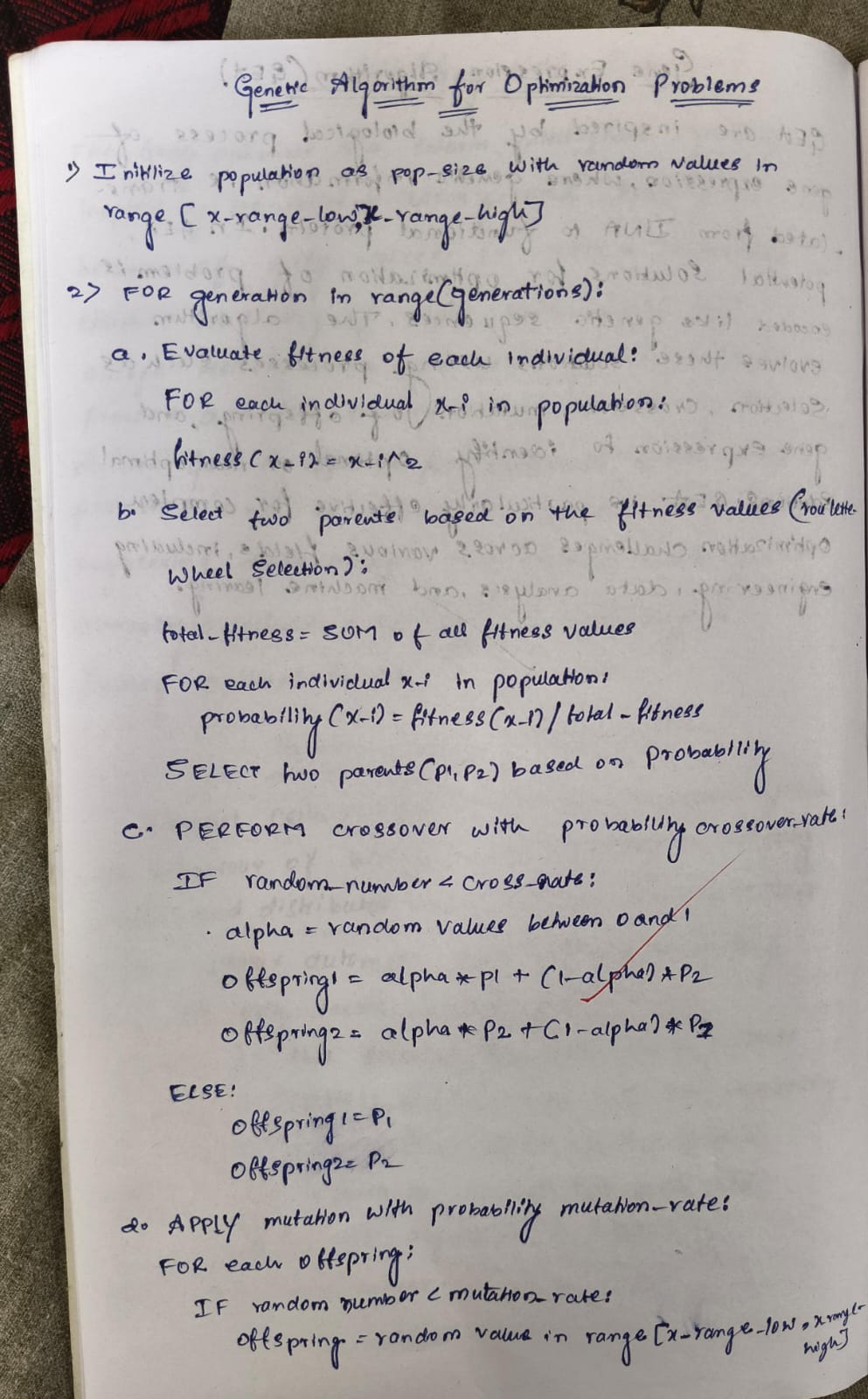
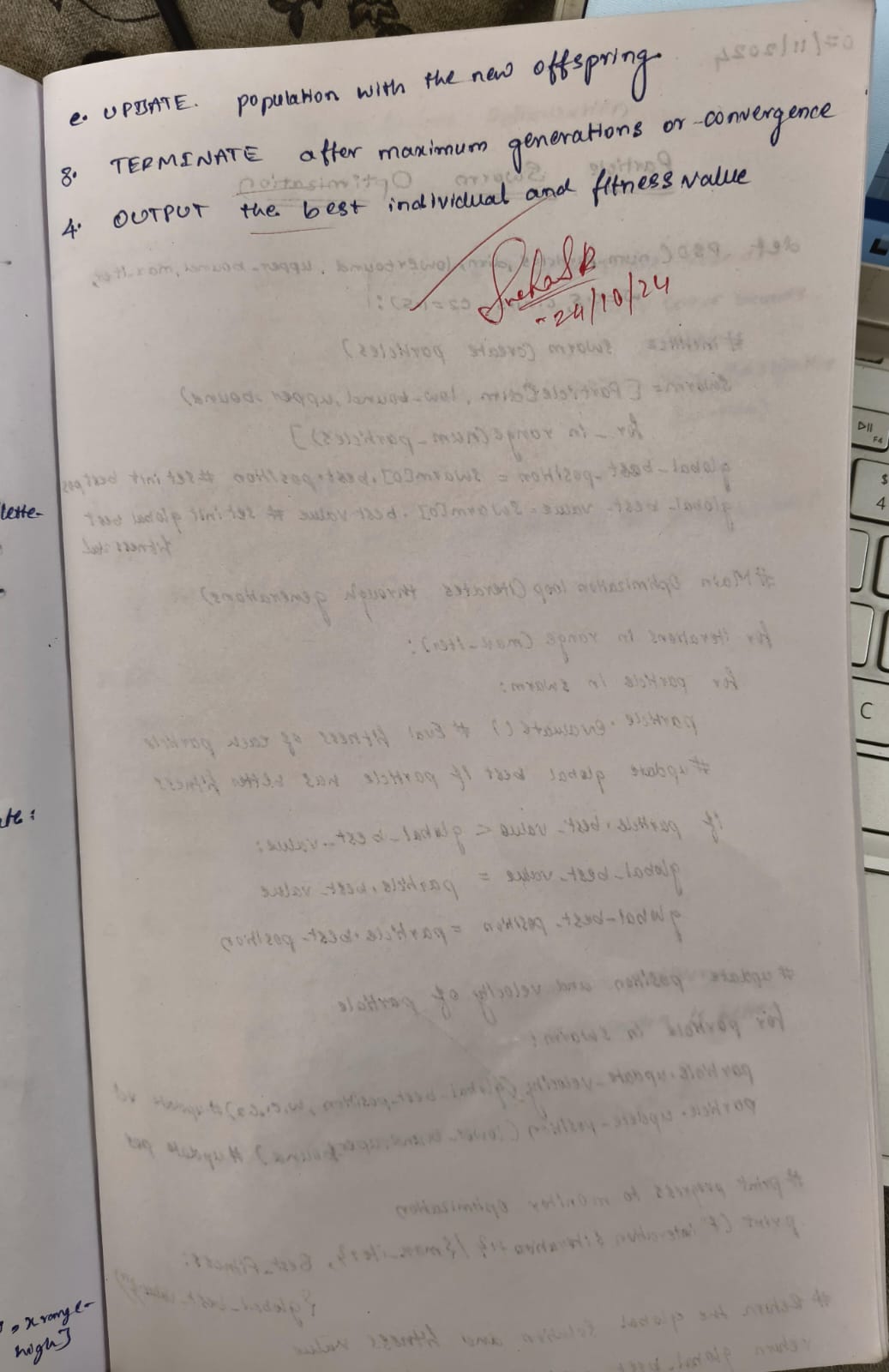
**Program 1: Genetic Algorithm for Optimization on Problems**

**Algorithm:**

****

****

**Code:**

#lab-2: genetic

import numpy as np

import random

# Objective function to maximize

def objective\_function(x):

    return x \*\* 2

# Initialize parameters

population\_size = 100

num\_generations = 50

mutation\_rate = 0.1

crossover\_rate = 0.7

range\_min = -10

range\_max = 10

# Create initial population

def initialize\_population(size, min\_val, max\_val):

    return np.random.uniform(min\_val, max\_val, size)

# Evaluate fitness of the population

def evaluate\_fitness(population):

    return np.array([objective\_function(x) for x in population])

# Selection using roulette-wheel method

def selection(population, fitness):

    total\_fitness = np.sum(fitness)

    probabilities = fitness / total\_fitness

    return population[np.random.choice(range(len(population)), size=2, p=probabilities)]

# Crossover between two parents

def crossover(parent1, parent2):

    if random.random() < crossover\_rate:

        return (parent1 + parent2) / 2  # Simple averaging for crossover

    return parent1  # No crossover

# Mutation of an individual

def mutate(individual):

    if random.random() < mutation\_rate:

        return np.random.uniform(range\_min, range\_max)

    return individual

# Genetic Algorithm function

def genetic\_algorithm():

    # Step 1: Initialize population

    population = initialize\_population(population\_size, range\_min, range\_max)

    for generation in range(num\_generations):

        # Step 2: Evaluate fitness

        fitness = evaluate\_fitness(population)

        # Track the best solution

        best\_index = np.argmax(fitness)

        best\_solution = population[best\_index]

        best\_fitness = fitness[best\_index]

        # print(f"Generation {generation + 1}: Best Solution = {best\_solution}, Fitness = {best\_fitness}")

        # Step 3: Create new population

        new\_population = []

        for \_ in range(population\_size):

            # Select parents

            parent1, parent2 = selection(population, fitness)

            # Crossover to create offspring

            offspring = crossover(parent1, parent2)

            # Mutate offspring

            offspring = mutate(offspring)

            new\_population.append(offspring)

        # Step 6: Replace old population with new population

        population = np.array(new\_population)

    return best\_solution, best\_fitness

# Run the Genetic Algorithm

best\_solution, best\_fitness = genetic\_algorithm()

