VISVESVARAYA TECHNOLOGICAL UNIVERSITY

"JnanaSangama", Belgaum -590014, Karnataka.

LAB RECORD

Bio Inspired Systems (23CS5BSBIS)

Submitted by

Setu Mishra (1BM22CS250)

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 **Setu Mishra(1BM22CS250),** 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. Jyothi S Nayak Professor & HOD Department of CSE, BMSCE

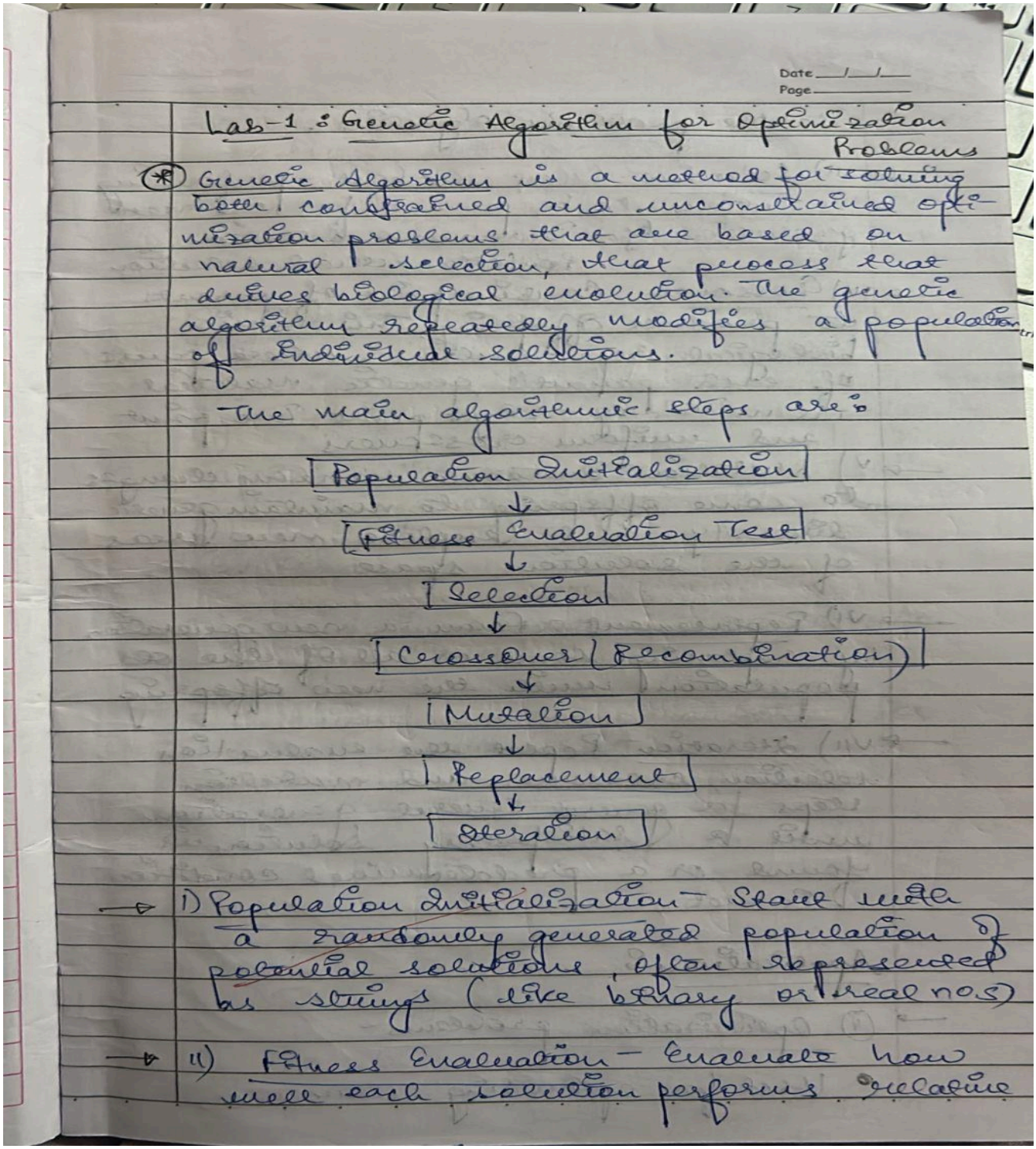
Index

Github Link:https://github.com/setu-mishra/BIS_LAB

Program 1

Genetic Algorithm for Optimization Problems

Algorithm:



Date __/__/__
Page _____

Lab-1 : Genetic Algorithm for Optimization Problems

(*) Genetic Algorithm is a method for solving both constrained and unconstrained optimization problems that are based on natural selection, that process that drives biological evolution. The genetic algorithm repeatedly modifies a population of individual solutions.

The main algorithmic steps are:

[Population Initialization]
↓
[Fitness Evaluation Test]
↓
[Selection]
↓
[Crossover (Recombination)]
↓
[Mutation]
↓
[Replacement]
↓
[Iteration]

→ i) Population Initialization - Start with a randomly generated population of potential solutions, often represented as strings (like binary or real nos.)

→ ii) Fitness Evaluation - Evaluate how well each solution performs, relative

Date __/__/__
Page _____

to the problem's objective.

→ III) Selection - Select the best-performing solutions to be parents for the next generation - Eg - Tournament selection, Roulette wheel selection

→ IV) Crossover - Combine pairs of parents to produce offspring. This mimics biological reproduction, where parts of each parent's genetic material are mixed. Eg:- Single point, two point and uniform crossovers

→ V) Mutation - Introduce random changes to some offspring to maintain genetic diversity and explore new areas of the solution space.

→ VI) Replacement - Form a new generation by replacing some or all of the old population with the new offspring.

→ VII) Iteration - Repeat the evaluation, selection, crossover, and mutation steps for several generations until a satisfactory solution is found or a predetermined condition is met.

Applications:

→ (1) Optimization problems -

- Used for optimizing structures,

components and systems in field like aerospace and civil engineering.

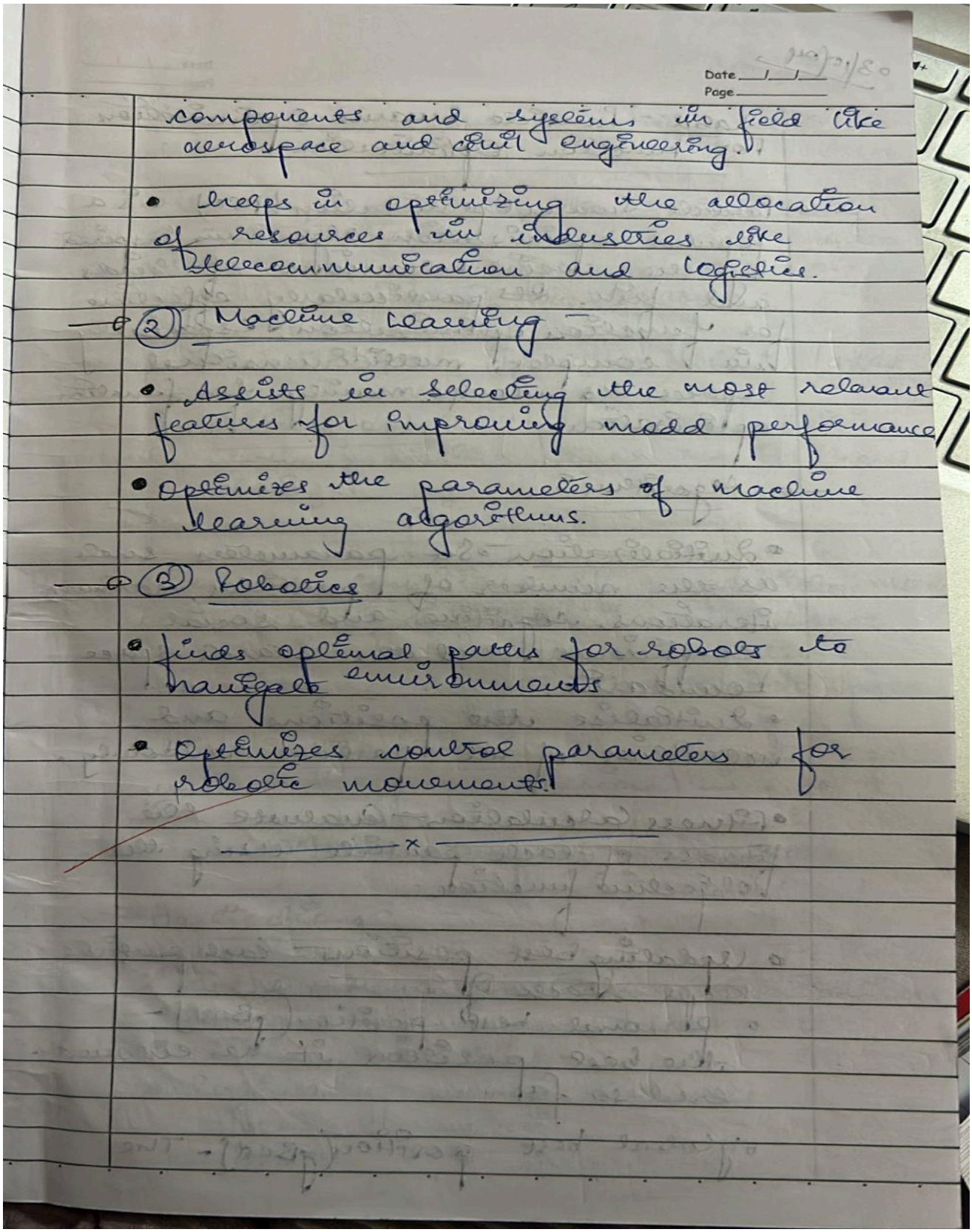
- Helps in optimizing the allocation of resources in industries like telecommunication and logistics.

