
def genetic_algorithm(pop_size, generations, mutation_rate, x_min, x_max):
    population = generate_population(pop_size, x_min, x_max)
    for generation in range(generations):
        fitnesses = [fitness_function(ind) for ind in population]
        new_population = []
        for _ in range(pop_size):
            parent1, parent2 = select_parents(population, fitnesses)
            child = crossover(parent1, parent2)
            child = mutate(child, mutation_rate, x_min, x_max)
            new_population.append(child)
        population = new_population
    best_solution = max(population, key=fitness_function)
    return best_solution

# User inputs
pop_size = int(input("Enter population size: "))
generations = int(input("Enter number of generations: "))
```

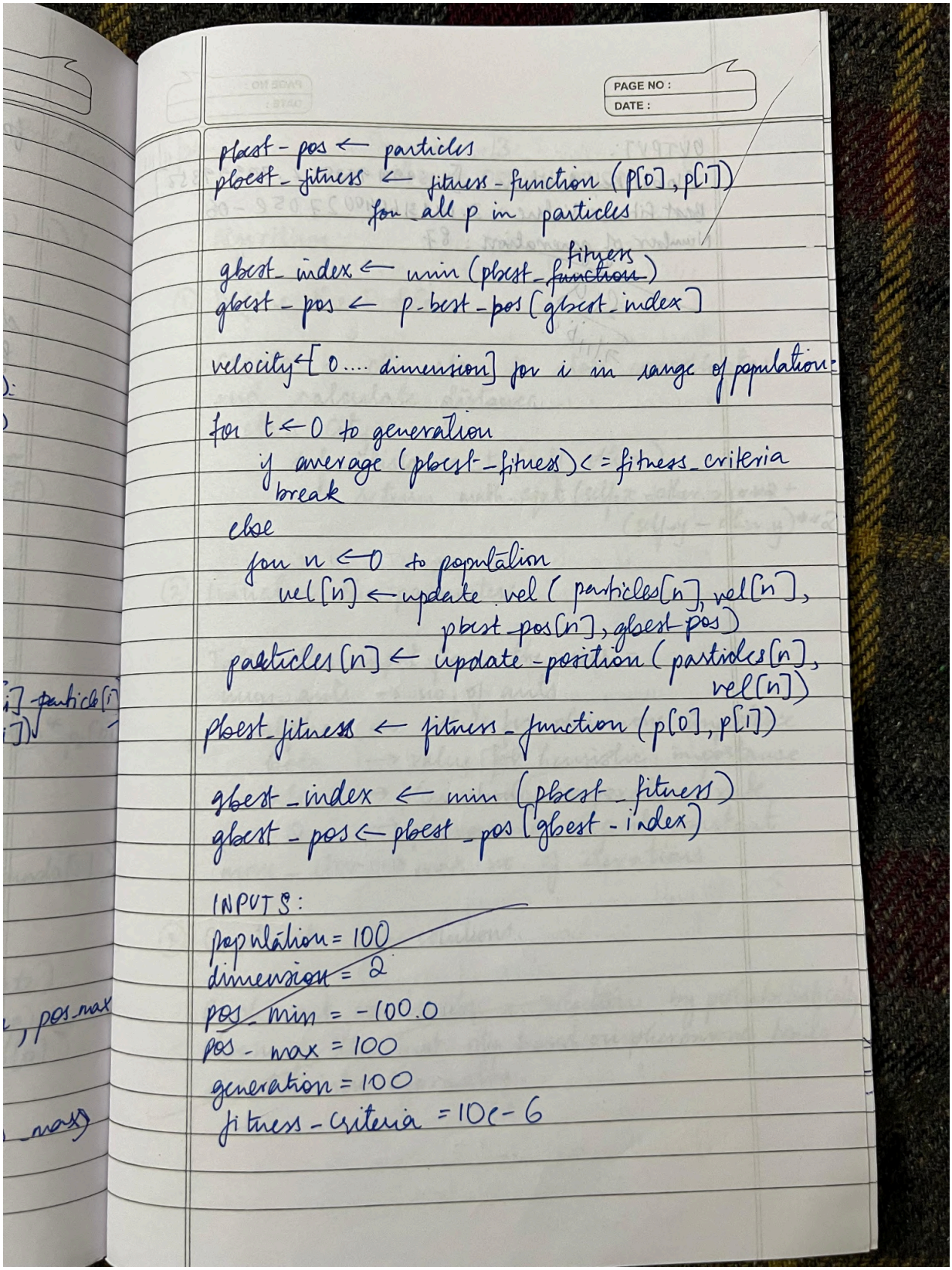
```
mutation_rate = float(input("Enter mutation rate (0-1): "))
x_min = float(input("Enter minimum value of x: "))
x_max = float(input("Enter maximum value of x: "))