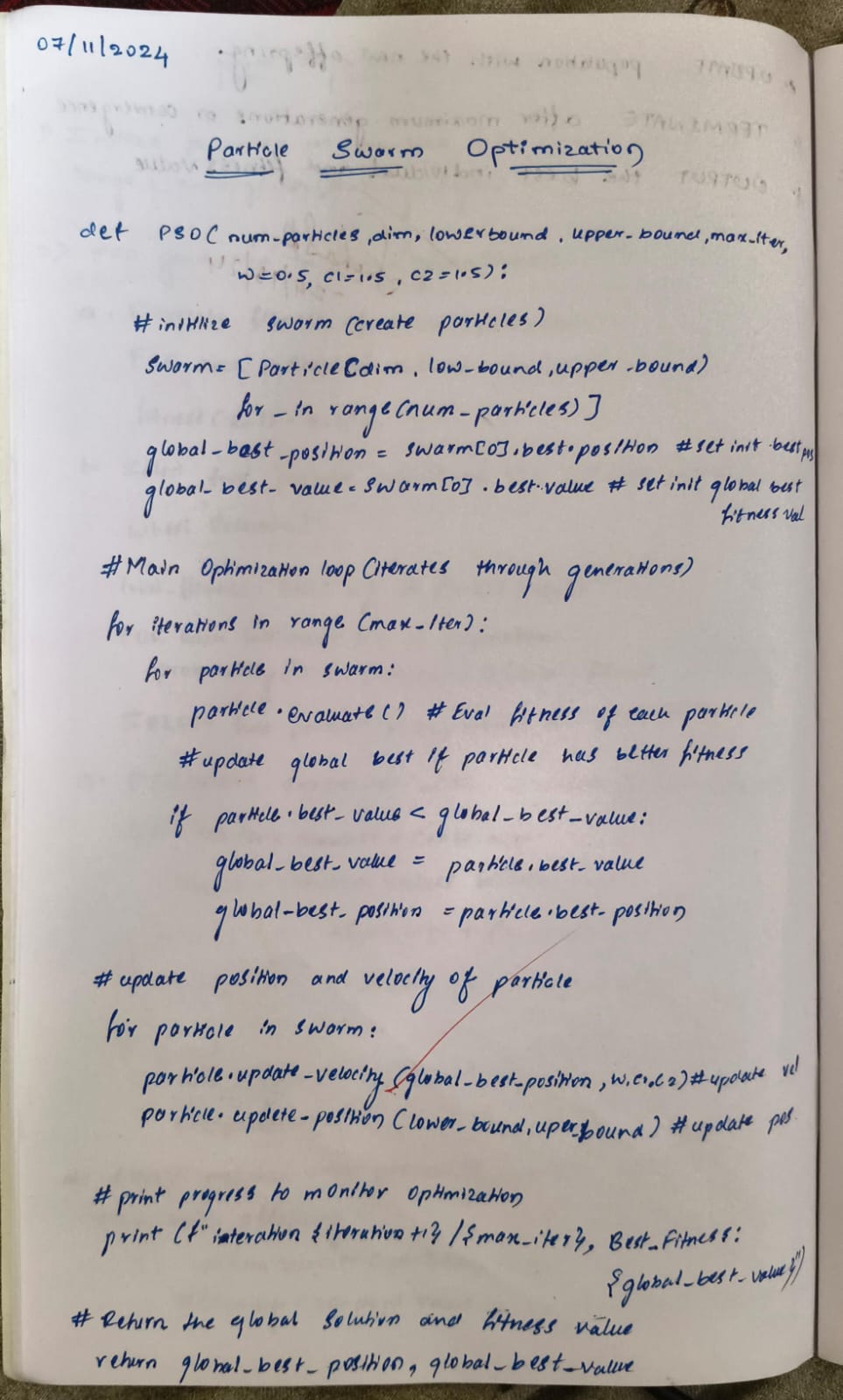
print(f"Best Solution Found: {best\_solution}, Fitness: {best\_fitness}")

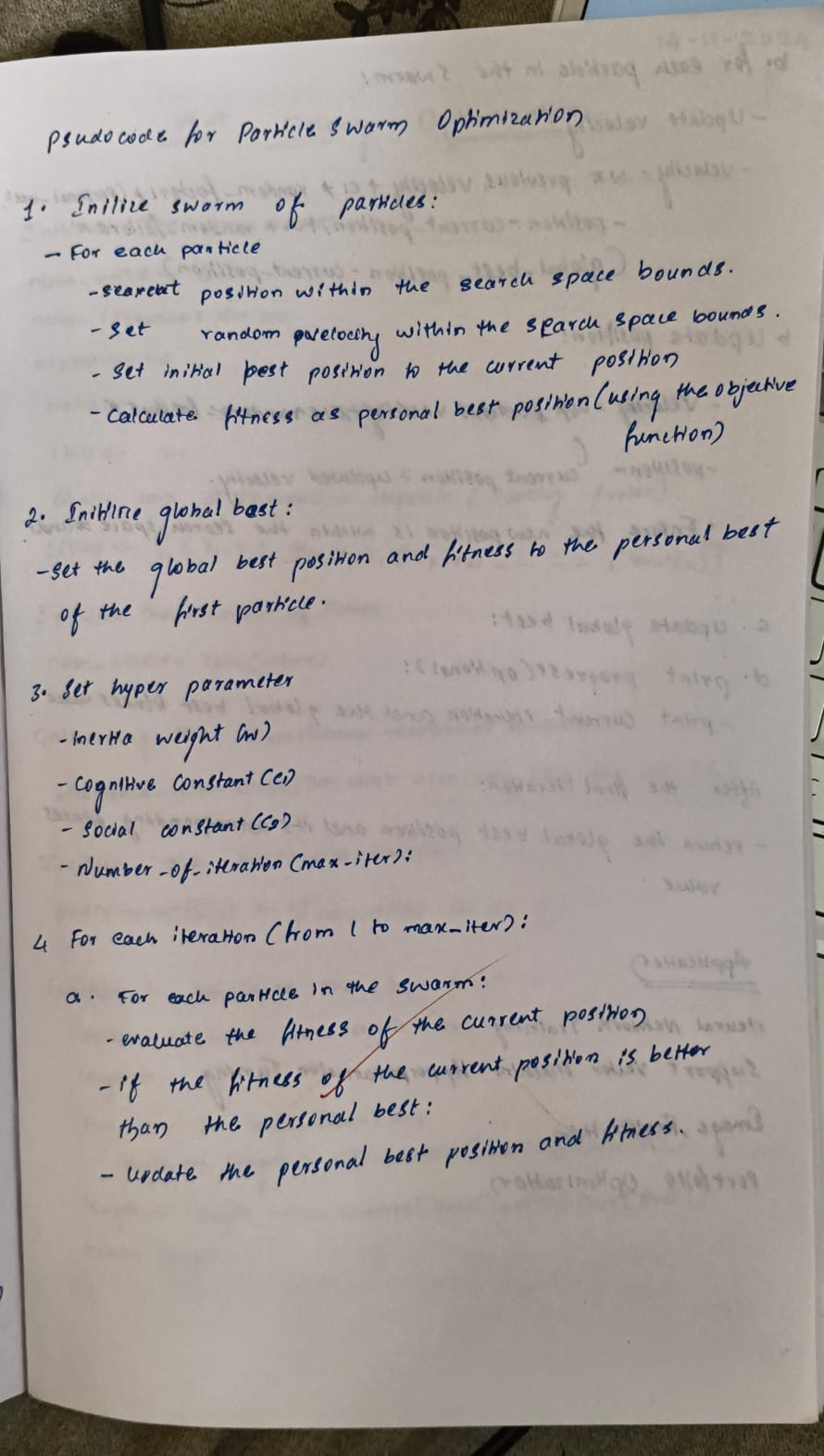
OUTPUT:

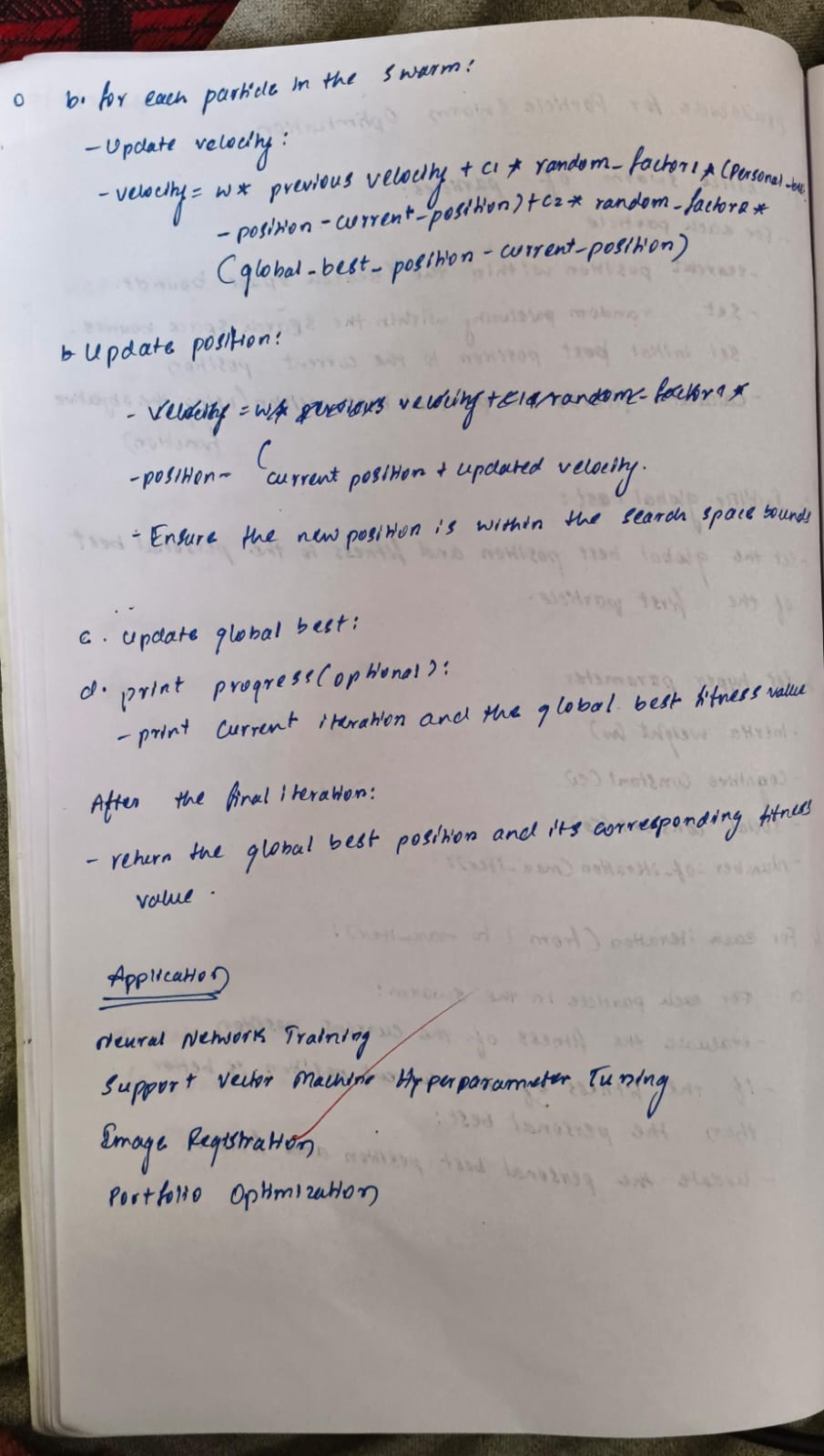


**Program 2: Particle Swarm Optimization for Function Optimization**

**Algorithm:**

****

****

****

**Code:**

#lab-3: pso

import numpy as np

import random

# Define the optimization problem (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])

# Particle Swarm Optimization (PSO) implementation

class Particle:

    def \_\_init\_\_(self, dimension, lower\_bound, upper\_bound):