② Machine learning –

- Assists in selecting the most relevant features for improving model performance
- Optimizes the parameters of machine learning algorithms.

③ Robotics

- finds optimal paths for robots to navigate environments
- Optimizes control parameters for robotic movements.



Code:

```
import random

# Set a random seed for reproducibility
random.seed(42)

def fitness(chromosome):
    x = int(''.join(map(str, chromosome)), 2)
    return x ** 2

def binary_string_to_chromosome(binary_string):
    return [int(bit) for bit in binary_string]

def generate_population_from_input():
    population = []
    for _ in range(population_size):
        while True:
            binary_string = input("Enter a binary string of size 5 (e.g., '11001'): ")
            if len(binary_string) == 5 and all(bit in '01' for bit in binary_string):
                population.append(binary_string_to_chromosome(binary_string))
                break
            else:
                print("Invalid input. Please enter a binary string of size 5.")
    return population

def select_pair(population, fitnesses):
    total_fitness = sum(fitnesses)
    selection_probs = [f / total_fitness for f in fitnesses]
    parent1 = population[random.choices(range(len(population)), selection_probs)[0]]
    parent2 = population[random.choices(range(len(population)), selection_probs)[0]]
    return parent1, parent2

def crossover(parent1, parent2):
    point = random.randint(1, len(parent1) - 1)
    offspring1 = parent1[:point] + parent2[point:]
    offspring2 = parent2[:point] + parent1[point:]
    return offspring1, offspring2

def mutate(chromosome, mutation_rate):
    return [gene if random.random() > mutation_rate else 1 - gene for gene in chromosome]

# Parameters
```

```
population_size = 4
generations = 20
mutation_rate = 0.01

# Initialize population from user input
population = generate_population_from_input()

for generation in range(generations):
    fitnesses = [fitness(chromosome) for chromosome in population]

    new_population = []

    # Create new population
    while len(new_population) < population_size:
        parent1, parent2 = select_pair(population, fitnesses)
        offspring1, offspring2 = crossover(parent1, parent2)
        new_population.append(mutate(offspring1, mutation_rate))
        new_population.append(mutate(offspring2, mutation_rate))

    # Ensure the new population has the right size
    population = new_population[:population_size]

# Get the maximum fitness
fitnesses = [fitness(chromosome) for chromosome in population]
max_fitness = max(fitnesses)

print(f"Maximum Possible Fitness: {max_fitness}")
```

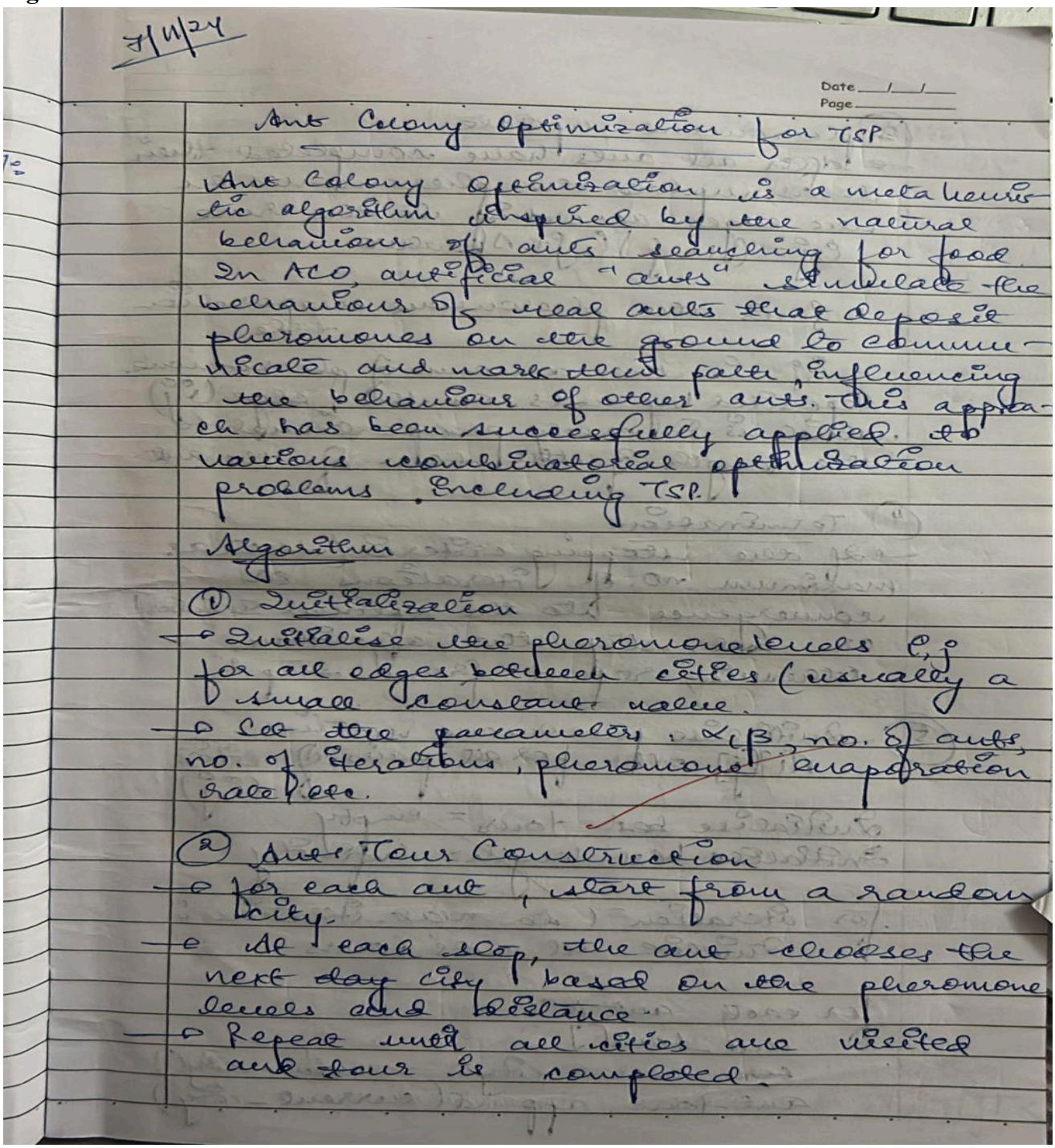
Output:

```
Enter a binary string of size 5 (e.g., '11001'): 11011
Enter a binary string of size 5 (e.g., '11001'): 01011
Enter a binary string of size 5 (e.g., '11001'): 11100
Enter a binary string of size 5 (e.g., '11001'): 01101
Maximum Possible Fitness: 841
```

Program 2

Ant Colony Optimization

Algorithm:



7/11/24

Date / /
Page

Ant Colony Optimization for TSP

Ant Colony Optimization is a metaheuristic algorithm inspired by the natural behaviours of ants searching for food. In ACO, artificial "ants" simulate the behaviours of real ants that deposit pheromones on the ground to communicate and mark their path, influencing the behaviour of other ants. This approach has been successfully applied to various combinatorial optimization problems, including TSP.

Algorithm

① Initialization
- Initialise the pheromone levels τ_{ij} for all edges between cities (usually a small constant value.
- Set the parameters: α, β, no. of ants, no. of iterations, pheromone evaporation rate, etc.

② Ant Tour Construction
- for each ant, start from a random city.
- At each step, the ant chooses the next day city based on the pheromone levels and distance.
- Repeat until all cities are visited and tour is completed.

Date __/__/__
Page _____

(3) Pheromone update
- After all ants have completed their tours, update the pheromones on the edges according to the rule
$\rho_{ij} = (1-\rho)\rho_{ij}, \rho \in$ [0,1] where
- ρ is the pheromone evaporation rate
- $\Delta\rho_{ij}$ is the amount of pheromone deposited by ants on the edge (i,j) which is inversely proportional to the length of the tour of the ant.

(4) Termination
- If the stopping criteria (such as maximum no. of iterations or convergence to the optimal solution) are met, stop the algorithm. Otherwise, repeat from step 2.

(5) Initialise pheromone matrix
r[i][j] = rd for all pairs of cities (i, j)
Initialise best tour = empty
Initialise best length = infinity
for iteration = 1 to max_iterations:
  Initialise all ants' tours
  for each ant k:
    ant_tour[]
    current_city = random_city()
    ant_tour.append(current_city)

Date __/__/__
Page _____

while tour is not complete (all cities visited):
    next_city = choose_next_city(current_city, ant_tour)
    ant_tour.append(next_city)
    current_city = next_city
tour_length = calculate_tour_length(ant_tour)
if tour_length < best_length:
    best_tour = ant_tour
    best_length = tour_length
Apply pheromone vaporization (v[i][j] = (1-ρ)*v[i][j])
for each ant k:
    for each edge (i, j) in ant_tour:
        update_pheromone(i, j, ant_tour, tour_length)
Return best_tour, best_length

// function to choose the next city using pheromone and distance
function choose_next_city(current_city, visited_cities):
    probabilities = []
    total_pheromone = 0
    for each city j not in visited cities:
        pheromone = v[current_city][j] ^ α

```
visibility = (1 / distance[current_city][j]) ** beta
total_pheromone += pheromone ** alpha * visibility
for each city j not in visited_cities:
    pheromone = r[current_city][j] ** alpha
    visibility = (1 / distance[current_city][j]) ** beta
    probability = (pheromone * visibility) / total_pheromone
    probabilities.append(probability)
next_city = select_city_based_on_probability(probabilities)
return next_city
```

```
// func to update pheromone on an edge (i, j)
func update_pheromone(i, j, ant_tour, tour_length):
    Δr = Q / tour_length   // Q -> constant
    r[i][j] += Δr
```