# Run the genetic algorithm
best_solution = genetic_algorithm(pop_size, generations, mutation_rate, x_min, x_max)
print(f"The best solution found is: {best_solution} with fitness value: {fitness_function(best_solution)}")
```

Program 2
Particle Swarm Optimization for Function Optimization.

Implement the PSO algorithm using Python to optimize a mathematical function.

Algorithm:





pbest_pos ← particles
pbest_fitness ← fitness_function(p[0], p[i])
for all p in particles

gbest_index ← min(pbest_fitness)
gbest_pos ← p_best_pos[gbest_index]

velocity ← [0.... dimension] for i in range of population:

for t ← 0 to generation
if average(pbest_fitness) <= fitness_criteria
break
else
for n ← 0 to population
vel[n] ← update_vel(particles[n], vel[n], pbest_pos[n], gbest_pos)
particles[n] ← update_position(particles[n], vel[n])
pbest_fitness ← fitness_function(p[0], p[i])

gbest_index ← min(pbest_fitness)
gbest_pos ← pbest_pos[gbest_index]

INPUTS:
population = 100
dimension = 2
pos_min = -100.0
pos_max = 100
generation = 100
fitness_criteria = 10e-6

PAGE NO :
DATE :

OUTPUT:

Global Best Position : [2.59999538 2.79999358]

Best fitness Value : 3.14131604002705e-06

Number of generation : 87

14/11/24

Prog

ANT CO

Algorithm :

(1) Define the Prob

Create a city
and calcula
class City:
def - dis
set

(2) Initialise the

Take the in
num-ants
alpha
beta
rho
Q
max-iter

(3) Construct t

Each ant c
choosing th
and heuristi

CODE:

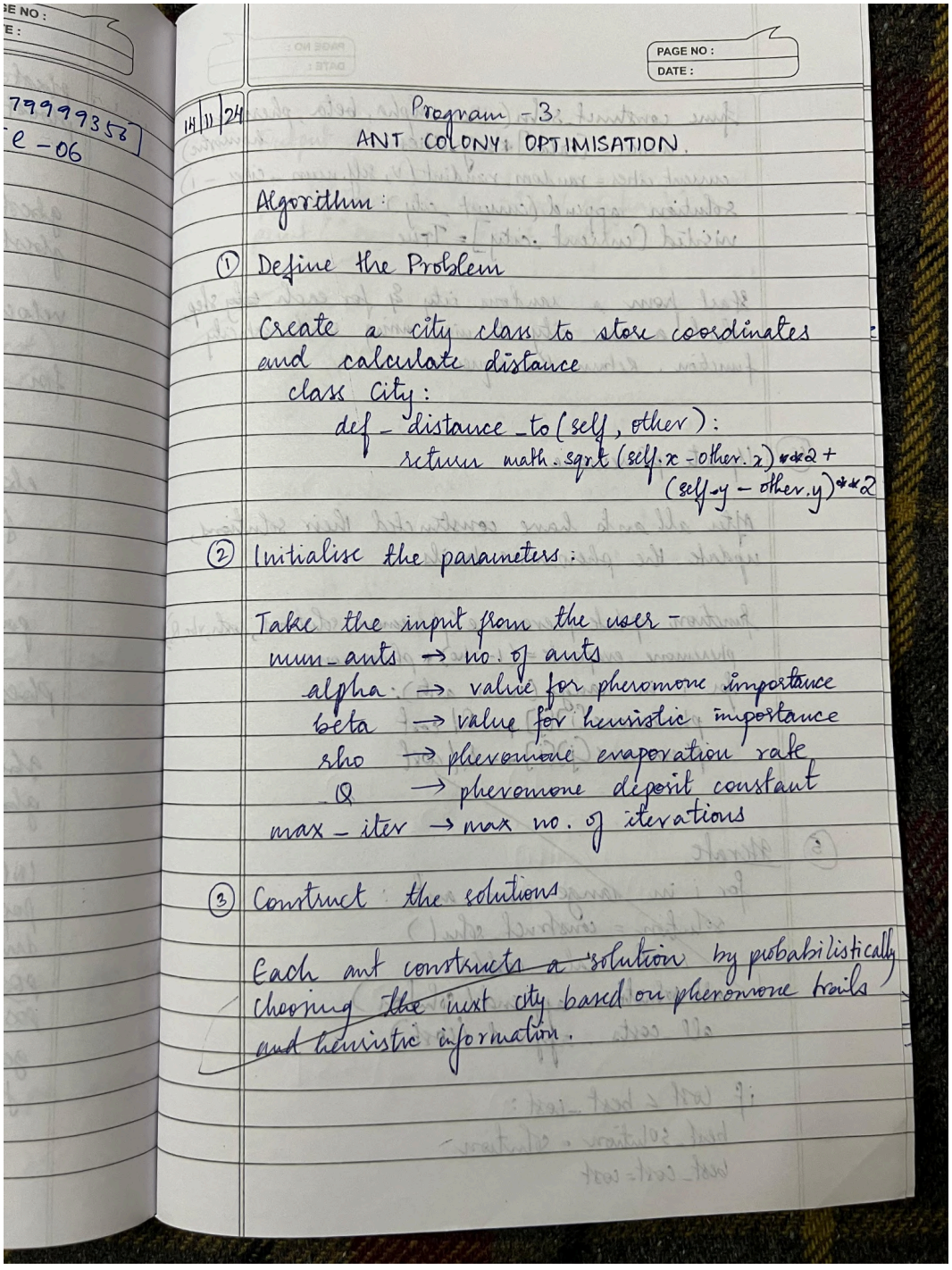
```
import numpy as np

# Objective function (Example: Rastrigin function)
def objective_function(position):
    return sum([x**2 - 10 * np.cos(2 * np.pi * x) + 10 for x in position])

# Particle Swarm Optimization
class Particle:
    def __init__(self, dimensions):
        self.position = np.random.uniform(-10, 10, dimensions)  # Initialize position
        self.velocity = np.random.uniform(-1, 1, dimensions)    # Initialize velocity
        self.best_position = self.position.copy()               # Personal best position
        self.best_score = float('inf')                          # Best score for personal best

    def update_velocity(self, global_best_position, inertia, cognitive_const, social_const):
        r1, r2 = np.random.rand(), np.random.rand()
        cognitive = cognitive_const * r1 * (self.best_position - self.position)
        social = social_const * r2 * (global_best_position - self.position)
        self.velocity = inertia * self.velocity + cognitive + social

    def update_position(self):
        self.position += self.velocity

# PSO Algorithm
def particle_swarm_optimization(objective_func, dimensions, num_particles, max_iter):
    inertia = 0.5           # Inertia weight
    cognitive_const = 1.5    # Cognitive constant
    social_const = 1.5       # Social constant

    # Initialize particles
    swarm = [Particle(dimensions) for _ in range(num_particles)]
    global_best_position = np.random.uniform(-10, 10, dimensions)
    global_best_score = float('inf')

    for iteration in range(max_iter):
        for particle in swarm:
            # Evaluate fitness
            fitness = objective_func(particle.position)
            # Update personal best
            if fitness < particle.best_score:
                particle.best_score = fitness
```

```
                particle.best_position = particle.position.copy()

            # Update global best
            if fitness < global_best_score:
                global_best_score = fitness
                global_best_position = particle.position.copy()

        # Update velocity and position for each particle
        for particle in swarm:
            particle.update_velocity(global_best_position, inertia, cognitive_const, social_const)
            particle.update_position()

        print(f"Iteration {iteration+1}/{max_iter}, Global Best Score: {global_best_score}")

    return global_best_position, global_best_score

# Example usage
best_position, best_score = particle_swarm_optimization(objective_function, dimensions=2,
num_particles=30, max_iter=100)
print("Best Position:", best_position)  print("Best Score:", best_score)
```

Program 3
Ant Colony Optimization for the Traveling Salesman Problem

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:



Program - 3
ANT COLONY OPTIMISATION.

Algorithm:

① Define the Problem

Create a city class to store coordinates and calculate distance