        # Initialize the particle position and velocity randomly

        self.position = np.random.uniform(lower\_bound, upper\_bound, dimension)

        self.velocity = np.random.uniform(-1, 1, dimension)

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

        self.best\_value = rastrigin(self.position)

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

        # Update the velocity of the particle

        r1 = np.random.rand(len(self.position))

        r2 = np.random.rand(len(self.position))

        # Inertia term

        inertia = w \* self.velocity

        # Cognitive term (individual best)

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

        # Social term (global best)

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

        # Update velocity

        self.velocity = inertia + cognitive + social

    def update\_position(self, lower\_bound, upper\_bound):

        # Update the position of the particle

        self.position = self.position + self.velocity

        # Ensure the particle stays within the bounds

        self.position = np.clip(self.position, lower\_bound, upper\_bound)

    def evaluate(self):

        # Evaluate the fitness of the particle

        fitness = rastrigin(self.position)

        # Update the particle's best position if necessary

        if fitness < self.best\_value:

            self.best\_value = fitness

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

def particle\_swarm\_optimization(dim, lower\_bound, upper\_bound, num\_particles=30, max\_iter=100, w=0.5, c1=1.5, c2=1.5):

    # Initialize particles

    particles = [Particle(dim, lower\_bound, upper\_bound) for \_ in range(num\_particles)]

    # Initialize the global best position and value

    global\_best\_position = particles[0].best\_position

    global\_best\_value = particles[0].best\_value

    for i in range(max\_iter):

        # Update each particle

        for particle in particles:

            particle.update\_velocity(global\_best\_position, w, c1, c2)

            particle.update\_position(lower\_bound, upper\_bound)

            particle.evaluate()

            # Update global best position if needed

            if particle.best\_value < global\_best\_value:

                global\_best\_value = particle.best\_value

                global\_best\_position = np.copy(particle.best\_position)

        # Optionally print the progress

        if (i+1 ) % 10 == 0:

            print(f"Iteration {i+1 }/{max\_iter} - Best Fitness: {global\_best\_value}")

    return global\_best\_position, global\_best\_value

# Set the parameters for the PSO algorithm

dim = 2                # Number of dimensions for the function

lower\_bound = -5.12    # Lower bound of the search space

upper\_bound = 5.12     # Upper bound of the search space

num\_particles = 30     # Number of particles in the swarm

max\_iter = 100         # Number of iterations

# Run the PSO

best\_position, best\_value = particle\_swarm\_optimization(dim, lower\_bound, upper\_bound, num\_particles, max\_iter)

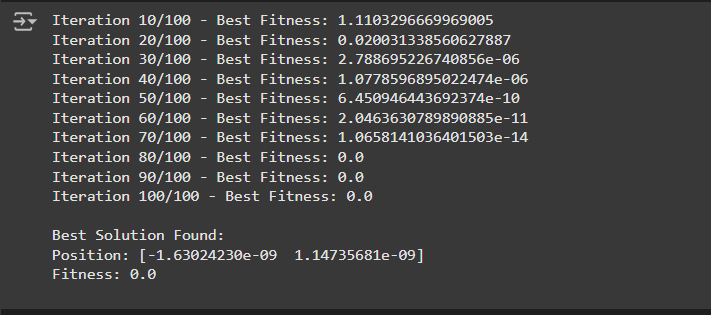
# Output the best solution found

print("\nBest Solution Found:")

print("Position:", best\_position)

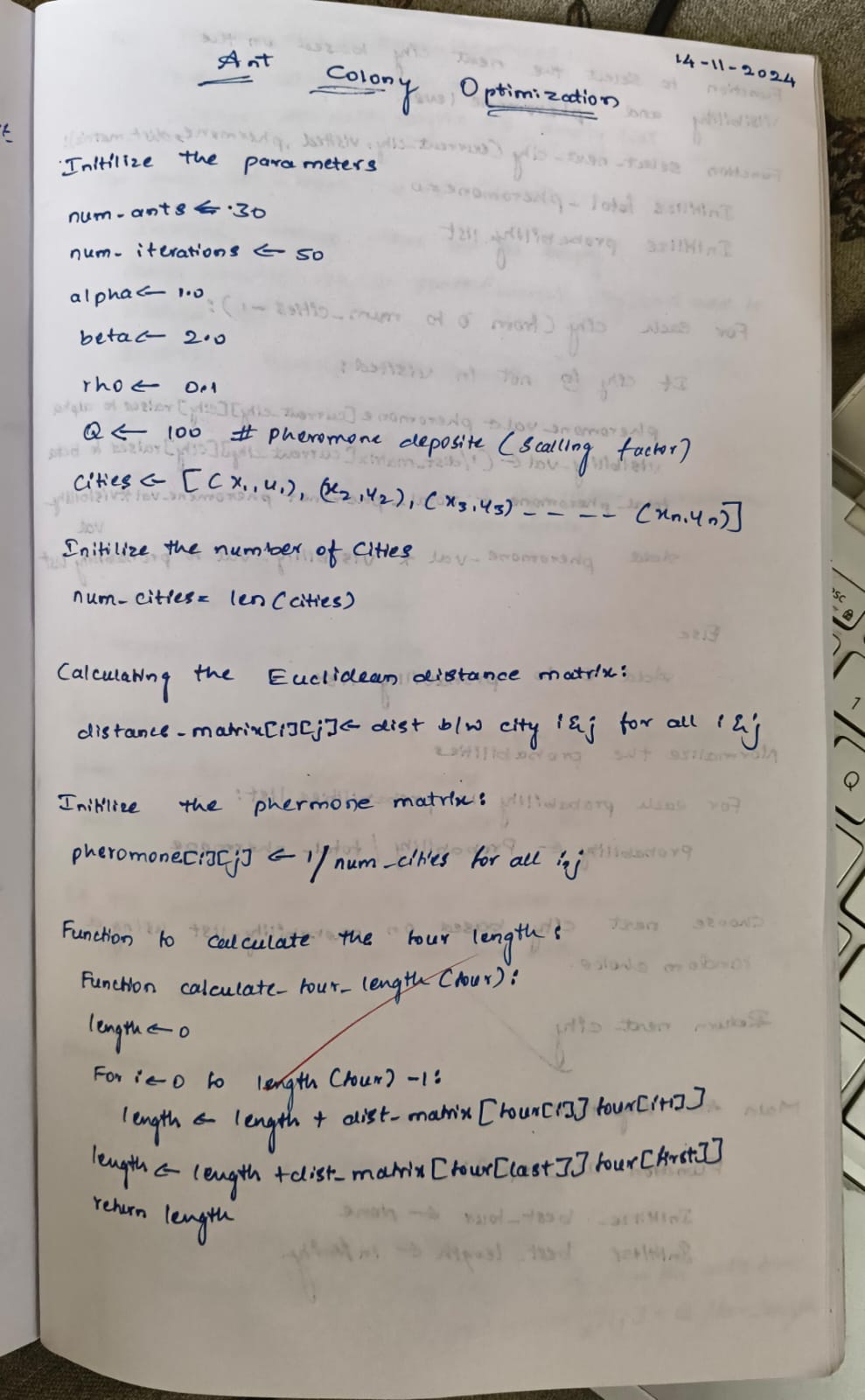
print("Fitness:", best\_value)

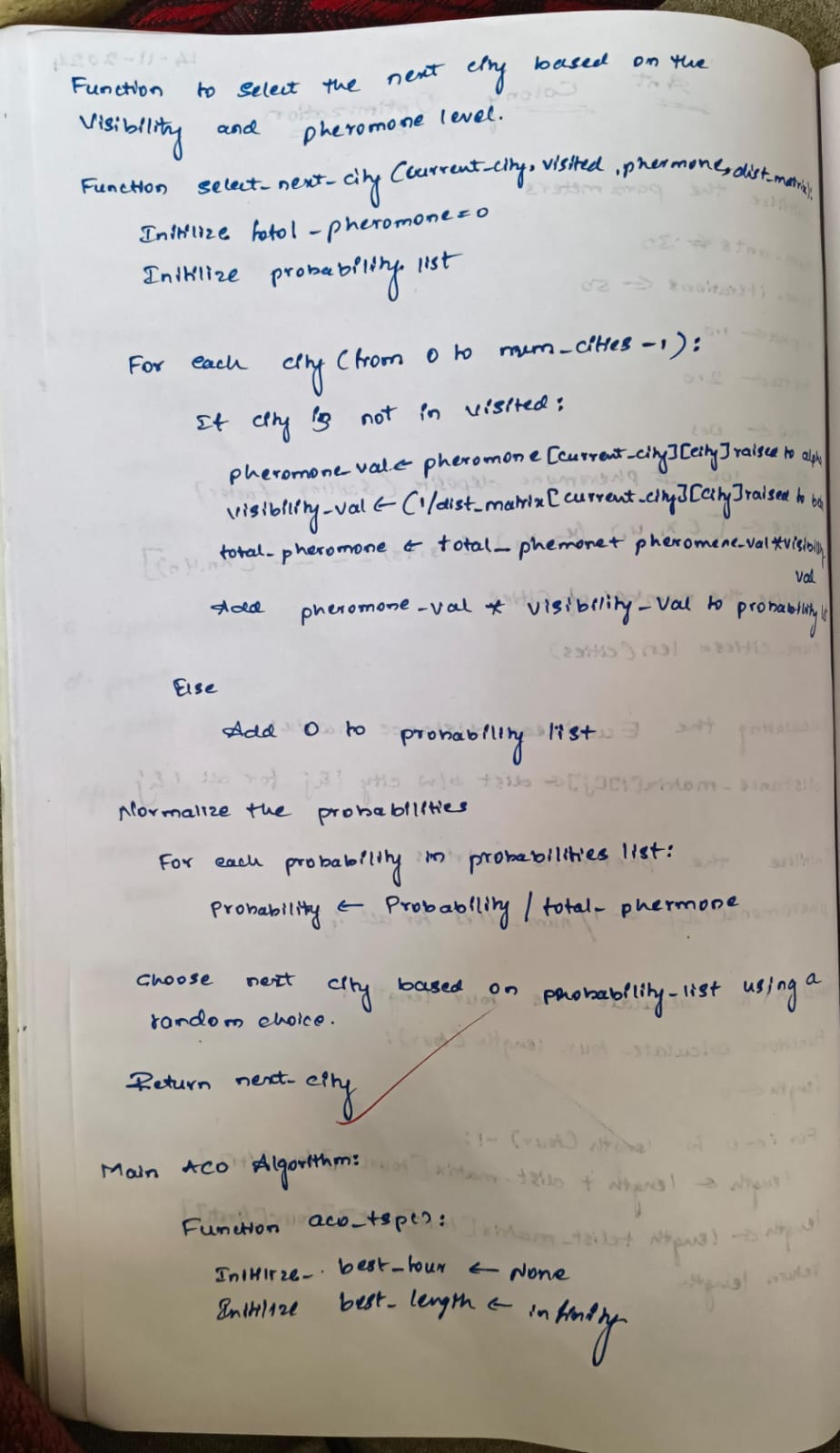
OUTPUT:

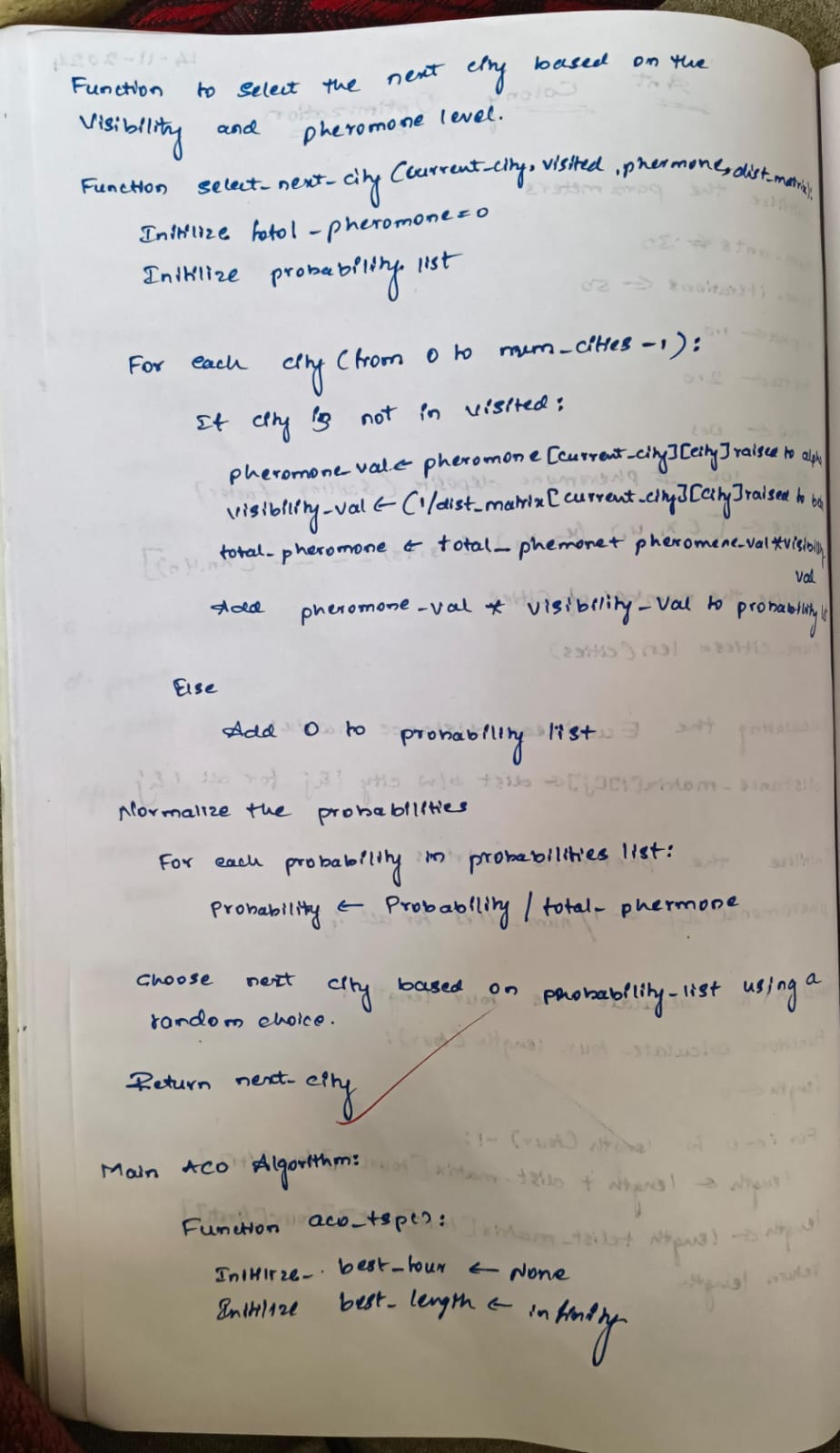


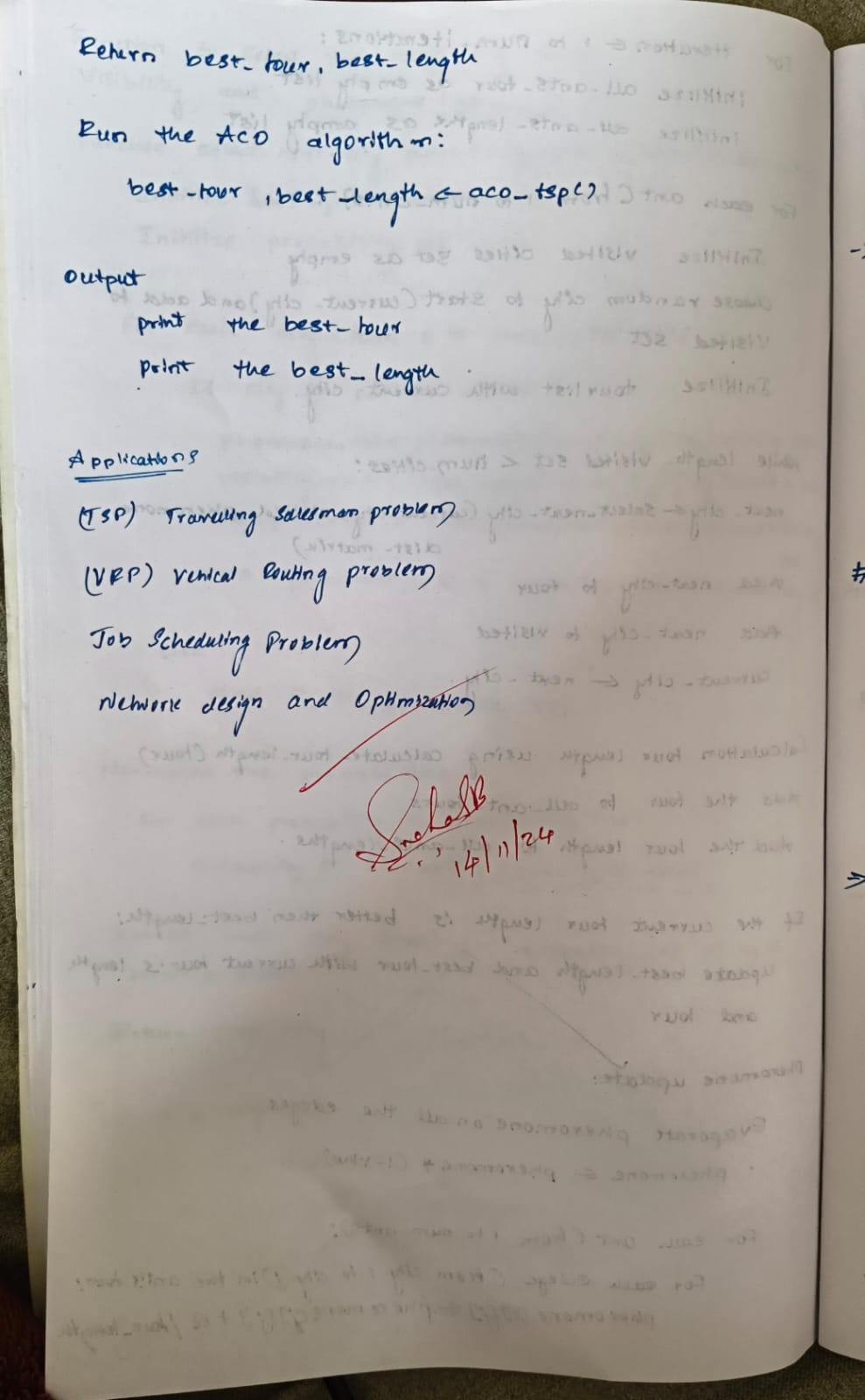
**Program 3: Ant Colony Optimization for the Traveling Salesman Problem**

**Algorithm:**

****

****

****

****

**Code:**

#ant colony

import numpy as np

import matplotlib.pyplot as plt

# 1. Define the Problem: Create a set of cities with their coordinates

cities = np.array([

    [0, 0],  # City 0

    [1, 5],  # City 1

    [5, 1],  # City 2

    [6, 4],  # City 3

    [7, 8],  # City 4

])

# Calculate the distance matrix between each pair of cities

def calculate\_distances(cities):

    num\_cities = len(cities)

    distances = np.zeros((num\_cities, num\_cities))

    for i in range(num\_cities):

        for j in range(num\_cities):

            distances[i][j] = np.linalg.norm(cities[i] - cities[j])

    return distances

distances = calculate\_distances(cities)

# 2. Initialize Parameters

num\_ants = 10

num\_cities = len(cities)

alpha = 1.0  # Influence of pheromone

beta = 5.0   # Influence of heuristic (inverse distance)

rho = 0.5    # Evaporation rate

num\_iterations = 30

initial\_pheromone = 1.0

# Pheromone matrix initialization

pheromone = np.ones((num\_cities, num\_cities)) \* initial\_pheromone

# 3. Heuristic information (Inverse of distance)

def heuristic(distances):

    with np.errstate(divide='ignore'):  # Ignore division by zero

        return 1 / distances

eta = heuristic(distances)

# 4. Choose next city probabilistically based on pheromone and heuristic info

def choose\_next\_city(pheromone, eta, visited):

    probs = []

    for j in range(num\_cities):

        if j not in visited:

            pheromone\_ij = pheromone[visited[-1], j] \*\* alpha

            heuristic\_ij = eta[visited[-1], j] \*\* beta

            probs.append(pheromone\_ij \* heuristic\_ij)

        else:

            probs.append(0)

    probs = np.array(probs)

    return np.random.choice(range(num\_cities), p=probs / probs.sum())