Applications:

Implemented 21/11/24

1. Reservoir optimization
2. Water distribution systems
3. Urban drainage system design
4. Finding best possible route
5. Job (resource) allocation

Code:

```
import random
import numpy as np
import operator

FUNCTIONS = {'+': operator.add, '-': operator.sub, '*': operator.mul, '/': operator.truediv}
TERMINALS = ['x', 1, 2, 3, 4]  # x and constants

def random_gene(length=10):
    return [random.choice(list(FUNCTIONS.keys()) + TERMINALS) for _ in range(length)]

def decode_chromosome(chromosome, x):
    stack = []
    for gene in chromosome:
        if gene in FUNCTIONS:  # If it's a function, pop arguments and apply
            if len(stack) < 2:  # Avoid errors if stack has fewer than 2 elements
                stack.append(0)
                continue
            b = stack.pop()
            a = stack.pop()
            try:
                result = FUNCTIONS[gene](a, b)
            except ZeroDivisionError:
                result = 1  # Avoid division by zero
            stack.append(result)
        elif gene == 'x':
            stack.append(x)
        else:
            stack.append(gene)
    return stack[0] if stack else 0  # Return top of stack as output

def fitness_function(chromosome, target_function, x_values):
    predictions = [decode_chromosome(chromosome, x) for x in x_values]
    targets = [target_function(x) for x in x_values]
    mse = np.mean([(p - t) ** 2 for p, t in zip(predictions, targets)])
    return mse


def selection(population, fitnesses):
```

```
    total_fitness = sum(1 / (f + 1e-6) for f in fitnesses)  # Avoid division by zero
    probabilities = [(1 / (f + 1e-6)) / total_fitness for f in fitnesses]
    return population[np.random.choice(len(population), p=probabilities)]

def mutate(chromosome, mutation_rate=0.1):
    new_chromosome = chromosome[:]
    for i in range(len(new_chromosome)):
        if random.random() < mutation_rate:
            new_chromosome[i] = random.choice(list(FUNCTIONS.keys()) + TERMINALS)
    return new_chromosome

def crossover(parent1, parent2):
    point = random.randint(1, len(parent1) - 1)
    child1 = parent1[:point] + parent2[point:]
    child2 = parent2[:point] + parent1[point:]
    return child1, child2

def ant_colony_optimization(cost_matrix, n_ants=10, n_iterations=100, evaporation_rate=0.5,
alpha=1, beta=2):
    n_nodes = len(cost_matrix)
    pheromones = np.ones((n_nodes, n_nodes))  # Initialize pheromones

    def calculate_probability(i, j, visited):
        if j in visited:
            return 0
        return (pheromones[i][j] ** alpha) * ((1 / cost_matrix[i][j]) ** beta)

    def construct_solution():
        path = [random.randint(0, n_nodes - 1)]
        while len(path) < n_nodes:
            i = path[-1]
            probabilities = [calculate_probability(i, j, path) for j in range(n_nodes)]
            total = sum(probabilities)
            probabilities = [p / total if total > 0 else 0 for p in probabilities]
            next_node = np.random.choice(range(n_nodes), p=probabilities)
            path.append(next_node)
        path.append(path[0])  # Return to start
        return path

    def path_cost(path):
        return sum(cost_matrix[path[i]][path[i + 1]] for i in range(len(path) - 1))
```

```
    best_path = None
    best_cost = float('inf')

    for iteration in range(n_iterations):
        solutions = [construct_solution() for _ in range(n_ants)]
        costs = [path_cost(solution) for solution in solutions]
        for i, cost in enumerate(costs):
            if cost < best_cost:
                best_cost = cost
                best_path = solutions[i]

        pheromones *= (1 - evaporation_rate)  # Evaporation
        for i, solution in enumerate(solutions):
            for j in range(len(solution) - 1):
                pheromones[solution[j]][solution[j + 1]] += 1 / costs[i]

        print(f"Iteration {iteration + 1}: Best Cost = {best_cost}")

    print("Best Path:", best_path)
    print("Best Cost:", best_cost)

cost_matrix = [
    [0, 2, 2, 5, 7],
    [2, 0, 4, 8, 2],
    [2, 4, 0, 1, 3],
    [5, 8, 1, 0, 2],
    [7, 2, 3, 2, 0]
]
ant_colony_optimization(cost_matrix, n_ants=5, n_iterations=20)
```

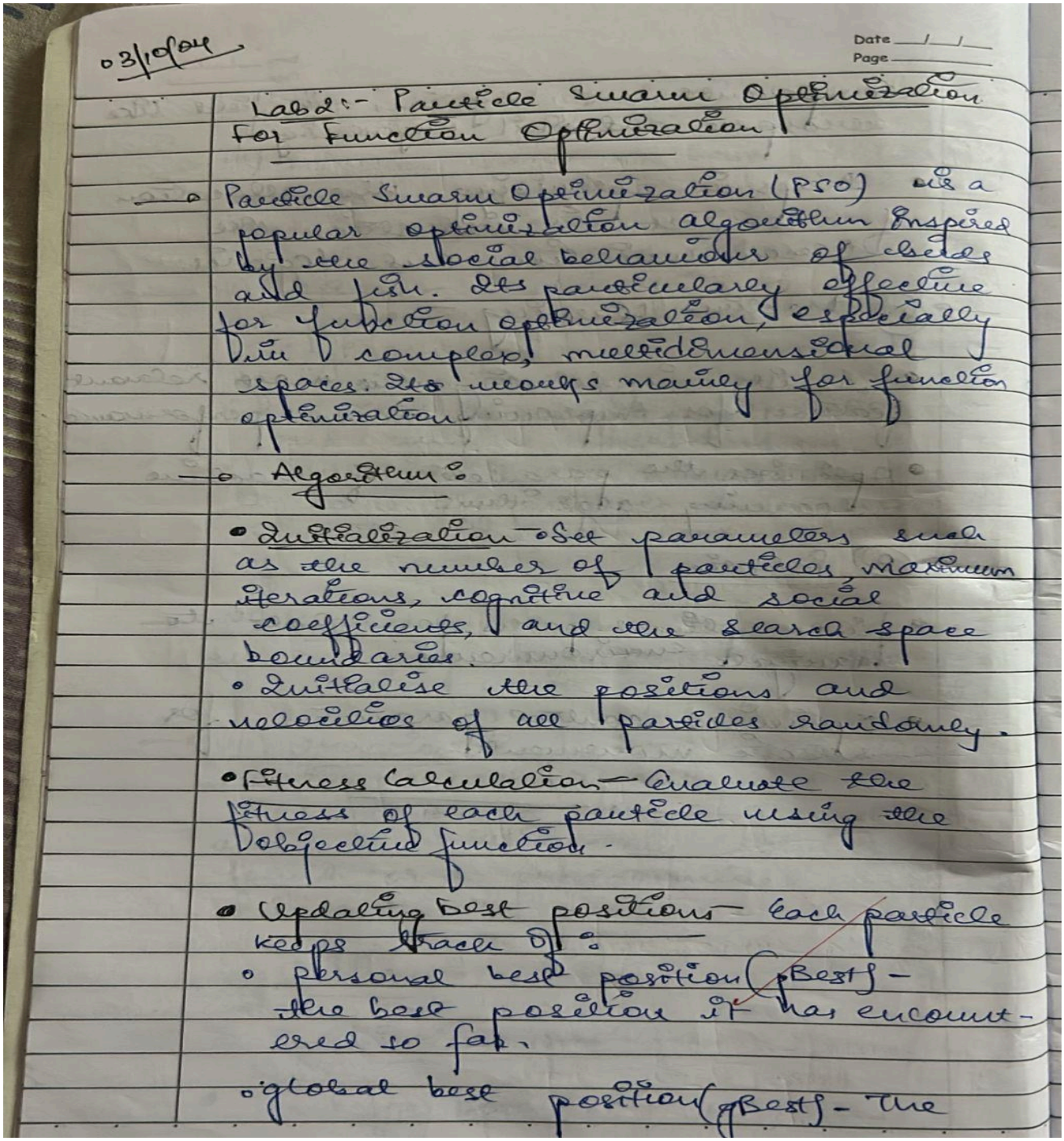
Output:

```
Iteration 15: Best Cost = 9
Iteration 16: Best Cost = 9
Iteration 17: Best Cost = 9
Iteration 18: Best Cost = 9
Iteration 19: Best Cost = 9
Iteration 20: Best Cost = 9
Best Path: [1, 0, 2, 3, 4, 1]
Best Cost: 9
```

Program 3

Particle Swarm Optimization

Algorithm:



Lab 2:- Particle Swarm Optimization for Function Optimization

- Particle Swarm Optimization (PSO) is a popular optimization algorithm inspired by the social behaviour of birds and fish. Its particularly effective for function optimization, especially in complex, multidimensional spaces. It works mainly for function optimization.

- Algorithm:

- Initialization - Set parameters such as the number of particles, maximum iterations, cognitive and social coefficients, and the search space boundaries.
- Initialise the positions and velocities of all particles randomly.
- Fitness Calculation - Evaluate the fitness of each particle using the objective function.
- Updating best positions - Each particle keeps track of:
  - personal best position (pBest) - the best position it has encountered so far.
  - global best position (gBest) - The

Date / /
Page

best position found by any particle in the swarm.

- Velocity update - Update the velocity of each particle based on its own best-known position and the best known position of the swarm:

$$V_i^{\circ} = w \cdot V_i^{\circ} + c_1 \cdot r_1 \cdot (pBest_i^{\circ} - x_i^{\circ}) + c_2 \cdot r_2 \cdot (gBest - x_i^{\circ})$$

- Position update - Update the position of each particle based on its new velocity

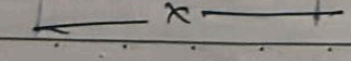
$$x_i^{\circ} = x_i^{\circ} + V_i^{\circ}$$

- Boundary conditions - If a particle moves out of the defined search space boundaries it can be reset to the boundary or reflected back into space

- Termination criteria - The process repeats until a stopping criterion is met like maximum no. of iterations, etc.