```
class City:
    def distance_to(self, other):
        return math.sqrt((self.x - other.x)**2 +
                         (self.y - other.y)**2)
```

② Initialise the parameters:

Take the input from the user -
num_ants → no. of ants
alpha → value for pheromone importance
beta → value for heuristic importance
rho → pheromone evaporation rate
Q → pheromone deposit constant
max_iter → max no. of iterations

③ Construct the solutions

Each ant constructs a solution by probabilistically choosing the next city based on pheromone trails and heuristic information.

PAGE NO :
DATE :

func construct_soln(cities, alpha, beta, pheromone, heuristic)
visited = [False] * self.cities
current_cities = random.randint(0, self.num_cities - 1)
solution.append(current_city)
visited[current_city] = True

Start from a random city & for each ~~city~~ step, select a new_city to visit using select_next_city function. Return the sequence.

(4) Update pheromone

After all ants have constructed their solutions, update the pheromone trails.

function update_pheromone(pheromone, solutions, costs, rho, Q)
pheromone_evaporate = (1 - rho) * pheromone
for i in range(num_ants):
pheromone[i][j] += Q/cost
pheromone[j][i] += Q/cost

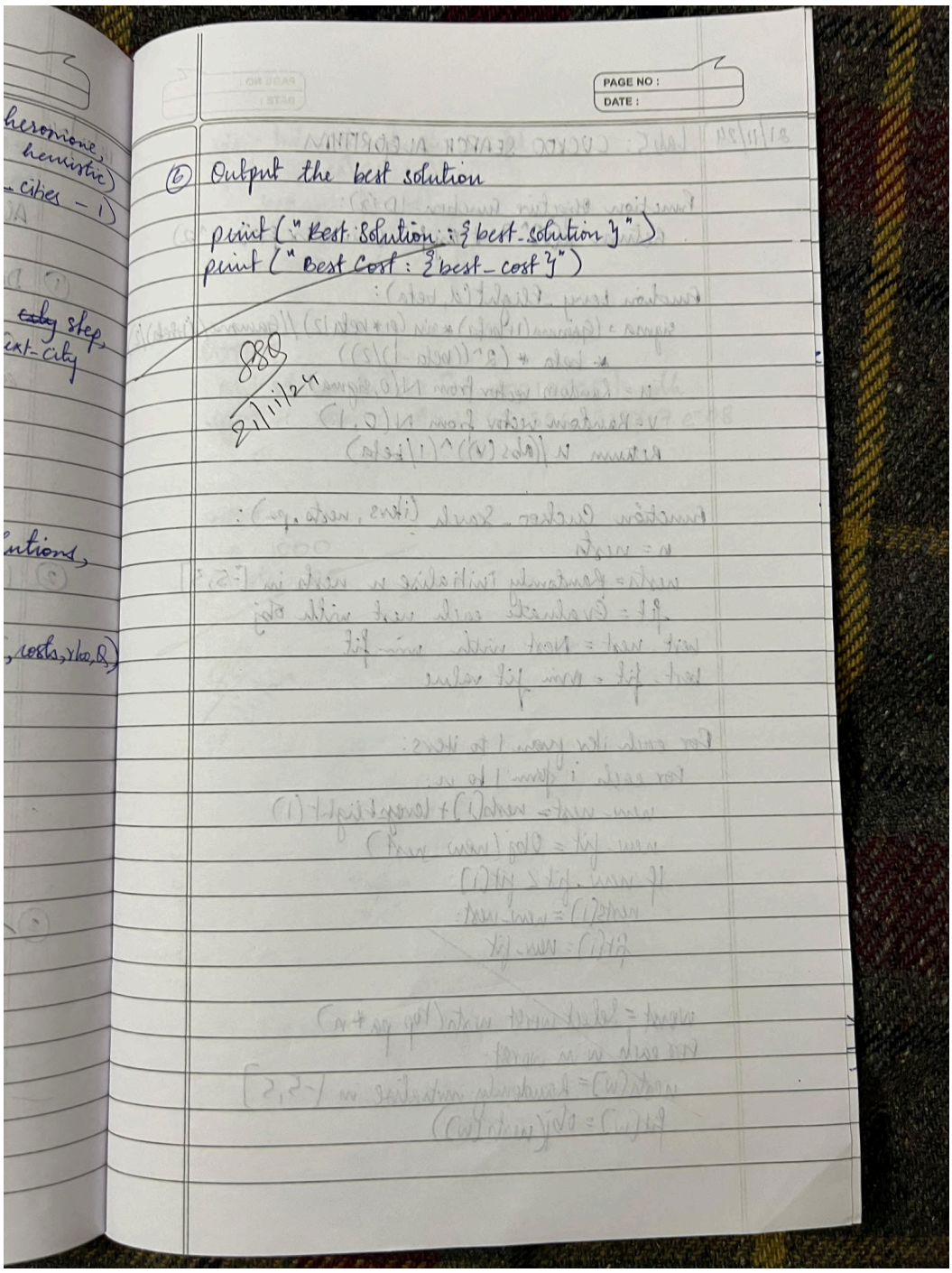
(5) Iterate

for i in range num_ants:
solution = construct_soln()
cost = calc_cost()
all_solutions.append(solution)
all_costs.append(cost)

if cost < best_cost:
best_solution = solution
best_cost = cost

(6) Output the

print("Best
print("Best



⑥ Output the best solution

print("Best Solution: {best_solution}")
print("Best Cost: {best_cost}")

CODE:

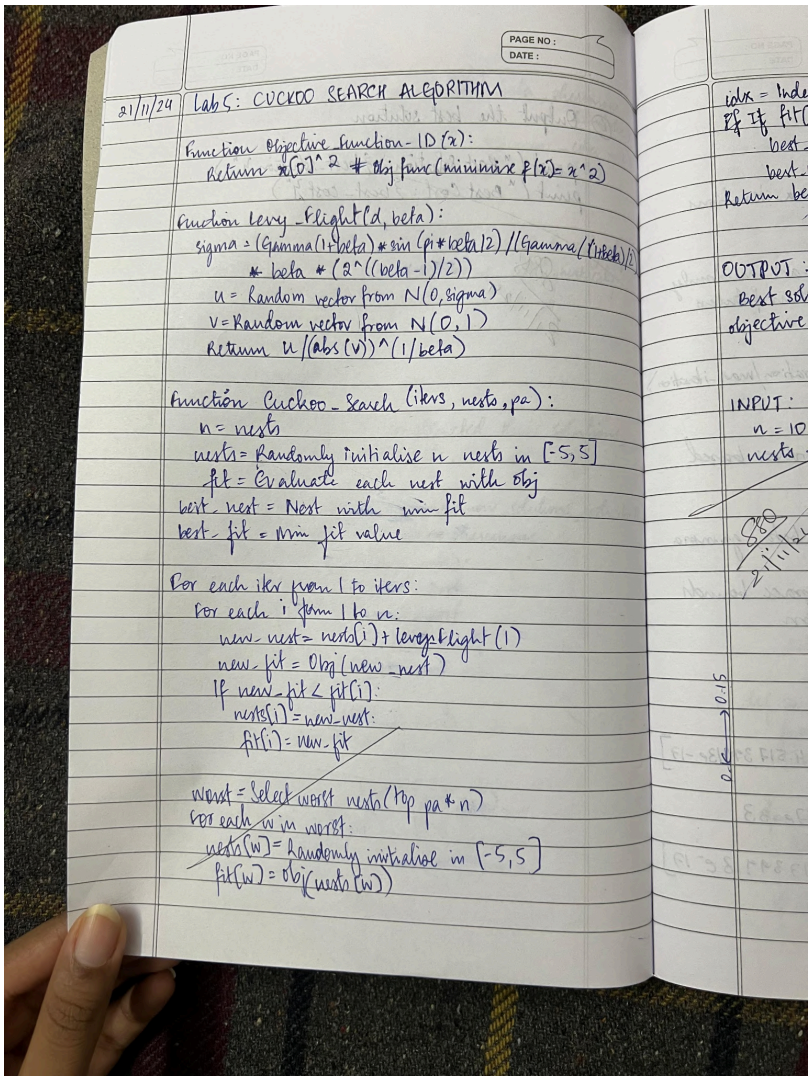
```
import numpy as np
import random