# Construct solution for a single ant

def construct\_solution(pheromone, eta):

    tour = [np.random.randint(0, num\_cities)]

    while len(tour) < num\_cities:

        next\_city = choose\_next\_city(pheromone, eta, tour)

        tour.append(next\_city)

    return tour

# 5. Update pheromones after all ants have constructed their tours

def update\_pheromones(pheromone, all\_tours, distances, best\_tour):

    pheromone \*= (1 - rho)  # Evaporate pheromones

    # Add pheromones for each ant's tour

    for tour in all\_tours:

        tour\_length = sum([distances[tour[i], tour[i + 1]] for i in range(-1, num\_cities - 1)])

        for i in range(-1, num\_cities - 1):

            pheromone[tour[i], tour[i + 1]] += 1.0 / tour\_length

    # Increase pheromones on the best tour

    best\_length = sum([distances[best\_tour[i], best\_tour[i + 1]] for i in range(-1, num\_cities - 1)])

    for i in range(-1, num\_cities - 1):

        pheromone[best\_tour[i], best\_tour[i + 1]] += 1.0 / best\_length

# 6. Main ACO Loop: Iterate over multiple iterations to find the best solution

def run\_aco(distances, num\_iterations):

    pheromone = np.ones((num\_cities, num\_cities)) \* initial\_pheromone

    best\_tour = None

    best\_length = float('inf')

    for iteration in range(num\_iterations):

        all\_tours = [construct\_solution(pheromone, eta) for \_ in range(num\_ants)]

        all\_lengths = [sum([distances[tour[i], tour[i + 1]] for i in range(-1, num\_cities - 1)]) for tour in all\_tours]

        current\_best\_length = min(all\_lengths)

        current\_best\_tour = all\_tours[all\_lengths.index(current\_best\_length)]

        if current\_best\_length < best\_length:

            best\_length = current\_best\_length

            best\_tour = current\_best\_tour

        update\_pheromones(pheromone, all\_tours, distances, best\_tour)

        print(f"Iteration {iteration + 1}, Best Length: {best\_length}")

    return best\_tour, best\_length

# Run the ACO algorithm

best\_tour, best\_length = run\_aco(distances, num\_iterations)

# 7. Output the Best Solution

print(f"Best Tour: {best\_tour}")

print(f"Best Tour Length: {best\_length}")

# 8. Plot the Best Route

def plot\_route(cities, best\_tour):

    plt.figure(figsize=(8, 6))

    for i in range(len(cities)):

        plt.scatter(cities[i][0], cities[i][1], color='red')

        plt.text(cities[i][0], cities[i][1], f"City {i}", fontsize=12)

    # Plot the tour as lines connecting the cities

    tour\_cities = np.array([cities[i] for i in best\_tour] + [cities[best\_tour[0]]])  # Complete the loop by returning to the start

    plt.plot(tour\_cities[:, 0], tour\_cities[:, 1], linestyle='-', marker='o', color='blue')

    plt.title(f"Best Tour (Length: {best\_length})")

    plt.xlabel("X Coordinate")

    plt.ylabel("Y Coordinate")

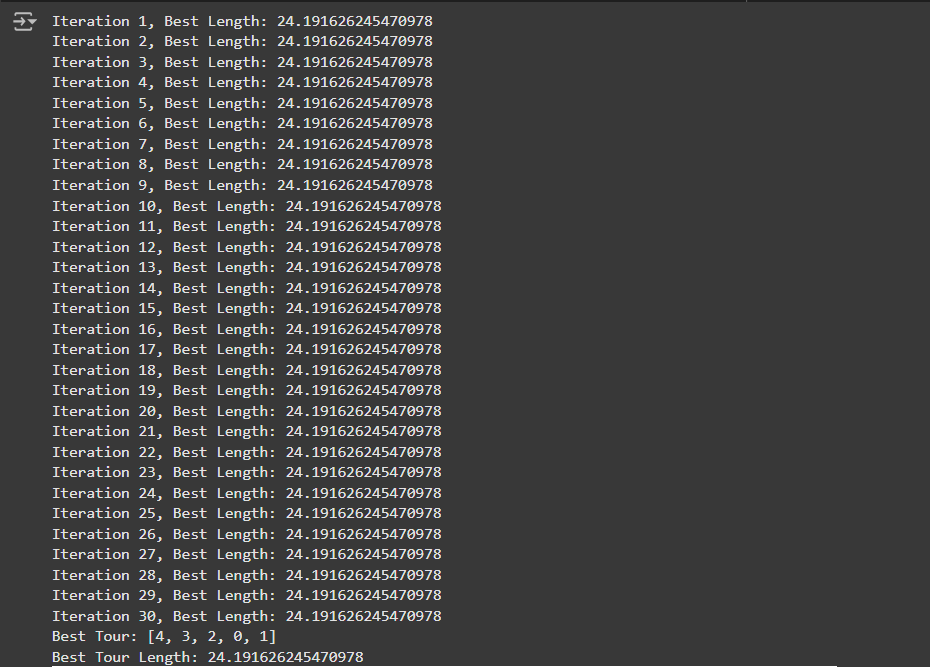
    plt.grid(True)

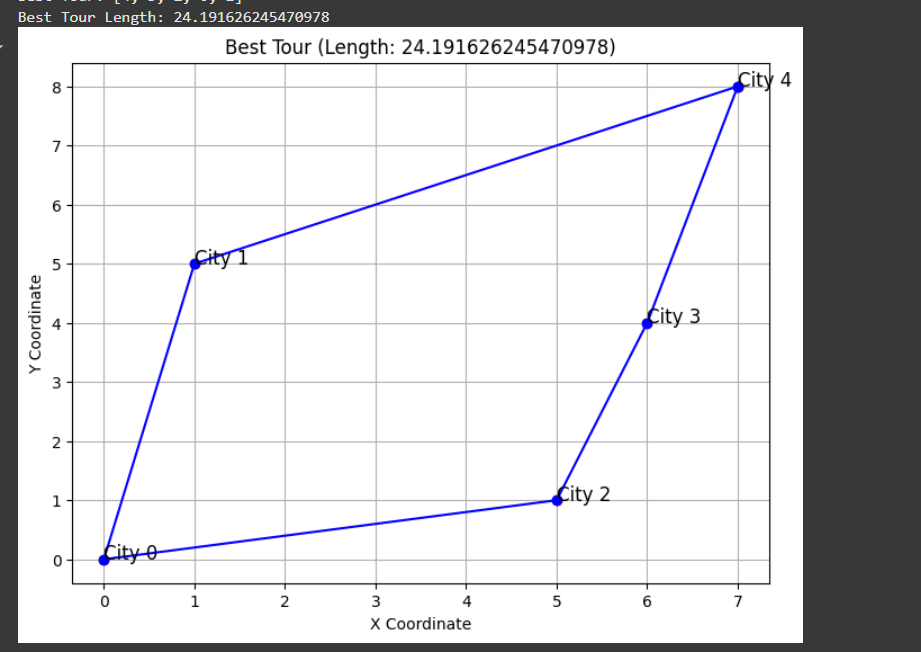
    plt.show()

# Call the plot function

plot\_route(cities, best\_tour)

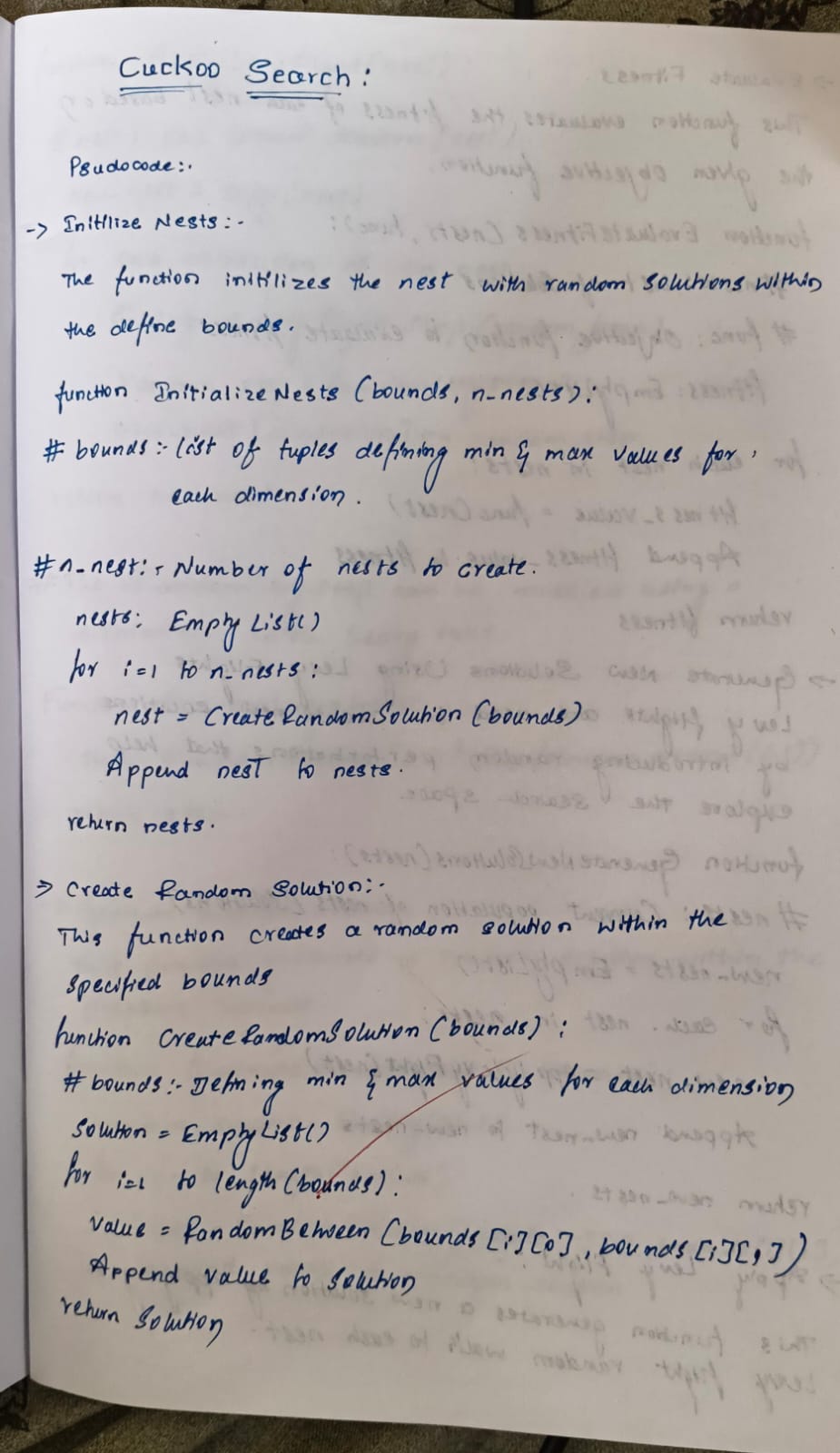
OUTPUT:

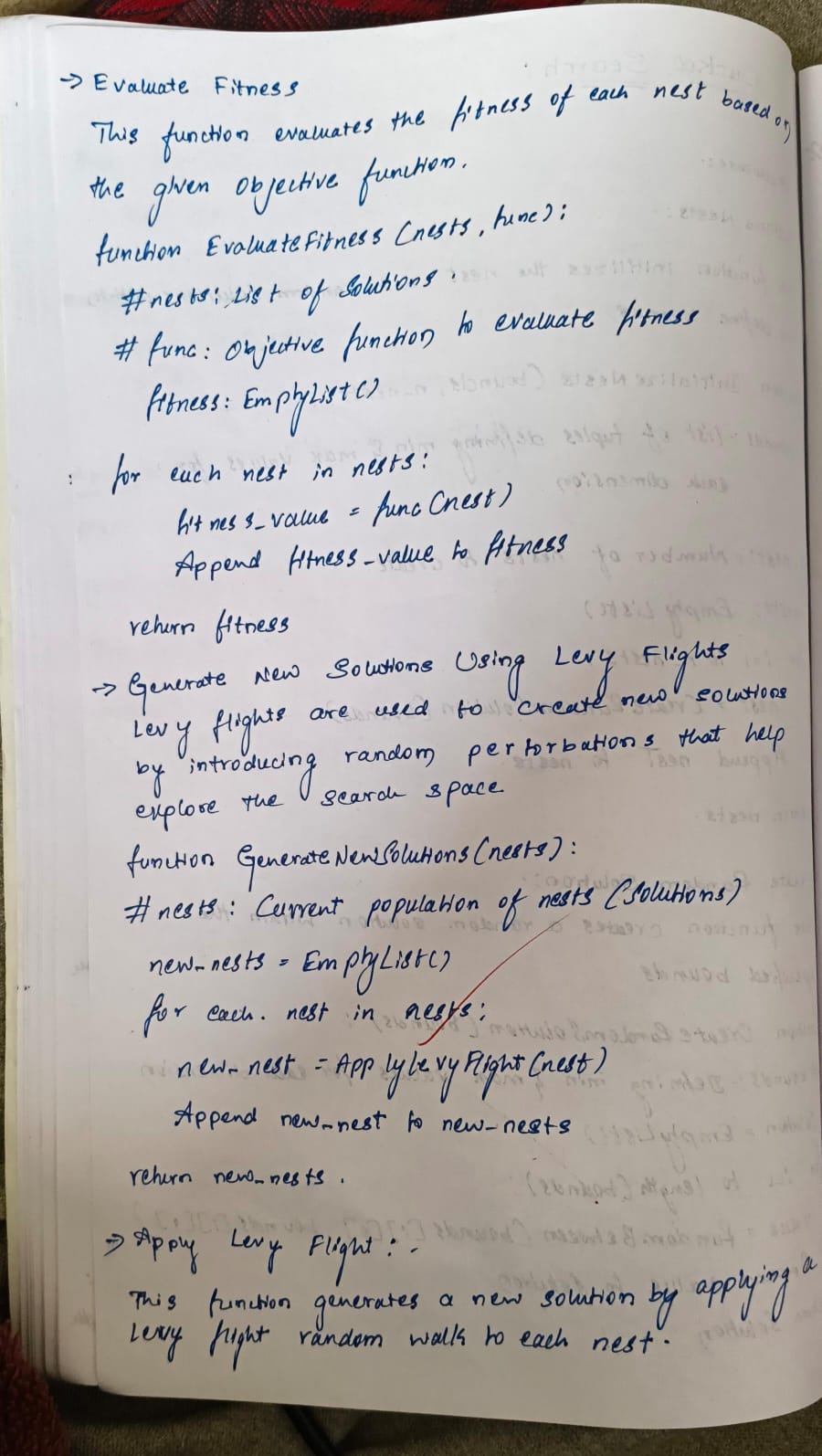


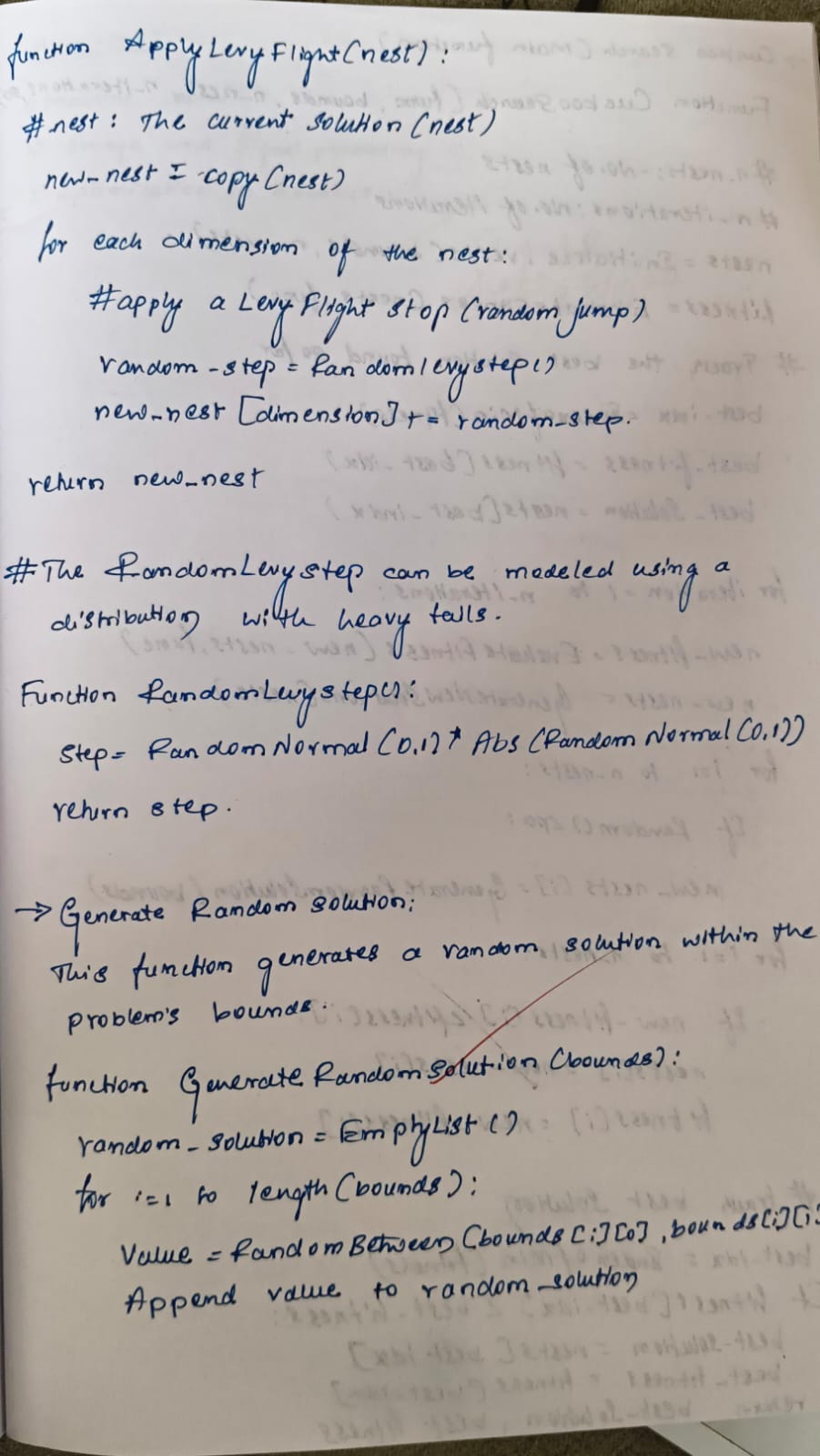


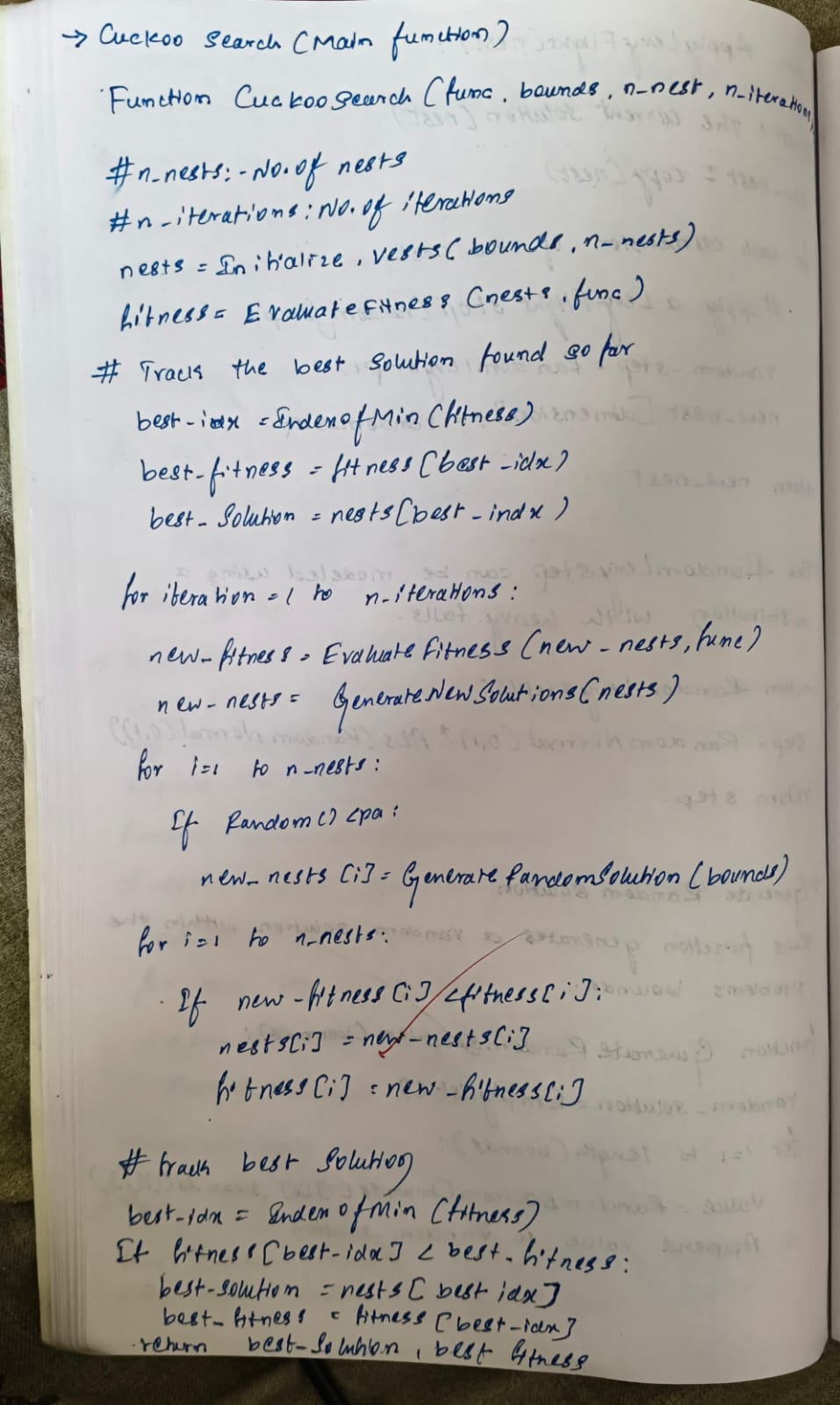
**Program 4: Cuckoo Search (CS)**

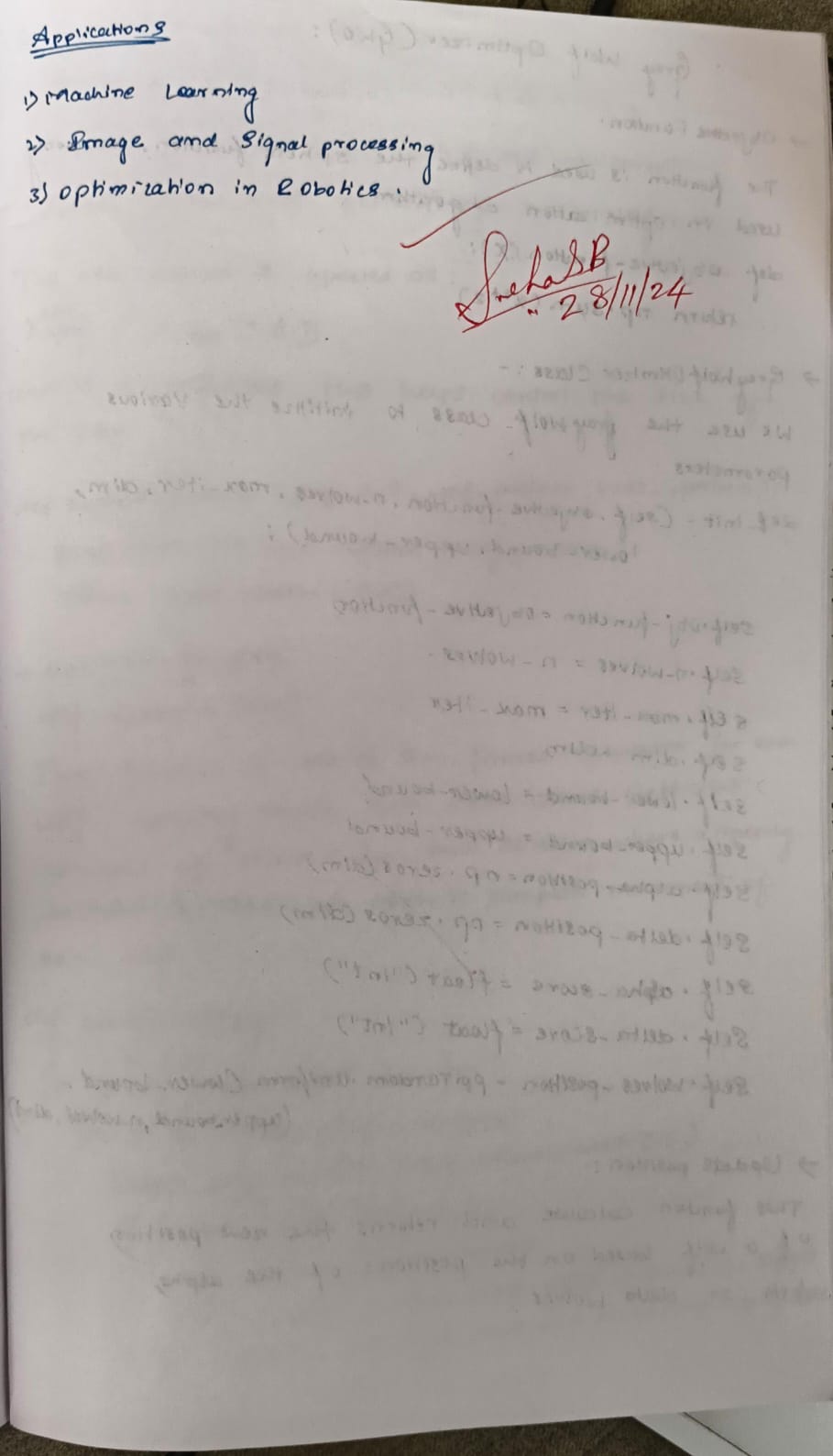
**Algorithm:**

****

****

****

****

****

**Code:**

#cuckoo search

import numpy as np

import random

import math

import matplotlib.pyplot as plt

# Define a sample function to optimize (Sphere function in this case)

def objective\_function(x):

    return np.sum(x \*\* 2)

# Lévy flight function

def levy\_flight(Lambda):

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

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

    sigma\_v = 1

    u = np.random.normal(0, sigma\_u, size=1)

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

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

    return step

# Cuckoo Search algorithm

def cuckoo\_search(num\_nests=25, num\_iterations=100, discovery\_rate=0.25, dim=5, lower\_bound=-10, upper\_bound=10):

    # Initialize nests

    nests = np.random.uniform(lower\_bound, upper\_bound, (num\_nests, dim))

    fitness = np.array([objective\_function(nest) for nest in nests])

    # Get the current best nest

    best\_nest\_idx = np.argmin(fitness)

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

    best\_fitness = fitness[best\_nest\_idx]

    Lambda = 1.5  # Parameter for Lévy flights

    fitness\_history = []  # To track fitness at each iteration

    for iteration in range(num\_iterations):

        # Generate new solutions via Lévy flight

        for i in range(num\_nests):

            step\_size = levy\_flight(Lambda)

            new\_solution = nests[i] + step\_size \* (nests[i] - best\_nest)