Applications

- Mathematical function optimization
- Machine learning
- Signal Processing
- Engineering Design Problems



24/10/24

Date __/__/__
Page

PSO - pseudocode

```
#Initialise parameters
num_particles = N
max_iterations = MAX_ITER
inertia_weight = w
c1 = cognitive_parameter   # personal best influence
c2 = social_parameter      # global best influence

#Initialising the swarm (particles)
for i in range(num_particles):
    particles[i].position = random_position_in_search_space()
    particles[i].velocity = random_velocity()
    particles[i].pBest = particles[i].position
                # personal best initialised to the starting position
    particles[i].pBest_value =
                fitness_function(particles[i].position)

# Initialise global best (gBest)
gBest = particles[0].pBest
gBest_value = particles[0].pBest_value

# main optimization loop
for iteration in range(max_iterations):
    for i in range(num_particles):
        # Evaluating fitness test of the current position
        fitness_value = fitness_function
                (particles[i].position)
```

Date __/__/__
Page _____

```
#update personal best (pBest)
if fitness_value < particles[i].pBest_value:
    particles[i].pBest = particles[i].position
    particles[i].pBest_value = fitness_value

#update global best (gBest)
if fitness_value < gBest_value:
    gBest = particles[i].position
    gBest_value = fitness_value

#update velocity and position of each particle
for i in range(num_particles):
    for d in range(num_dimensions):
        r1 = random(0,1)
        r2 = random(0,1)
#update velocity
particles[i].velocity[d] = (inertia_weight * particles[i].velocity[d]
    + c1*r1*(particles[i].pBest[d] - particles[i].position[d]) +
    c2*r2*(gBest[d] - particles[i].position[d]))

#update position
particles[i].position[d] = particles[i].position[d] + particles[i].velocity[d]
```

Date __/__/__
Page _____

```
# Ensure position is within bounds
if particles[i].position[d] < lower_bound[d]:
    particles[i].position[d] = lower_bound[d]
if particles[i].position[d] > upper_bound[d]:
    particles[i].position[d] = upper_bound[d]
# Return the best soln found
return gBest, gBest_value
```

Code:

```
import random
import numpy as np
from matplotlib import pyplot as plt
from matplotlib import animation

def fitness_function(x1, x2):
    f1 = x1 + 2 * -x2 + 3
    f2 = 2 * x1 + x2 - 8
    z = f1**2 + f2**2
    return z

def update_velocity(particle, velocity, pbest, gbest, w_min=0.5, max=1.0, c=0.1):
    new_velocity = np.zeros_like(particle)
    r1 = random.uniform(0, max)
    r2 = random.uniform(0, max)
    w = random.uniform(w_min, max)

    for i in range(len(particle)):
        new_velocity[i] = (w * velocity[i] +
                           c * r1 * (pbest[i] - particle[i]) +
                           c * r2 * (gbest[i] - particle[i]))
    return new_velocity

def update_position(particle, velocity):
    new_particle = particle + velocity
    return new_particle

def pso_2d(population, dimension, position_min, position_max, generation, fitness_criterion):
    # Initialization
    particles = np.array([[random.uniform(position_min, position_max) for _ in range(dimension)] for
_ in range(population)])
    pbest_position = particles.copy()
    pbest_fitness = np.array([fitness_function(p[0], p[1]) for p in particles])

    gbest_index = np.argmin(pbest_fitness)
    gbest_position = pbest_position[gbest_index]

    velocity = np.zeros((population, dimension))
```

```
images = []  # For animation

for t in range(generation):
    if np.average(pbest_fitness) <= fitness_criterion:
        break

    for n in range(population):
        velocity[n] = update_velocity(particles[n], velocity[n], pbest_position[n], gbest_position)
        particles[n] = update_position(particles[n], velocity[n])

    pbest_fitness = np.array([fitness_function(p[0], p[1]) for p in particles])
    for n in range(population):
        if pbest_fitness[n] < fitness_function(pbest_position[n][0], pbest_position[n][1]):
            pbest_position[n] = particles[n]

    gbest_index = np.argmin(pbest_fitness)
    gbest_position = pbest_position[gbest_index]

    # Plotting the current positions of the particles
    fig = plt.figure(figsize=(10, 10))
    ax = fig.add_subplot(111, projection='3d')
    ax.set_xlabel('x')
    ax.set_ylabel('y')
    ax.set_zlabel('z')

    x = np.linspace(position_min, position_max, 80)
    y = np.linspace(position_min, position_max, 80)
    X, Y = np.meshgrid(x, y)
    Z = fitness_function(X, Y)
    ax.plot_wireframe(X, Y, Z, color='r', linewidth=0.2)

    ax.scatter3D(
        particles[:, 0],
        particles[:, 1],
        [fitness_function(p[0], p[1]) for p in particles],
        c='b'
    )

    # Capture the frame for animation
    plt.title(f'Generation: {t + 1}')
```

```
        plt.tight_layout()
        plt.savefig(f'frame_{t}.png')
        plt.close(fig)

    # Create animation
    frames = [plt.imread(f'frame_{i}.png') for i in range(t)]
    fig, ax = plt.subplots(figsize=(10, 10))
    ax.axis('off')
    image = ax.imshow(frames[0])

    def update(frame):
        image.set_array(frames[frame])
        return image,

    ani = animation.FuncAnimation(fig, update, frames=len(frames), interval=100)
    ani.save('./pso_simple.gif', writer='pillow')

    # Print the results
    print('Global Best Position: ', gbest_position)
    print('Best Fitness Value: ', min(pbest_fitness))
    print('Average Particle Best Fitness Value: ', np.average(pbest_fitness))
    print('Number of Generations: ', t)

# Run the PSO algorithm
pso_2d(population=30, dimension=2, position_min=-10, position_max=10, generation=100,
fitness_criterion=1e-3)
```

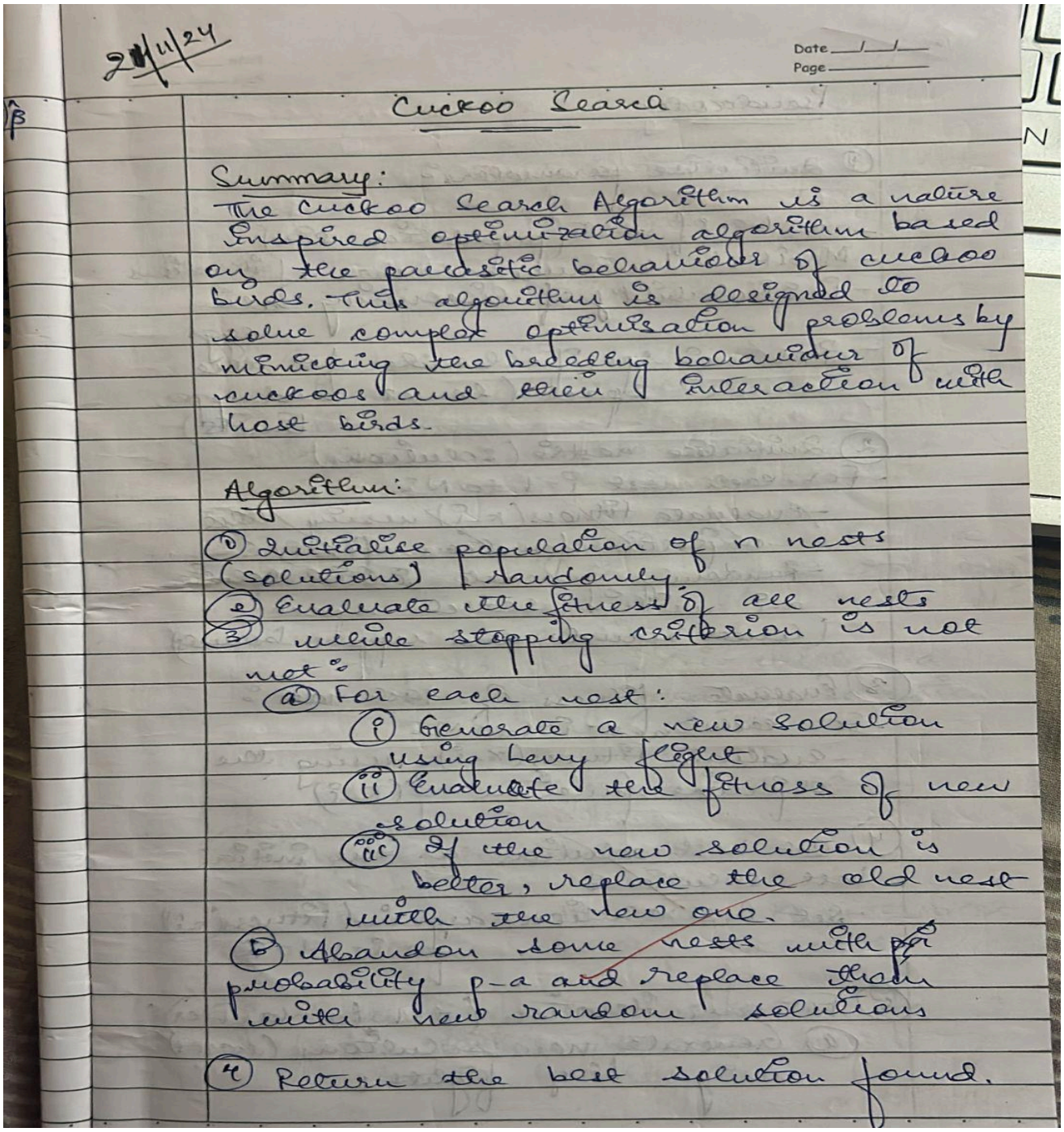
Output:

```
Global Best Position:  [2.59992843 2.79914636]
Best Fitness Value:  3.6691186243893878e-06
Average Particle Best Fitness Value:  0.0007223322365523365
Number of Generations:  45
```

Program 4

Cuckoo Search Algorithm

Algorithm:



Cuckoo Search

Summary:
The Cuckoo Search Algorithm is a nature inspired optimization algorithm based on the parasitic behaviour of cuckoo birds. This algorithm is designed to solve complex optimisation problems by mimicking the breeding behaviour of cuckoos and their interaction with host birds.