class AntColony:
    def __init__(self, distance_matrix, n_ants, n_iterations, decay, alpha=1, beta=1):
        self.distance_matrix = distance_matrix
        self.pheromone = np.ones(distance_matrix.shape) / len(distance_matrix)
        self.n_ants = n_ants
        self.n_iterations = n_iterations
        self.decay = decay
        self.alpha = alpha  # Pheromone importance
        self.beta = beta    # Distance importance
        self.all_indices = range(len(distance_matrix))

    def run(self):
        shortest_path = None
        all_time_shortest_path = ("path", np.inf)

        for _ in range(self.n_iterations):
            all_paths = self.generate_all_paths()
            self.update_pheromones(all_paths)
            shortest_path = min(all_paths, key=lambda x: x[1])
            if shortest_path[1] < all_time_shortest_path[1]:
                all_time_shortest_path = shortest_path

        return all_time_shortest_path

    def generate_all_paths(self):
        all_paths = []
        for _ in range(self.n_ants):
            path = self.generate_path(0)  # Start from city 0
            path_dist = self.calculate_path_distance(path)
            all_paths.append((path, path_dist))
        return all_paths

    def generate_path(self, start):
        path = [start]
        visited = set(path)
        while len(visited) < len(self.distance_matrix):
            move = self.select_next_city(path[-1], visited)
            path.append(move)
            visited.add(move)
        path.append(start)  # Return to starting city
        return path

    def select_next_city(self, current_city, visited):
        pheromone = np.copy(self.pheromone[current_city])
```

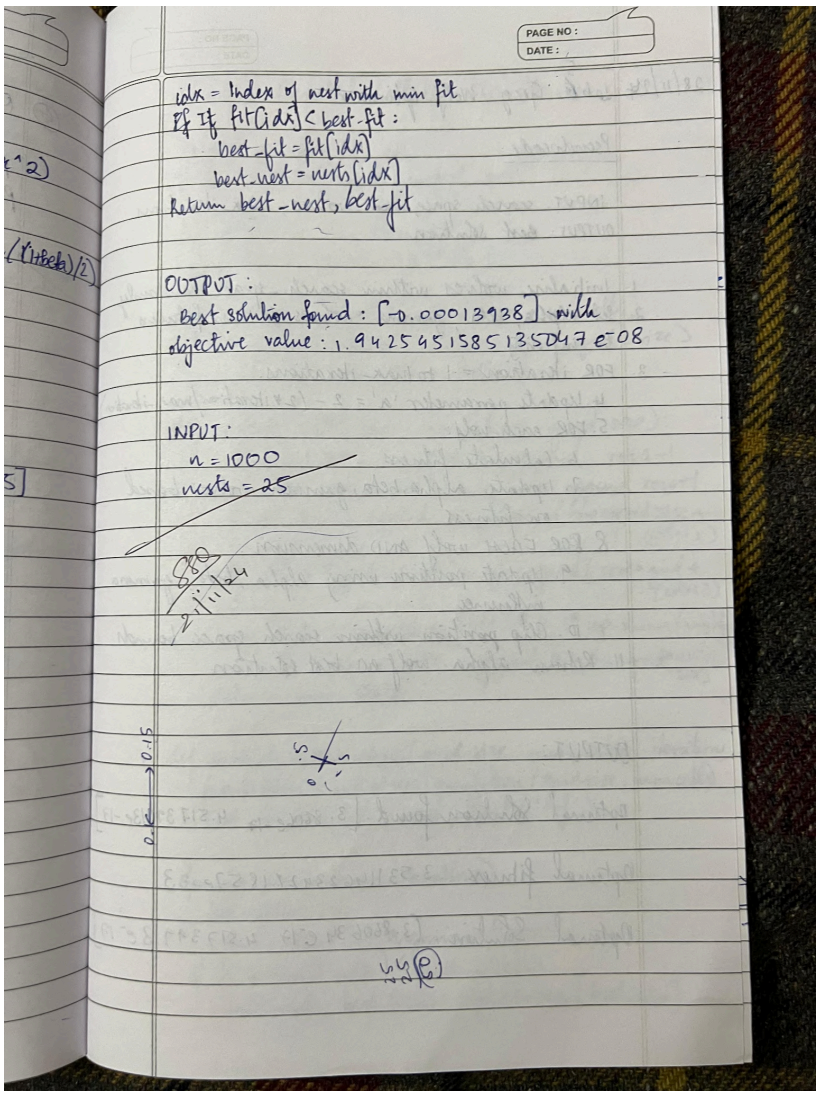
```
        pheromone[list(visited)] = 0  # Avoid visiting already visited cities

        probabilities = pheromone ** self.alpha * ((1 / self.distance_matrix[current_city]) ** self.beta)
        probabilities /= probabilities.sum()  # Normalize probabilities

        next_city = np.random.choice(self.all_indices, p=probabilities)
        return next_city

    def calculate_path_distance(self, path):
        total_dist = 0
        for i in range(len(path) - 1):
            total_dist += self.distance_matrix[path[i]][path[i + 1]]
        return total_dist

    def update_pheromones(self, all_paths):
        self.pheromone *= (1 - self.decay)  # Pheromone evaporation
        for path, dist in all_paths:
            for i in range(len(path) - 1):
                self.pheromone[path[i]][path[i + 1]] += 1 / dist  # Update pheromone based on path quality