            new\_solution = np.clip(new\_solution, lower\_bound, upper\_bound)

            new\_fitness = objective\_function(new\_solution)

            # Replace nest if new solution is better

            if new\_fitness < fitness[i]:

                nests[i] = new\_solution

                fitness[i] = new\_fitness

        # Discover some nests with probability 'discovery\_rate'

        random\_nests = np.random.choice(num\_nests, int(discovery\_rate \* num\_nests), replace=False)

        for nest\_idx in random\_nests:

            nests[nest\_idx] = np.random.uniform(lower\_bound, upper\_bound, dim)

            fitness[nest\_idx] = objective\_function(nests[nest\_idx])

        # Update the best nest

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

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

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

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

        # Store fitness for plotting

        fitness\_history.append(best\_fitness)

        # Print the best solution at each iteration (optional)

        print(f"Iteration {iteration+1}/{num\_iterations}, Best Fitness: {best\_fitness}")

    # Plot fitness convergence graph

    plt.plot(fitness\_history)

    plt.title('Fitness Convergence Over Iterations')

    plt.xlabel('Iteration')

    plt.ylabel('Best Fitness')

    plt.show()

    # Return the best solution found

    return best\_nest, best\_fitness

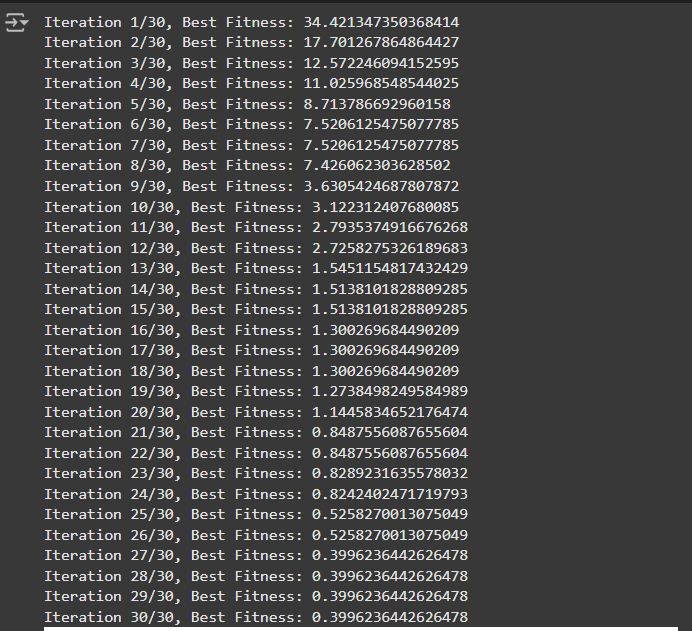
# Example usage

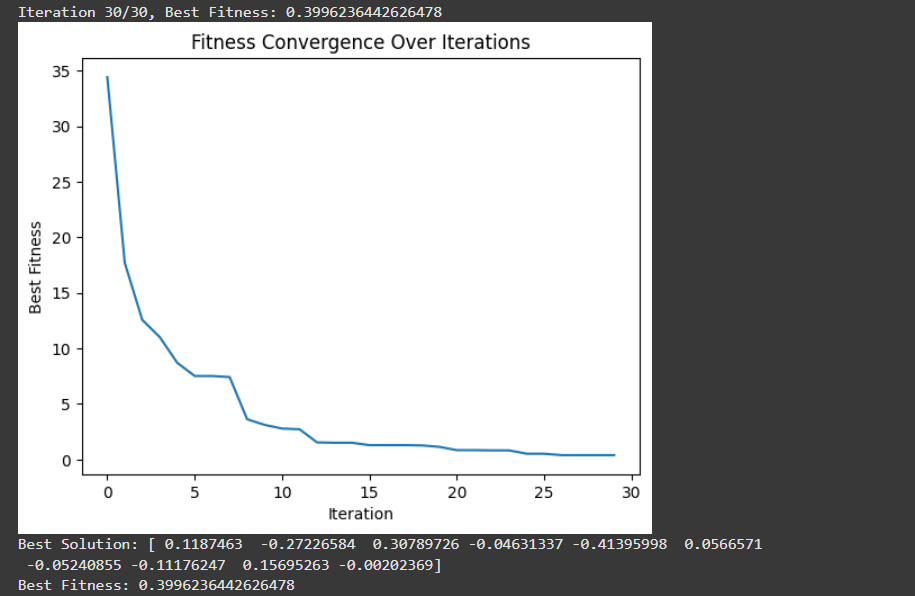
best\_nest, best\_fitness = cuckoo\_search(num\_nests=30, num\_iterations=30, dim=10, lower\_bound=-5, upper\_bound=5)

print("Best Solution:", best\_nest)

print("Best Fitness:", best\_fitness)