Algorithm:

1. Initialise population of n nests (solutions) randomly.
2. Evaluate the fitness of all nests
3. while stopping criterion is not met:
   a. For each nest:
      i. Generate a new solution using Levy flight
      ii. Evaluate the fitness of new solution
      iii. If the new solution is better, replace the old nest with the new one.
   b. Abandon some nests with probability p-a and replace them with new random solutions
4. Return the best solution found.

Date / /
Page

Pseudocode:

① Initialise parameters:
- N: No. of nests (population size)
- MaxIter: Maximum number of iterations
- Pa: Probability of discovering alien eggs
- Problem: specific bounds for the search space (lower and upper limits)

② Initialise nests: (solutions)
- For each nest i = 1 to N:
  - Evaluate fitness (x_i) using the objective function f(x_i)
  - Randomly generate initial solution x_i within search space (lower bounds and upper bounds)

③ Evaluate fitness of each nest:
- For each nest i = 1 to N:
  - evaluate fitness (x_i) using the objective function f(x_i)

④ Set best solution as the initial best solution:
- Set BestSolution = argmin(Fitness(x_i)) for i = 1 to N.

⑤ For each iteration (t = 1 to MaxIter):
  ⓐ Generate new solutions (nests) using Lévy flights:

Date / /
Page

$x_i' = x_i + \alpha \cdot \text{LevyFlight}()$

- Ensure x_i' stays within the search space bounds.
  If x_i' is out of bounds, adjust it to stay within the boundaries.
- Evaluate fitness (x_i')

(b) Evaluate and compare new solutions with the current nests.
- If fitness(x_i') < fitness(x_i):
  - Replace the old solution x_i with the new solution x_i'.

(c) Randomly replace nests with a probability Pa (discovery of alien eggs)
- For each nest $i = 1$ to N:
  - With probability Pa:
    - Replace nest x_i with a randomly generated solution x_{rand} within the search space.
    - Evaluate fitness(x_{rand})

(d) Update the best solution.
- If a better solution is found:
  Update BestSolution to the best solution among all nests

(e) Return the best solution found after MaxIter iterations.
- Return BestSolution.

21/11/24

Code:

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

# Objective function: Rastrigin Function
def rastrigin(x):
    A = 10
    return A * len(x) + sum(xi**2 - A * np.cos(2 * np.pi * xi) for xi in x)

# Lévy flight function for generating random steps
def levy_flight(beta=1.5, dim=2):
    sigma_u = np.power(np.math.gamma(1 + beta) * np.sin(np.pi * beta / 2) / np.math.gamma((1 +
beta) / 2) / np.power(2, (beta - 1) / 2), 1 / beta)
    sigma_v = 1
    u = np.random.normal(0, sigma_u, dim)
    v = np.random.normal(0, sigma_v, dim)
    return u / np.power(np.abs(v), 1 / beta)

# Cuckoo Search Algorithm
class CuckooSearch:
    def __init__(self, func, dim, population_size, max_generations, pa=0.25, beta=1.5,
lower_bound=-5, upper_bound=5):
        self.func = func              # Objective function
        self.dim = dim                # Dimension of the problem
        self.population_size = population_size  # Number of nests (solutions)
        self.max_generations = max_generations  # Maximum number of generations
        self.pa = pa                  # Probability of alien eggs (nest replacement)
        self.beta = beta              # Lévy flight exponent
        self.lower_bound = lower_bound  # Lower bound of the search space
        self.upper_bound = upper_bound  # Upper bound of the search space

        # Initialize population (nests)
        self.nests = np.random.uniform(self.lower_bound, self.upper_bound, (self.population_size,
self.dim))
        self.fitness = np.array([self.func(nest) for nest in self.nests])  # Fitness of each nest
        self.best_nest = self.nests[np.argmin(self.fitness)]  # Best solution found
        self.best_fitness = np.min(self.fitness)  # Best fitness value

    # Update nests using Lévy flights and objective function evaluations
    def generate_new_nests(self):
```

```
        new_nests = []
        for i in range(self.population_size):
            step = levy_flight(self.beta, self.dim)
            new_nest = self.nests[i] + step
            # Apply boundary check
            new_nest = np.clip(new_nest, self.lower_bound, self.upper_bound)
            new_nests.append(new_nest)
        return np.array(new_nests)

    # Main cuckoo search algorithm
    def search(self):
        history = []  # To record the best fitness values over generations

        for generation in range(self.max_generations):
            # Generate new nests based on Lévy flight
            new_nests = self.generate_new_nests()
            new_fitness = np.array([self.func(nest) for nest in new_nests])

            # Replace nests with new ones if they are better
            for i in range(self.population_size):
                if new_fitness[i] < self.fitness[i] or np.random.rand() < self.pa:
                    self.nests[i] = new_nests[i]
                    self.fitness[i] = new_fitness[i]

            # Find the best nest in the current population
            current_best_fitness = np.min(self.fitness)
            current_best_nest = self.nests[np.argmin(self.fitness)]

            # Update the global best solution
            if current_best_fitness < self.best_fitness:
                self.best_fitness = current_best_fitness
                self.best_nest = current_best_nest

            # Record the best fitness for the current generation
            history.append(self.best_fitness)
            print(f"Generation {generation+1}: Best fitness = {self.best_fitness}")

        return self.best_nest, self.best_fitness, history

# Analyze the Cuckoo Search Algorithm
def analyze_cuckoo_search():
```

```
    # Set up parameters for Cuckoo Search
    dim = 2
    population_size = 50
    max_generations = 100
    cuckoo_search = CuckooSearch(func=rastrigin, dim=dim, population_size=population_size,
max_generations=max_generations)

    # Run the Cuckoo Search algorithm
    best_nest, best_fitness, history = cuckoo_search.search()

    # Plot the convergence curve
    plt.plot(history)
    plt.title("Convergence Curve of Cuckoo Search Algorithm")
    plt.xlabel("Generation")
    plt.ylabel("Best Fitness")
    plt.show()

    print(f"Best solution found: {best_nest}")
    print(f"Best fitness: {best_fitness}")