# Example: A 4-city TSP problem
if __name__ == "__main__":
    distance_matrix = np.array([[np.inf, 12, 12, 15],
                                [12, np.inf, 13, 14],
                                [12, 13, np.inf, 11],
                                [15, 14, 11, np.inf]])

    colony = AntColony(distance_matrix, n_ants=10, n_iterations=100, decay=0.1, alpha=1, beta=2)
    best_path = colony.run()
    print("Best path found:", best_path)
```

Program 4

Cuckoo Search (CS)

Algorithm:





CODE:

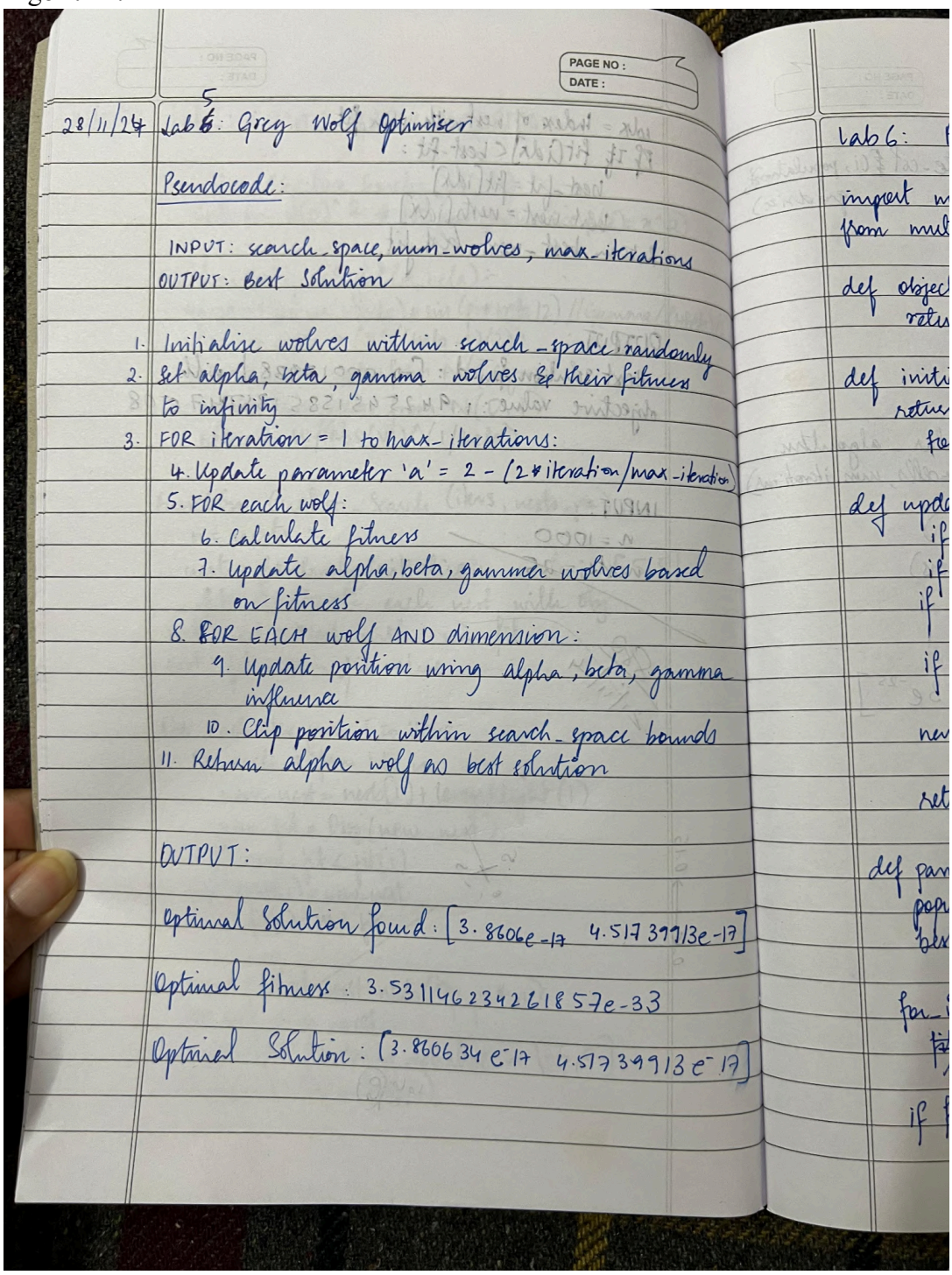
```
import numpy as np

# Objective function (example: Sphere function)
def objective_function(x):
    return np.sum(x ** 2)

# Levy flight implementation
def levy_flight(Lambda, dim, alpha=1.0):
    u = np.random.normal(0, 1, size=dim)
    v = np.random.normal(0, 1, size=dim)
    step = alpha * (u / (np.abs(v) ** (1 / Lambda)))  # Lévy step
    return step

# Cuckoo Search Algorithm
def cuckoo_search(n, max_generations, pa, lower_bound, upper_bound, dim):
    # Step 1: Initialize nests randomly
    nests = np.random.uniform(lower_bound, upper_bound, size=(n, dim))
```

```
    fitness = np.array([objective_function(nest) for nest in nests])
    best_nest = nests[np.argmin(fitness)]
    best_fitness = np.min(fitness)

    # Iterative optimization
    for t in range(max_generations):
        # Rule 1: Generate new solutions via Lévy flight
        for i in range(n):
            new_nest = nests[i] + levy_flight(1.5, dim)
            new_nest = np.clip(new_nest, lower_bound, upper_bound)
            new_fitness = objective_function(new_nest)

            # Rule 2: Replace nests if better
            if new_fitness < fitness[i]:
                nests[i] = new_nest
                fitness[i] = new_fitness

                # Update global best
                if new_fitness < best_fitness:
                    best_nest = new_nest
                    best_fitness = new_fitness

        # Rule 3: Abandon some nests and create new random ones
        abandon = np.random.rand(n) < pa
        nests[abandon] = np.random.uniform(lower_bound, upper_bound, size=(np.sum(abandon), dim))
        fitness[abandon] = np.array([objective_function(nest) for nest in nests[abandon]])

    return best_nest, best_fitness

# Parameters
n = 25  # Number of nests
dim = 5  # Dimensionality of the problem
max_generations = 100  # Max iterations
pa = 0.25  # Abandonment probability
lower_bound = -10  # Lower bound of the search space
upper_bound = 10  # Upper bound of the search space

# Run Cuckoo Search
best_solution, best_value = cuckoo_search(n, max_generations, pa, lower_bound, upper_bound, dim)

print("Best solution found:", best_solution)
print("Best objective value:", best_value)
```

Program 5

Grey Wolf Optimizer (GWO):

Algorithm:



28/11/24 Lab 5: Grey Wolf Optimiser

Pseudocode:

INPUT: search_space, num_wolves, max_iterations
OUTPUT: Best Solution

1. Initialise wolves within search_space randomly
2. Set alpha, beta, gamma wolves & their fitness to infinity
3. FOR iteration = 1 to max_iterations:
4. Update parameter 'a' = 2 - (2*iteration/max_iteration)
5. FOR each wolf:
6. Calculate fitness
7. Update alpha, beta, gamma wolves based on fitness
8. FOR EACH wolf AND dimension:
9. Update position using alpha, beta, gamma influence
10. Clip position within search_space bounds
11. Return alpha wolf as best solution

OUTPUT:

Optimal solution found: [3.8606e-17 4.51739713e-17]

Optimal fitness: 3.5311462342618 57e-33

Optimal Solution: [3.860634 e-17 4.5173 9913 e-17]

CODE:

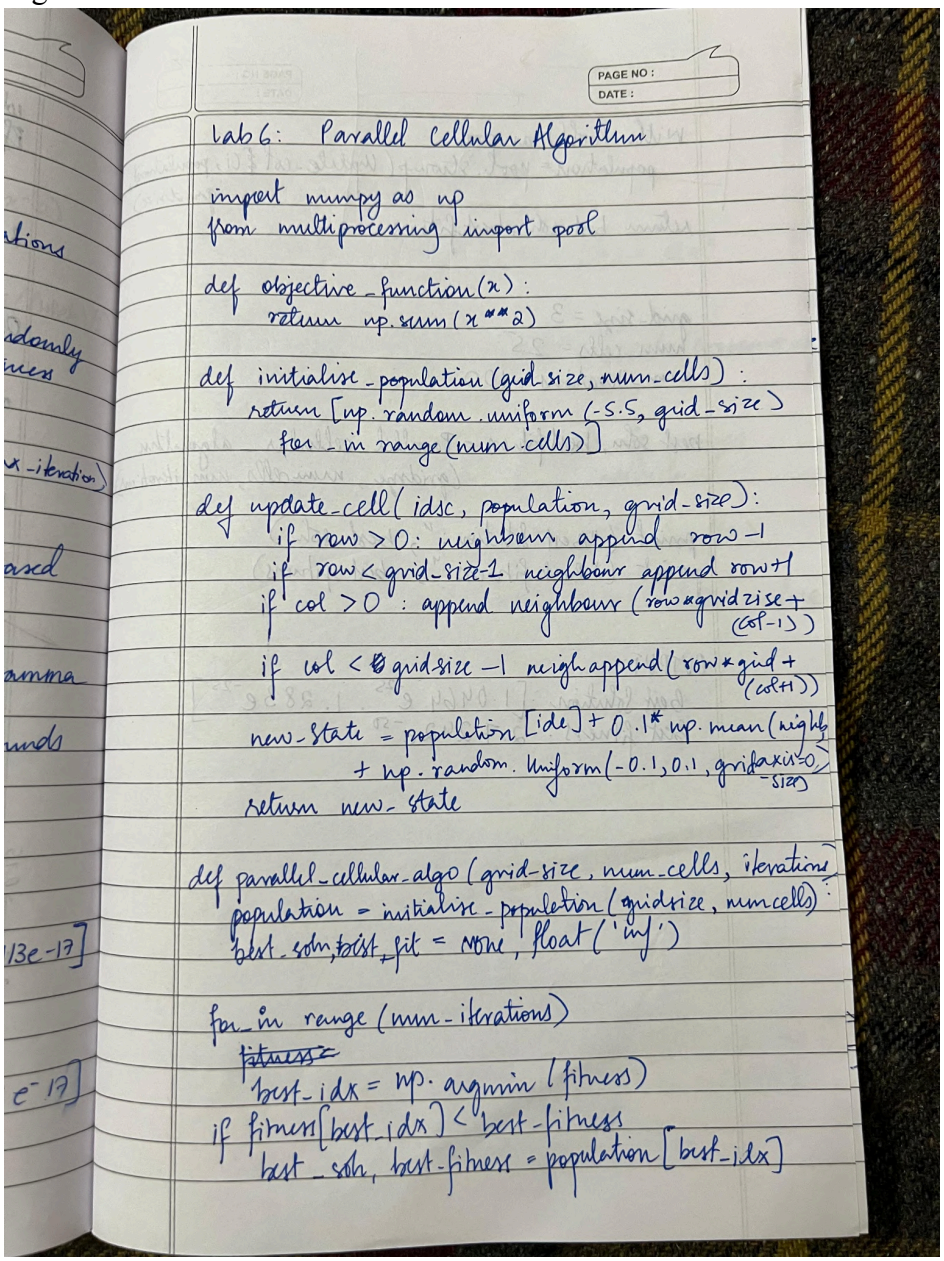
```
import numpy as np

# Objective function (e.g., Sphere function)
def objective_function(position):
    return sum(x**2 for x in position)

# Grey Wolf Optimizer
def grey_wolf_optimizer(obj_function, dim, pop_size, max_iter, bounds=(-10, 10)):
    a = 2  # Coefficient, decreases linearly from 2 to 0
    alpha_position = np.zeros(dim)
    alpha_score = float('inf')  # Best fitness (alpha)
    beta_position = np.zeros(dim)
    beta_score = float('inf')  # Second-best fitness (beta)
    delta_position = np.zeros(dim)
    delta_score = float('inf')  # Third-best fitness (delta)