# Run the analysis
analyze_cuckoo_search()
```

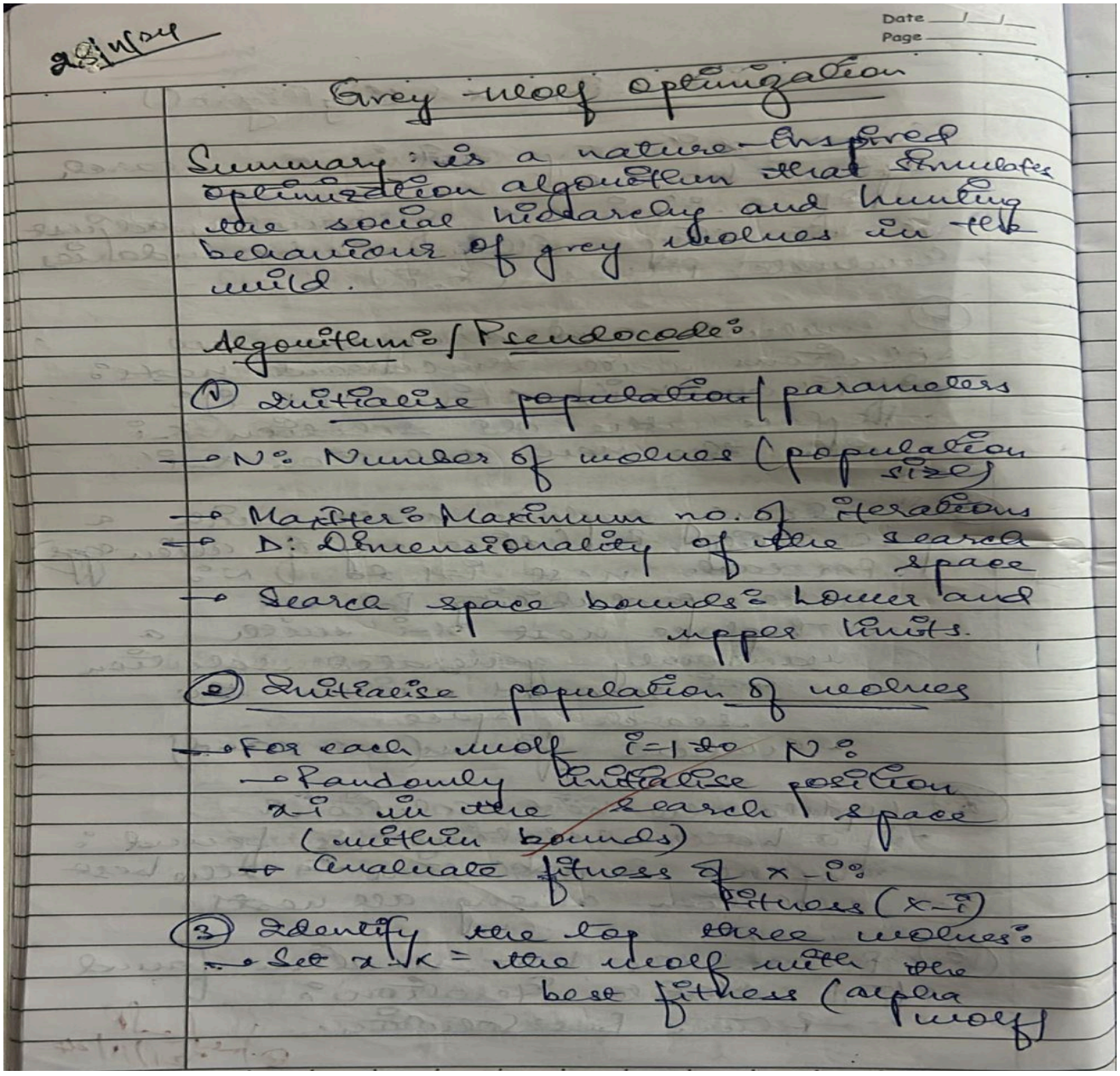
Output:

```
Best solution found: [1.30548027 2.02026344]
Best fitness: 0.16306139523513963
```

Program 5

Grey Wolf Optimizer

Algorithm:



Grey wolf optimization

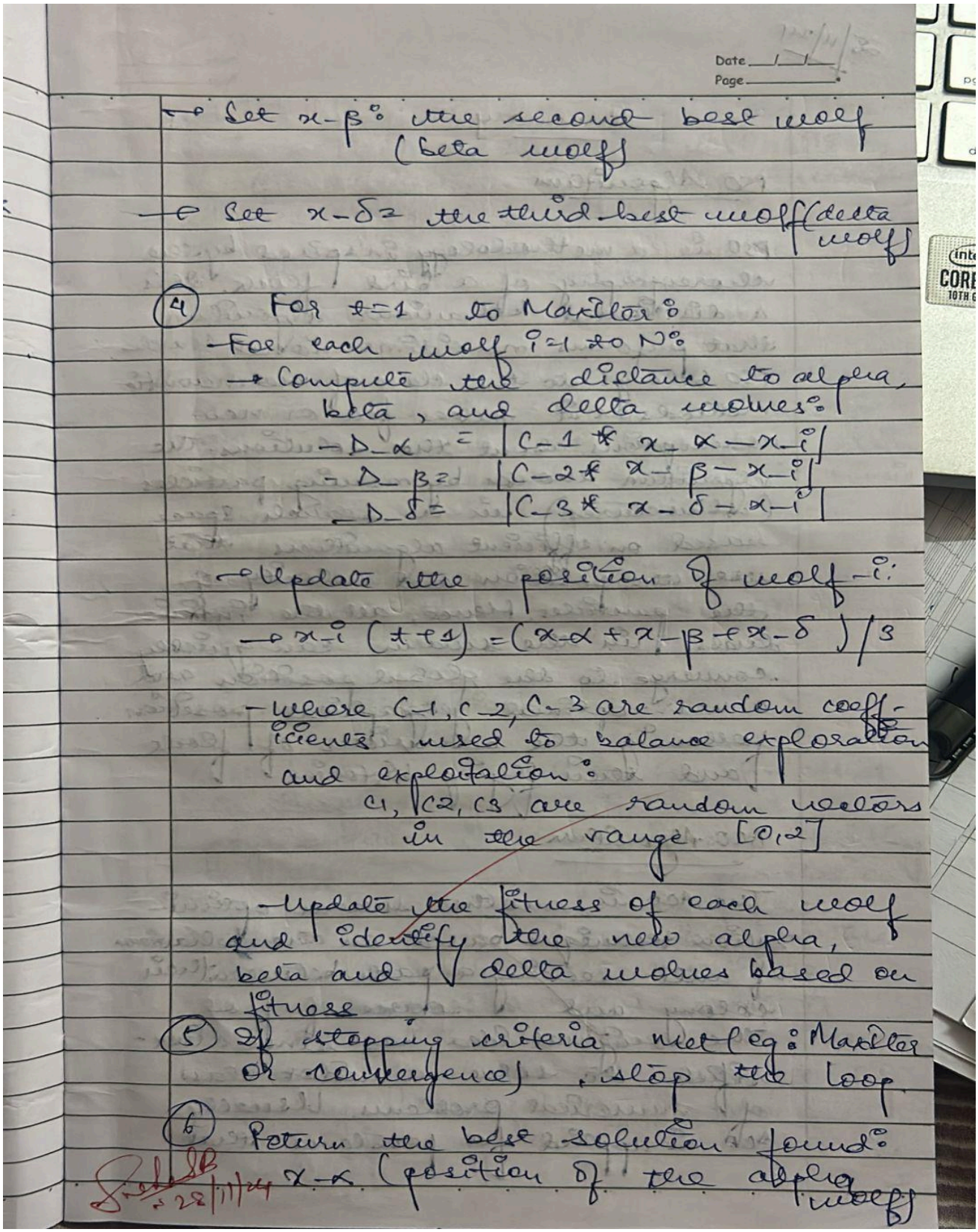
Summary: is a nature-inspired optimization algorithm that simulates the social hierarchy and hunting behaviour of grey wolves in the wild.

Algorithm: / Pseudocode:

① Initialise parameters
- N: Number of wolves (population size)
- MaxIter: Maximum no. of iterations
- D: Dimensionality of the search space
- Search space bounds: lower and upper limits.

② Initialise population of wolves
- For each wolf i=1 to N:
  - Randomly initialise position x_i in the search space (within bounds)
  - Evaluate fitness of x_i: fitness(x_i)

③ Identify the top three wolves:
- Set x_α = the wolf with the best fitness (alpha wolf)


Date __/__/__
Page _____

→ Set x_β° the second best wolf (beta wolf)

→ Set x_δ= the third best wolf (delta wolf)

(4) For t=1 to Max_iter°
- For each wolf i=1 to N°
  - → Compute the distance to alpha, beta, and delta wolves:
    - D_α = |C_1 * x_α − x_i°|
    - D_β = |C_2 * x_β − x_i°|
    - D_δ = |C_3 * x_δ − x_i°|
  - → Update the position of wolf-i:
    - → x_i (t+1) = (x_α + x_β + x_δ)/3
  - Where C_1, C_2, C_3 are random coefficients used to balance exploration and exploitation:
    C_1, C_2, C_3 are random vectors in the range [0,2]
  - Update the fitness of each wolf and identify the new alpha, beta and delta wolves based on fitness.

(5) If stopping criteria met (eg: Max_iter or convergence), stop the loop.

(6) Return the best solution found: x_α (position of the alpha wolf)

28/11/24

Code:

```
import numpy as np

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

class GreyWolfOptimizer:
    def __init__(self, objective_function, n_wolves, n_variables, max_iter, lb, ub):
        self.obj_func = objective_function  # Objective function
        self.n_wolves = n_wolves  # Number of wolves
        self.n_variables = n_variables  # Number of variables in the problem
        self.max_iter = max_iter  # Maximum number of iterations
        self.lb = lb  # Lower bound for the search space
        self.ub = ub  # Upper bound for the search space

        self.wolves = np.random.uniform(self.lb, self.ub, (self.n_wolves, self.n_variables))

        self.alpha = np.zeros(self.n_variables)
        self.beta = np.zeros(self.n_variables)
        self.delta = np.zeros(self.n_variables)
        self.alpha_score = float("inf")
        self.beta_score = float("inf")
        self.delta_score = float("inf")

    def update_wolves(self):
        fitness = np.apply_along_axis(self.obj_func, 1, self.wolves)

        sorted_indices = np.argsort(fitness)
        self.wolves = self.wolves[sorted_indices]
        fitness = fitness[sorted_indices]

        # Update alpha, beta, and delta wolves
        self.alpha = self.wolves[0]
        self.beta = self.wolves[1]
        self.delta = self.wolves[2]
        self.alpha_score = fitness[0]
        self.beta_score = fitness[1]
        self.delta_score = fitness[2]

    def optimize(self):
        for t in range(self.max_iter):
```

```
        A = 2 * np.random.random((self.n_wolves, self.n_variables)) - 1  # Random values for exploration
        C = 2 * np.random.random((self.n_wolves, self.n_variables))  # Random values for exploitation
        for i in range(self.n_wolves):
            D_alpha = np.abs(C[i] * self.alpha - self.wolves[i])  # Distance to alpha wolf
            D_beta = np.abs(C[i] * self.beta - self.wolves[i])  # Distance to beta wolf
            D_delta = np.abs(C[i] * self.delta - self.wolves[i])  # Distance to delta wolf

            self.wolves[i] = self.alpha - A[i] * D_alpha

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

        self.update_wolves()

        print(f"Iteration {t+1}/{self.max_iter}, Best Score: {self.alpha_score}")

    return self.alpha, self.alpha_score  # Return the best solution found

n_wolves = 30  # Number of wolves
n_variables = 5  # Number of decision variables
max_iter = 100  # Maximum number of iterations
lb = -10  # Lower bound of the search space
ub = 10  # Upper bound of the search space