    # Initialize the positions of the wolves
    wolves = np.random.uniform(bounds[0], bounds[1], (pop_size, dim))

    for iteration in range(max_iter):
        for i, wolf in enumerate(wolves):
            fitness = obj_function(wolf)

            # Update alpha, beta, and delta
            if fitness < alpha_score:
                delta_position = beta_position.copy()
                delta_score = beta_score
                beta_position = alpha_position.copy()
                beta_score = alpha_score
                alpha_position = wolf.copy()
                alpha_score = fitness
            elif fitness < beta_score:
                delta_position = beta_position.copy()
                delta_score = beta_score
                beta_position = wolf.copy()
                beta_score = fitness
            elif fitness < delta_score:
                delta_position = wolf.copy()
                delta_score = fitness

        # Update positions
        for i, wolf in enumerate(wolves):
            r1, r2 = np.random.rand(dim), np.random.rand(dim)
            A1 = 2 * a * r1 - a
            C1 = 2 * r2
            D_alpha = abs(C1 * alpha_position - wolf)
            X1 = alpha_position - A1 * D_alpha
```

```
            r1, r2 = np.random.rand(dim), np.random.rand(dim)
            A2 = 2 * a * r1 - a
            C2 = 2 * r2
            D_beta = abs(C2 * beta_position - wolf)
            X2 = beta_position - A2 * D_beta

            r1, r2 = np.random.rand(dim), np.random.rand(dim)
            A3 = 2 * a * r1 - a
            C3 = 2 * r2
            D_delta = abs(C3 * delta_position - wolf)
            X3 = delta_position - A3 * D_delta

            wolves[i] = (X1 + X2 + X3) / 3

        # Linearly decrease a
        a -= 2 / max_iter

        print(f"Iteration {iteration+1}/{max_iter}, Alpha Score: {alpha_score}")

    return alpha_position, alpha_score

# Example usage
best_position, best_score = grey_wolf_optimizer(objective_function, dim=2, pop_size=30, max_iter=100)
print("Best Position:", best_position)
print("Best Score:", best_score)
```

Program 6

Parallel Cellular Algorithms and Programs:

Algorithm:



```
Lab 6: Parallel Cellular Algorithm

import numpy as np
from multiprocessing import pool

def objective_function(x):
    return np.sum(x**2)

def initialise_population(grid_size, num_cells):
    return [np.random.uniform(-5.5, grid_size)
     for _ in range(num_cells)]

def update_cell(idx, population, grid_size):
    if row > 0: neighbours append row-1
    if row < grid_size-1 neighbour append row+1
    if col > 0: append neighbour (row*gridsize+(col-1))
    if col < gridsize-1 neigh append(row*grid+(col+1))
    new_state = population[idx] + 0.1* np.mean(neigh
        + np.random.uniform(-0.1, 0.1, gridaxis=0, size)
    return new_state

def parallel_cellular_algo(grid_size, num_cells, iterations)
    population = initialise_population(gridsize, numcells)
    best_soln, best_fit = none, float('inf')

    for in range(num_iterations)
        fitness
        best_idx = np.argmin(fitness)
    if fitness[best_idx] < best_fitness
        best_soln, best_fitness = population[best_idx]
```

PAGE NO :
DATE :

```
with Pool() as pool:
    population = pool.starmap(update_cell, [(i, population, grid_size)
return best_soln, best_fitness


grid_size = 3
num_cells = 25
num_iterations = 100

best_soln, best_fitness = Parallel_cellular_algorithm
                          (grid_size, num_cells, num_iterations)

print("Best Solution:", best_soln)
print("Best fitness:", best_fitness)
```

OUTPUT:

Best Solution : $[1.0464\ e^{-25}\ .\ 1.283e^{-25}]$

Best fitness : $2.7242e^{-50}$

CODE:

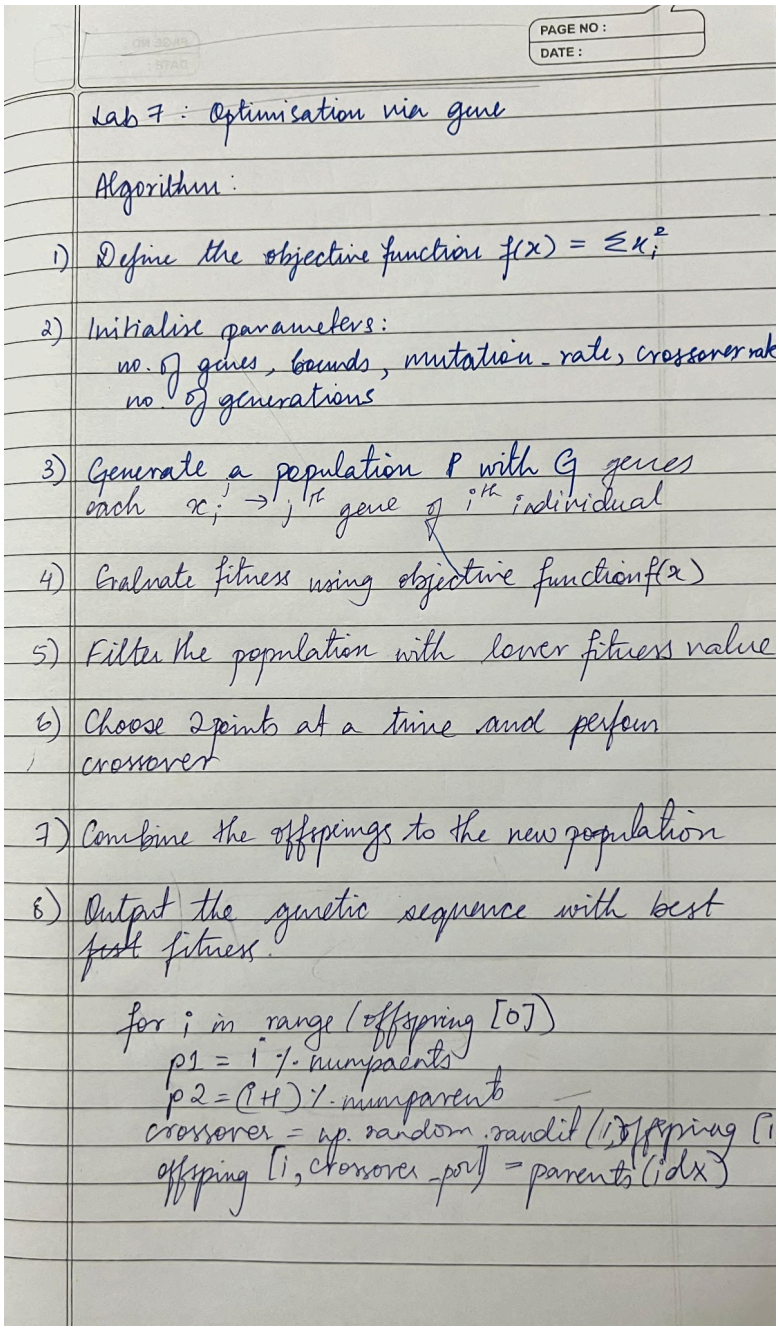
```
import numpy as np