gwo = GreyWolfOptimizer(objective_function, n_wolves, n_variables, max_iter, lb, ub)
best_solution, best_score = gwo.optimize()
print("Best Solution Found:", best_solution)
print("Best Score:", best_score)
```

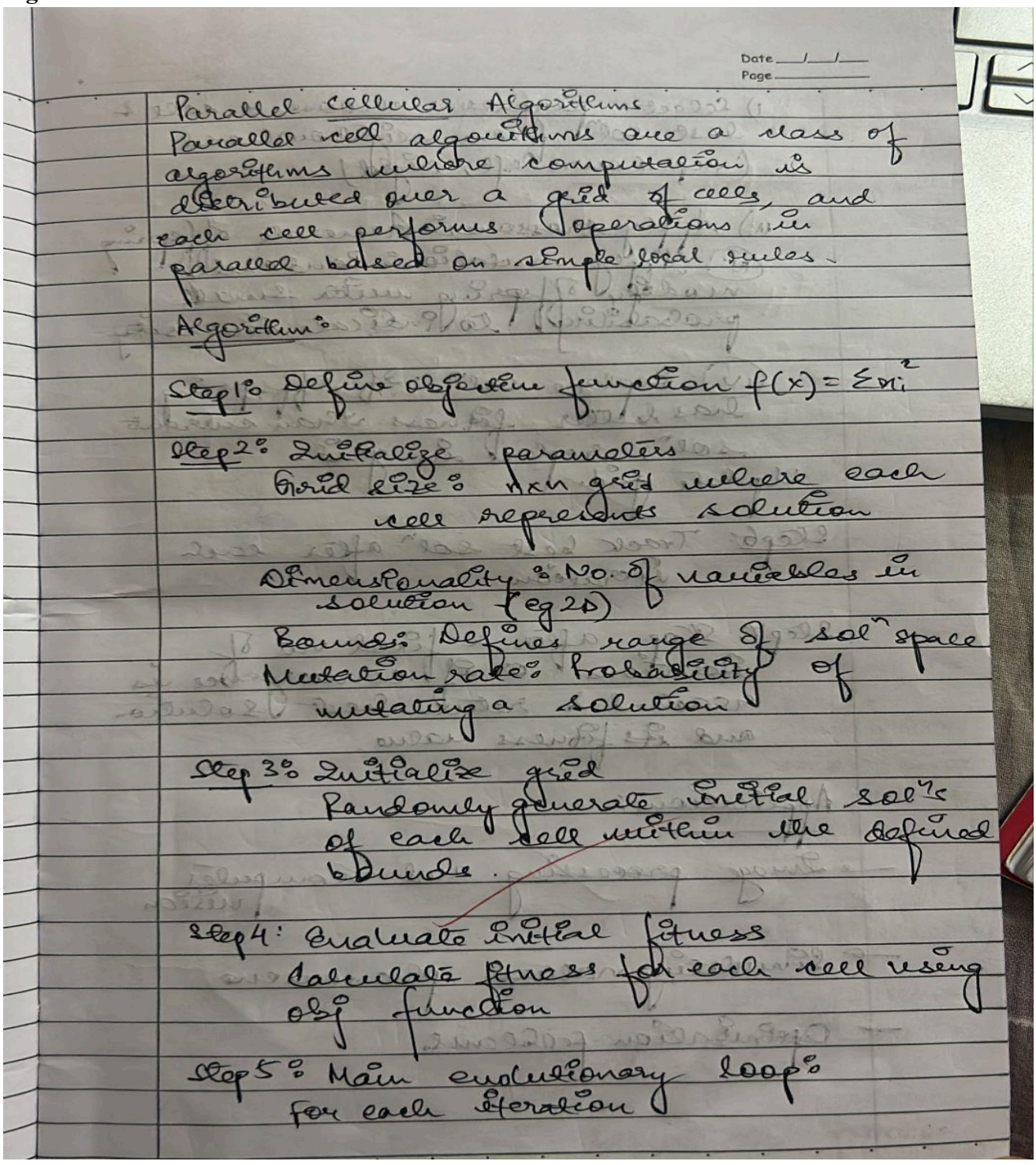
Output:

```
Iteration 100/100, Best Score: 1.985808550535119e-30
Best Solution Found: [-4.38373504e-17 -4.54363691e-16 -1.31663573e-15 -2.05502414e-16
  4.09828696e-17]
Best Score: 1.985808550535119e-30
```

Program 6

Parallel Cellular Algorithm

Algorithm:



Parallel cellular Algorithms

Parallel cell algorithms are a class of algorithms where computation is distributed over a grid of cells, and each cell performs operations in parallel based on simple local rules.

Algorithm:

Step 1: Define objective function $f(x) = \sum x_i^2$

Step 2: Initialize parameters

Grid size: nxn grid where each cell represents solution

Dimensionality: No. of variables in solution (eg 2D)

Bounds: Defines range of sol^n space

Mutation rate: Probability of mutating a solution

Step 3: Initialize grid

Randomly generate initial sol^{n}s of each cell within the defined bounds.

Step 4: Evaluate initial fitness

Calculate fitness for each cell using obj function

Step 5: Main evolutionary loop:

For each iteration

Date __/__/__
Page _____

i) Select parents: choose a cell & best sol^n from neighbourhood (up, down, left, right)

ii) Apply crossover: Generate offspring

iii) Apply mutation: Randomly modify offspring with small probability to indicate diversity

iv) Replace sol^n: If offspring has better fitness than current sol^n, replace in grid.

Update grid.

Step 6: Track best sol^n after each iteration

Step 7: Stop after a fixed no of iterations or if convergence is reached & return best solution and its fitness value.

Applications:

- Image processing and computer vision
- Simulating natural phenomena
- Optimisation problems.

Code:

```
import numpy as np
from multiprocessing import Pool
def update_cell(cell_index, grid, size):
    x, y = cell_index
    neighbors = [
        ((x-1) % size, y), ((x+1) % size, y),
        (x, (y-1) % size), (x, (y+1) % size)
    ]
    new_state = sum(grid[n[0], n[1]] for n in neighbors) % 2  # example: majority rule
    return (x, y, new_state)
def parallel_update(grid, size, num_iterations):
    pool = Pool(processes=4)
    for iteration in range(num_iterations):
        print(f"Iteration {iteration + 1}:")
        indices = [(x, y) for x in range(size) for y in range(size)]
        result = pool.starmap(update_cell, [(i, grid, size) for i in indices])

        for x, y, new_state in result:
            grid[x, y] = new_state
        print(grid)
    return grid
grid_size = 10
grid = np.random.randint(2, size=(grid_size, grid_size))
print("Initial state:")
print(grid)
num_iterations = 2
updated_grid = parallel_update(grid, grid_size, num_iterations)
```

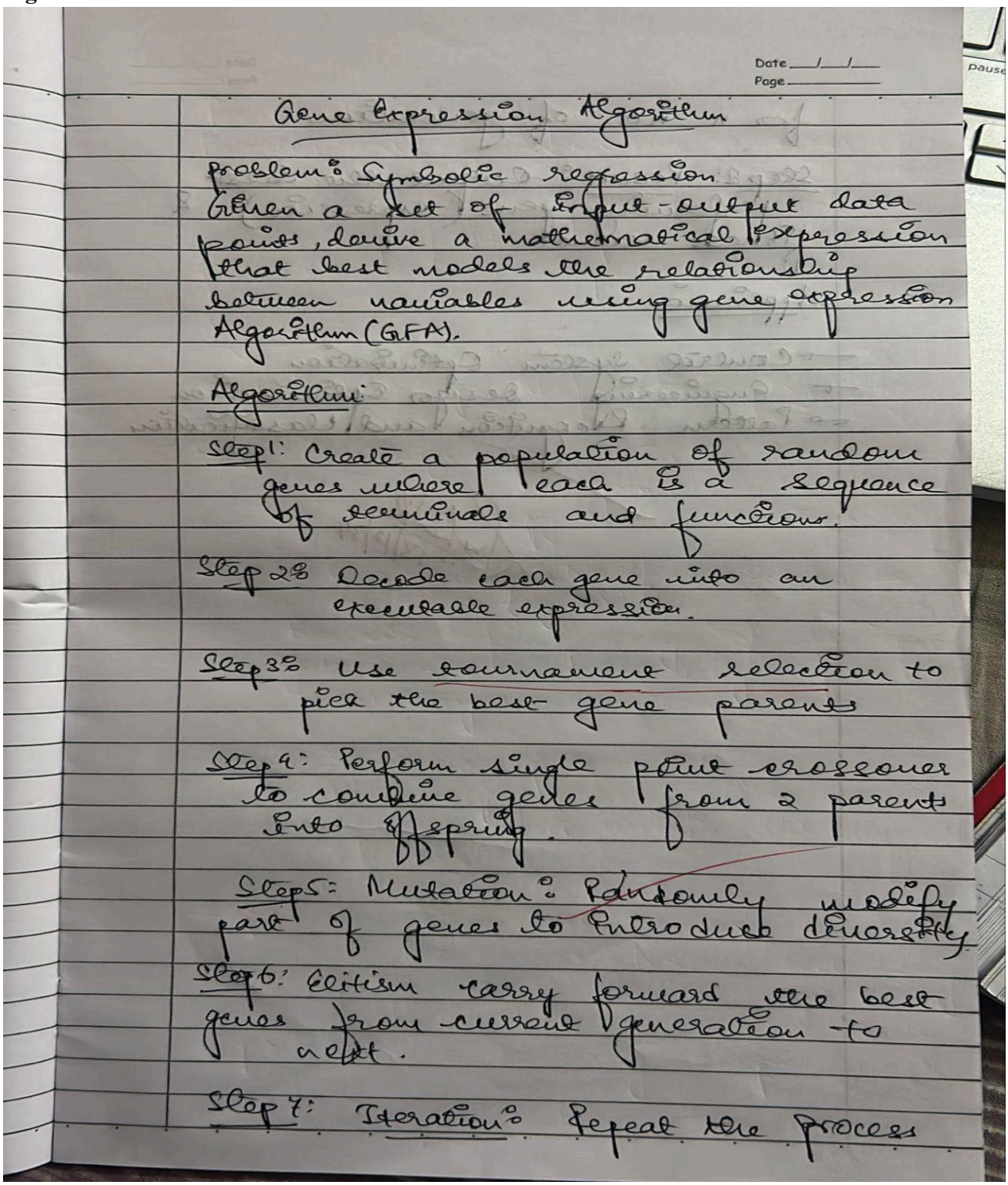
Output:

```
Iteration 1:
[[1 0 0 1]
 [1 0 1 0]
 [1 0 0 1]
 [0 1 0 1]]
Iteration 2:
[[0 0 0 0]
 [0 0 0 0]
 [0 0 0 0]
 [0 0 0 0]]
```

Program 7

Gene Expression Algorithm

Algorithm:



Gene Expression Algorithm

problem: Symbolic regression
Given a set of input-output data points, derive a mathematical expression that best models the relationship between variables using gene expression Algorithm (GEA).

Algorithm:

Step 1: Create a population of random genes where each is a sequence of terminals and functions.

Step 2: Decode each gene into an executable expression.

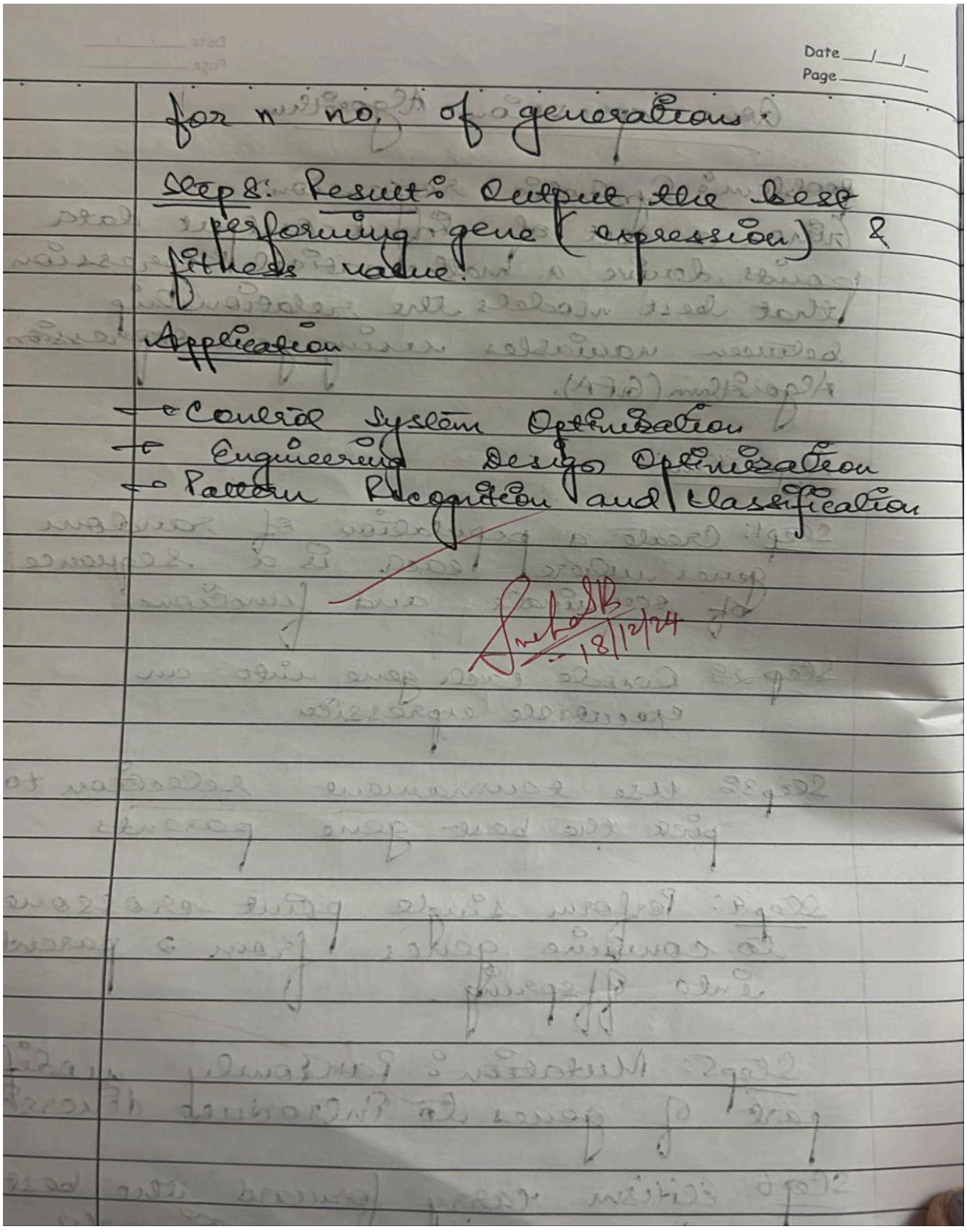
Step 3: Use tournament selection to pick the best gene parents

Step 4: Perform single point crossover to combine genes from 2 parents into offspring.

Step 5: Mutation: Randomly modify part of genes to introduce diversity

Step 6: Elitism carry forward the best genes from current generation to next.

Step 7: Iteration: Repeat the process


Date __/__/__
Page
for n no. of generations.
Step 8: Result: Output the best performing gene (expression) & fitness value.
Application
- Control System Optimization
- Engineering design Optimization
- Pattern Recognition and Classification
18/12/24

Code:

```
import random
import numpy as np
import operator

# Function set and terminal set
FUNCTIONS = {'+': operator.add, '-': operator.sub, '*': operator.mul, '/': operator.truediv}
TERMINALS = ['x', 1, 2, 3, 4]  # x and constants

def random_gene(length=10):
    """Generate a random chromosome (gene)."""
    return [random.choice(list(FUNCTIONS.keys()) + TERMINALS) for _ in range(length)]


def decode_chromosome(chromosome, x):
    """Decode chromosome into a functional expression tree (phenotype)."""
    stack = []
    for gene in chromosome:
        if gene in FUNCTIONS:  # If it's a function, pop arguments and apply
            if len(stack) < 2:  # Avoid errors if stack has fewer than 2 elements
                stack.append(0)
                continue
            b = stack.pop()
            a = stack.pop()
            try:
                result = FUNCTIONS[gene](a, b)
            except ZeroDivisionError:
                result = 1  # Avoid division by zero
            stack.append(result)
        elif gene == 'x':
            stack.append(x)
        else:
            stack.append(gene)
    return stack[0] if stack else 0  # Return top of stack as output


def fitness_function(chromosome, target_function, x_values):
    """Calculate fitness based on Mean Squared Error."""
    predictions = [decode_chromosome(chromosome, x) for x in x_values]
    targets = [target_function(x) for x in x_values]
    mse = np.mean([(p - t) ** 2 for p, t in zip(predictions, targets)])
```

```
    return mse


def selection(population, fitnesses):
    """Select individuals based on fitness (roulette wheel selection)."""
    total_fitness = sum(1 / (f + 1e-6) for f in fitnesses)  # Avoid division by zero
    probabilities = [(1 / (f + 1e-6)) / total_fitness for f in fitnesses]
    return population[np.random.choice(len(population), p=probabilities)]


def mutate(chromosome, mutation_rate=0.1):
    """Apply mutation to a chromosome."""
    new_chromosome = chromosome[:]
    for i in range(len(new_chromosome)):
        if random.random() < mutation_rate:
            new_chromosome[i] = random.choice(list(FUNCTIONS.keys()) + TERMINALS)
    return new_chromosome


def crossover(parent1, parent2):
    """Perform one-point crossover between two parents."""
    point = random.randint(1, len(parent1) - 1)
    child1 = parent1[:point] + parent2[point:]
    child2 = parent2[:point] + parent1[point:]
    return child1, child2


def gene_expression_algorithm(target_function, x_values, population_size=10, generations=20):
    """Main Gene Expression Algorithm."""
    # Initialize random population
    population = [random_gene() for _ in range(population_size)]

    print("Initial Population:")
    for i, chrom in enumerate(population):
        print(f"Chromosome {i}: {chrom}")

    for generation in range(generations):
        print(f"\nGeneration {generation + 1}:")
        # Calculate fitness for each individual
        fitnesses = [fitness_function(chrom, target_function, x_values) for chrom in population]
        for i, (chrom, fit) in enumerate(zip(population, fitnesses)):
```

```
            print(f"Chromosome {i}: {chrom}, Fitness: {fit:.4f}")

        # Select the next generation
        new_population = []
        for _ in range(population_size // 2):
            parent1 = selection(population, fitnesses)
            parent2 = selection(population, fitnesses)
            child1, child2 = crossover(parent1, parent2)
            child1 = mutate(child1)
            child2 = mutate(child2)
            new_population.extend([child1, child2])
        population = new_population

    # Final results
    print("\nFinal Population and Fitness:")
    fitnesses = [fitness_function(chrom, target_function, x_values) for chrom in population]
    for i, (chrom, fit) in enumerate(zip(population, fitnesses)):
        print(f"Chromosome {i}: {chrom}, Fitness: {fit:.4f}")

    best_index = np.argmin(fitnesses)
    print("\nBest Solution:")
    print(f"Chromosome: {population[best_index]}, Fitness: {fitnesses[best_index]:.4f}")

# Target function for regression
def target_function(x):
    return x**2 + 2*x + 1  # Example: f(x) = x^2 + 2x + 1

# Input values
x_values = np.linspace(-10, 10, 20)

# Run the algorithm
gene_expression_algorithm(target_function, x_values, population_size=10, generations=10)
```

Output:

```
Best Solution:
Chromosome: [1, 3, '+', 2, 1, 4, '*', '*', '*', 3], Fitness: 1259.2067
<ipython-input-3-6df17022c257>:25: RuntimeWarning: divide by zero encountered in scalar divide
  result = FUNCTIONS[gene](a, b)
```