# Define the objective function
def objective_function(x):
    return x**2 - 4*x + 4

# Initialize the grid
def initialize_grid(grid_size, search_range):
    return np.random.uniform(search_range[0], search_range[1], (grid_size, grid_size))

# Compute fitness for the grid
def evaluate_fitness(grid, objective_function):
    return objective_function(grid)

# Update the grid based on neighborhood average
def update_grid(grid):
    new_grid = np.copy(grid)
    for i in range(grid.shape[0]):
        for j in range(grid.shape[1]):
            # Get neighbors' values
            neighbors = []
            for di in [-1, 0, 1]:
                for dj in [-1, 0, 1]:
                    ni, nj = i + di, j + dj
                    if 0 <= ni < grid.shape[0] and 0 <= nj < grid.shape[1]:
                        neighbors.append(grid[ni, nj])
            # Update state to the average of neighbors
            new_grid[i, j] = np.mean(neighbors)
    return new_grid

# Main function to run the algorithm
def parallel_cellular_algorithm(grid_size, search_range, iterations):
    grid = initialize_grid(grid_size, search_range)  # Step 2: Initialize grid
    for _ in range(iterations):
        fitness = evaluate_fitness(grid, objective_function)  # Step 3: Evaluate fitness
        grid = update_grid(grid)  # Step 4: Update states
    # Find the best solution
    best_value = grid[np.unravel_index(np.argmin(fitness), fitness.shape)]
    return best_value, objective_function(best_value)

# Parameters
grid_size = 10  # 10x10 grid
search_range = (-10, 10)  # Search range for cell values
iterations = 100  # Number of iterations

# Run the algorithm
best_value, best_fitness = parallel_cellular_algorithm(grid_size, search_range, iterations)
```

```
# Output the results
print(f"Best Value: {best_value}")
print(f"Best Fitness: {best_fitness}")
```

Program 7

Optimization via Gene Expression Algorithms:

Algorithm:



Lab 7 : Optimisation via gene

Algorithm :

1) Define the objective function $f(x) = \Sigma x_i^2$

2) Initialise parameters :
no. of genes, bounds, mutation_rate, crossover rate
no. of generations

3) Generate a population P with G genes each $x_{ij} \rightarrow j^{th}$ gene of i^{th} individual

4) Evalnate fitness using objective function f(x)

5) Filter the population with lower fitness value

6) Choose 2 pints at a time and perform crossover

7) Combine the offsprings to the new population

8) Output the genetic sequence with best ~~post~~ fitness.

```
for i in range (offspring [0])
    p1 = i % numparents
    p2 = (i+1) % numparents
    crossover = np. random. randint (1, offspring [1
    offspring [i, crossover_pt] = parents (idx)
```

CODE:

```
import numpy as np
import random

# Define the Rastrigin function (objective function)
def rastrigin_function(x):
    A = 10
    n = len(x)
    return A * n + sum([(xi ** 2 - A * np.cos(2 * np.pi * xi)) for xi in x])

# Initialize population
def initialize_population(pop_size, gene_length, search_range):
    return [np.random.uniform(search_range[0], search_range[1], gene_length) for _ in range(pop_size)]

# Evaluate fitness
def evaluate_fitness(population):
    return [rastrigin_function(ind) for ind in population]

# Selection (tournament selection)
def selection(population, fitness):
    selected = []
    for _ in range(len(population)):
        i, j = random.sample(range(len(population)), 2)
        selected.append(population[i] if fitness[i] < fitness[j] else population[j])
    return selected

# Crossover (uniform crossover)
def crossover(parent1, parent2):
    child = []
    for p1, p2 in zip(parent1, parent2):
        child.append(p1 if random.random() < 0.5 else p2)
    return np.array(child)

# Mutation (random mutation)
def mutate(individual, mutation_rate, search_range):
    for i in range(len(individual)):
        if random.random() < mutation_rate:
            individual[i] = random.uniform(search_range[0], search_range[1])
    return individual

# Gene Expression Algorithm
def gene_expression_algorithm(pop_size, gene_length, generations, mutation_rate, search_range):
    # Step 1: Initialize population
    population = initialize_population(pop_size, gene_length, search_range)

    for generation in range(generations):
        # Step 2: Evaluate fitness
        fitness = evaluate_fitness(population)
```

```
        # Step 3: Selection
        selected_population = selection(population, fitness)

        # Step 4: Crossover and Mutation
        next_population = []
        for i in range(0, len(selected_population), 2):
            if i + 1 < len(selected_population):
                # Crossover
                child1 = crossover(selected_population[i], selected_population[i + 1])
                child2 = crossover(selected_population[i + 1], selected_population[i])
            else:
                child1 = selected_population[i]
                child2 = selected_population[i]
            # Mutation
            next_population.append(mutate(child1, mutation_rate, search_range))
            next_population.append(mutate(child2, mutation_rate, search_range))

        population = next_population[:pop_size]  # Maintain population size

    # Final fitness evaluation
    fitness = evaluate_fitness(population)
    best_individual = population[np.argmin(fitness)]
    return best_individual, rastrigin_function(best_individual)

# Parameters
pop_size = 50
gene_length = 10
generations = 100
mutation_rate = 0.1
search_range = (-5.12, 5.12)  # Search range for Rastrigin function

# Run the algorithm
best_solution, best_fitness = gene_expression_algorithm(pop_size, gene_length, generations, mutation_rate,
search_range)

print(f"Best Solution: {best_solution}")
print(f"Best Fitness: {best_fitness}")
```