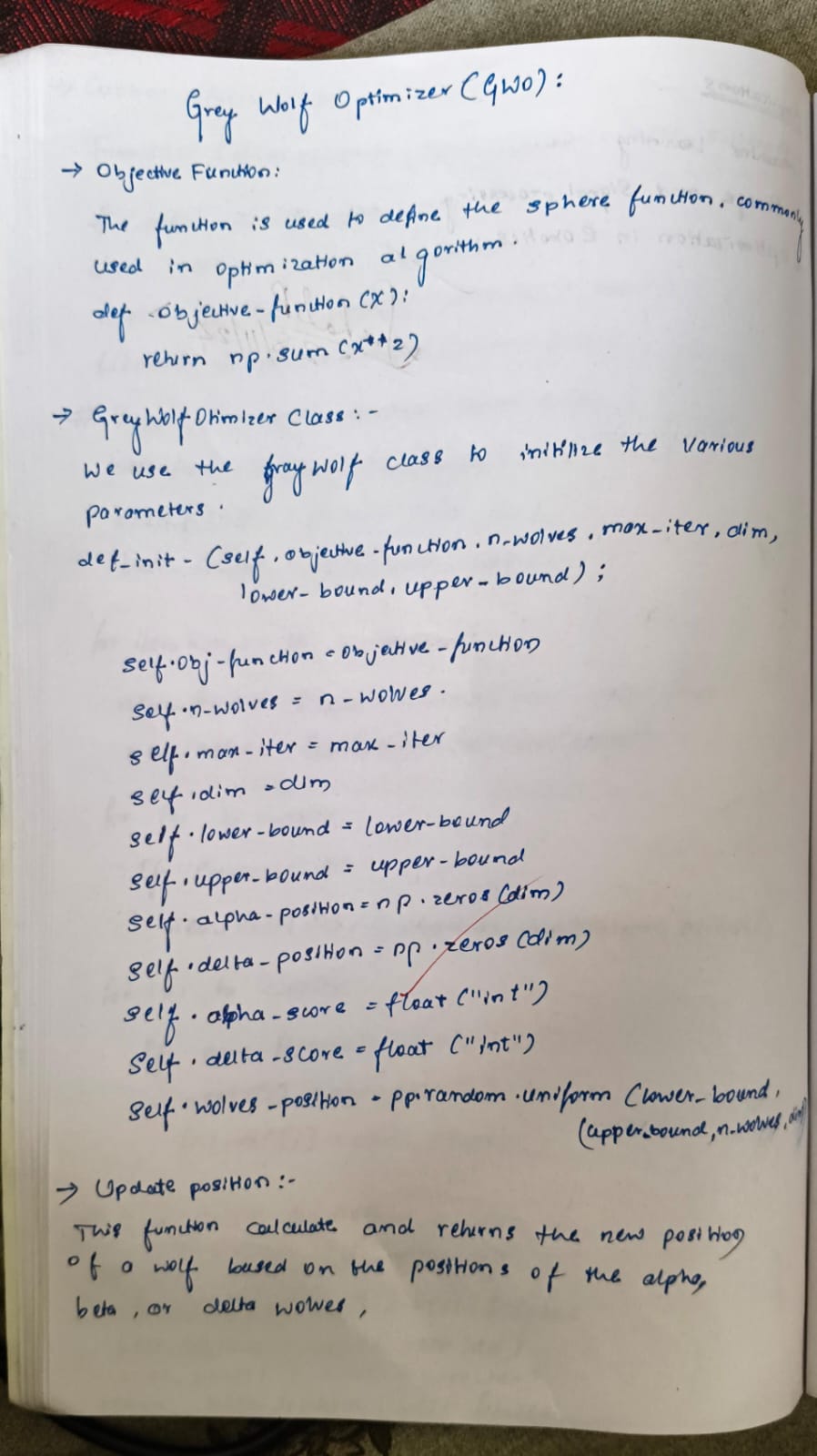
OUTPUT:

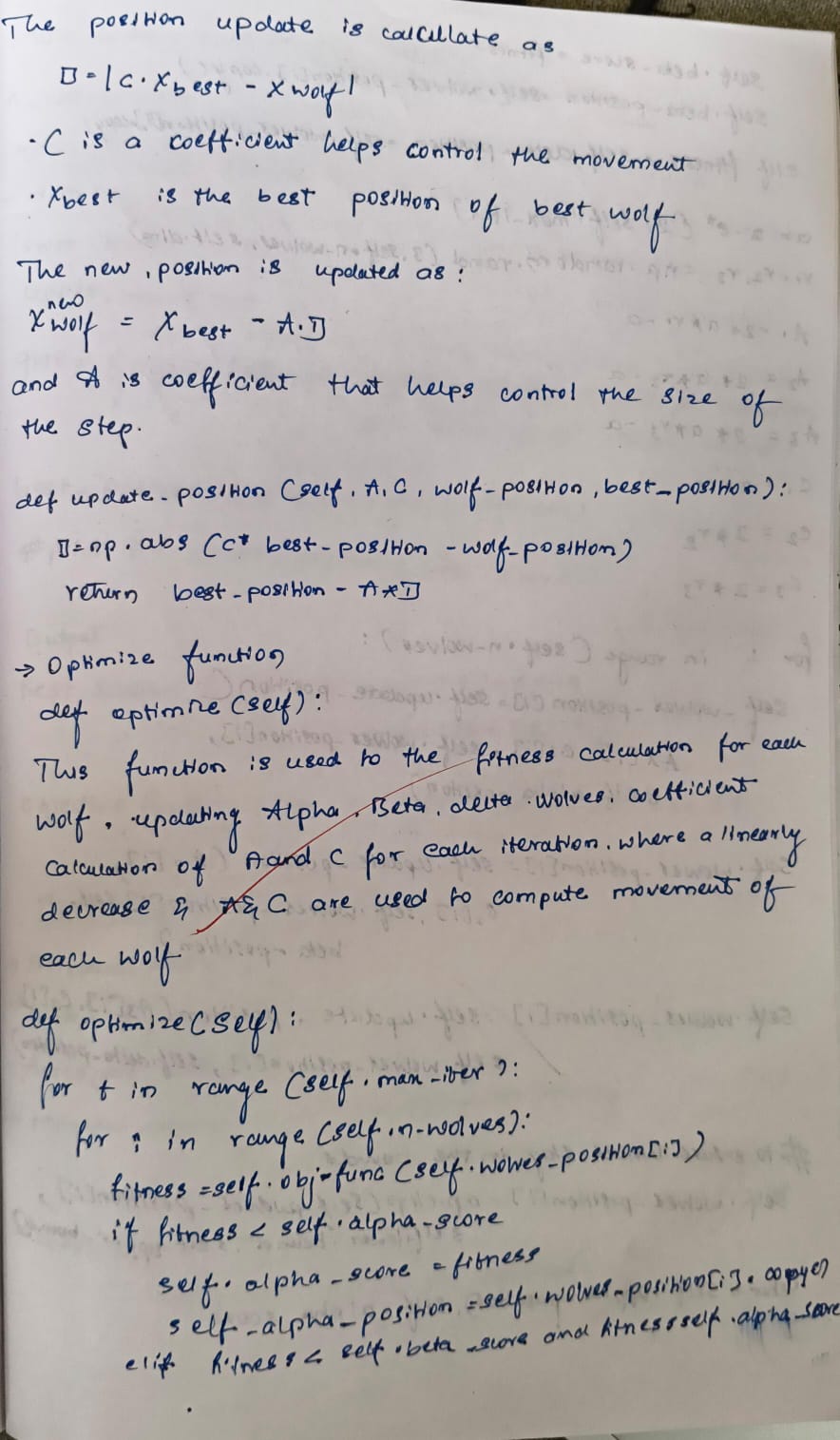


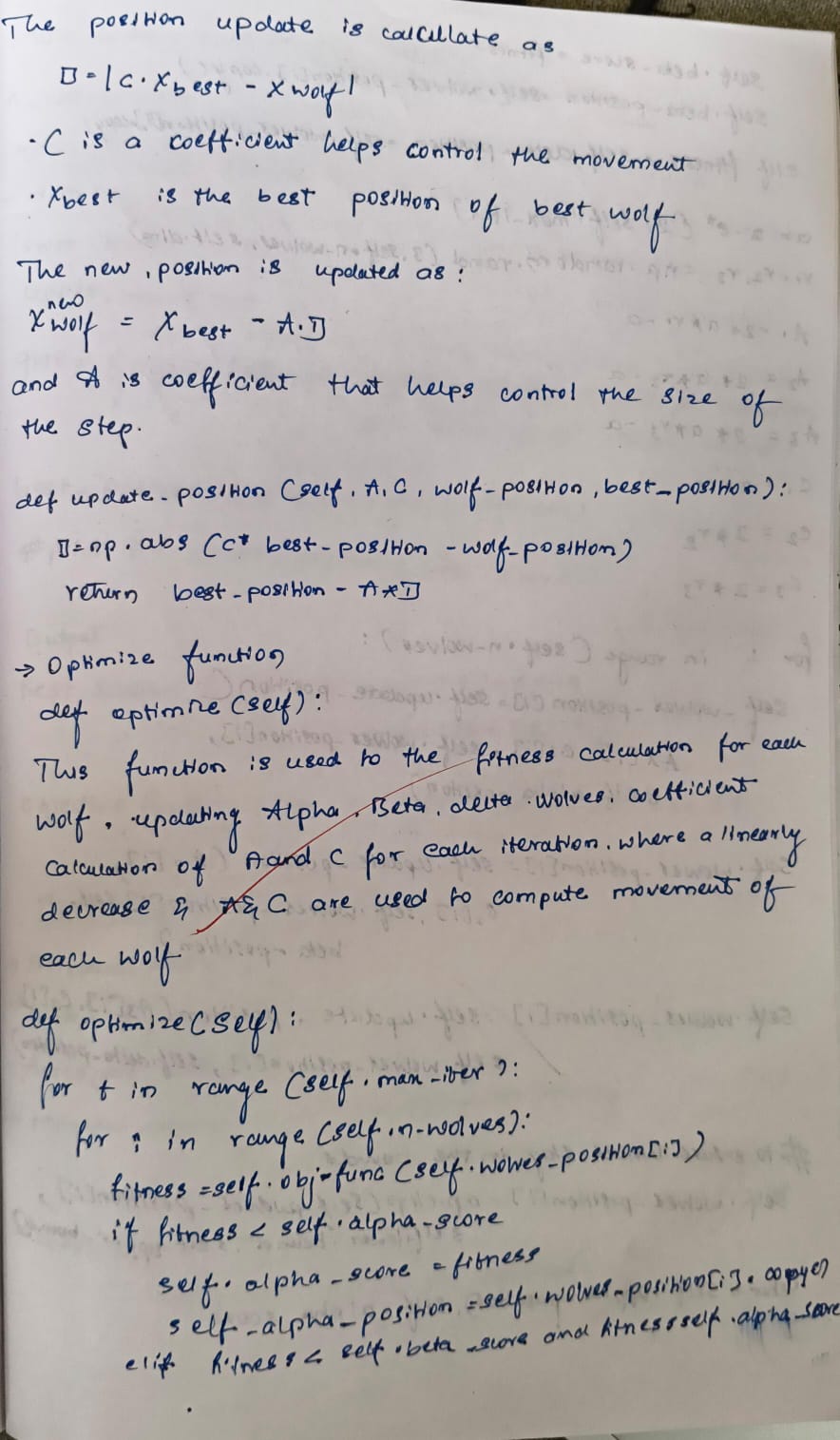


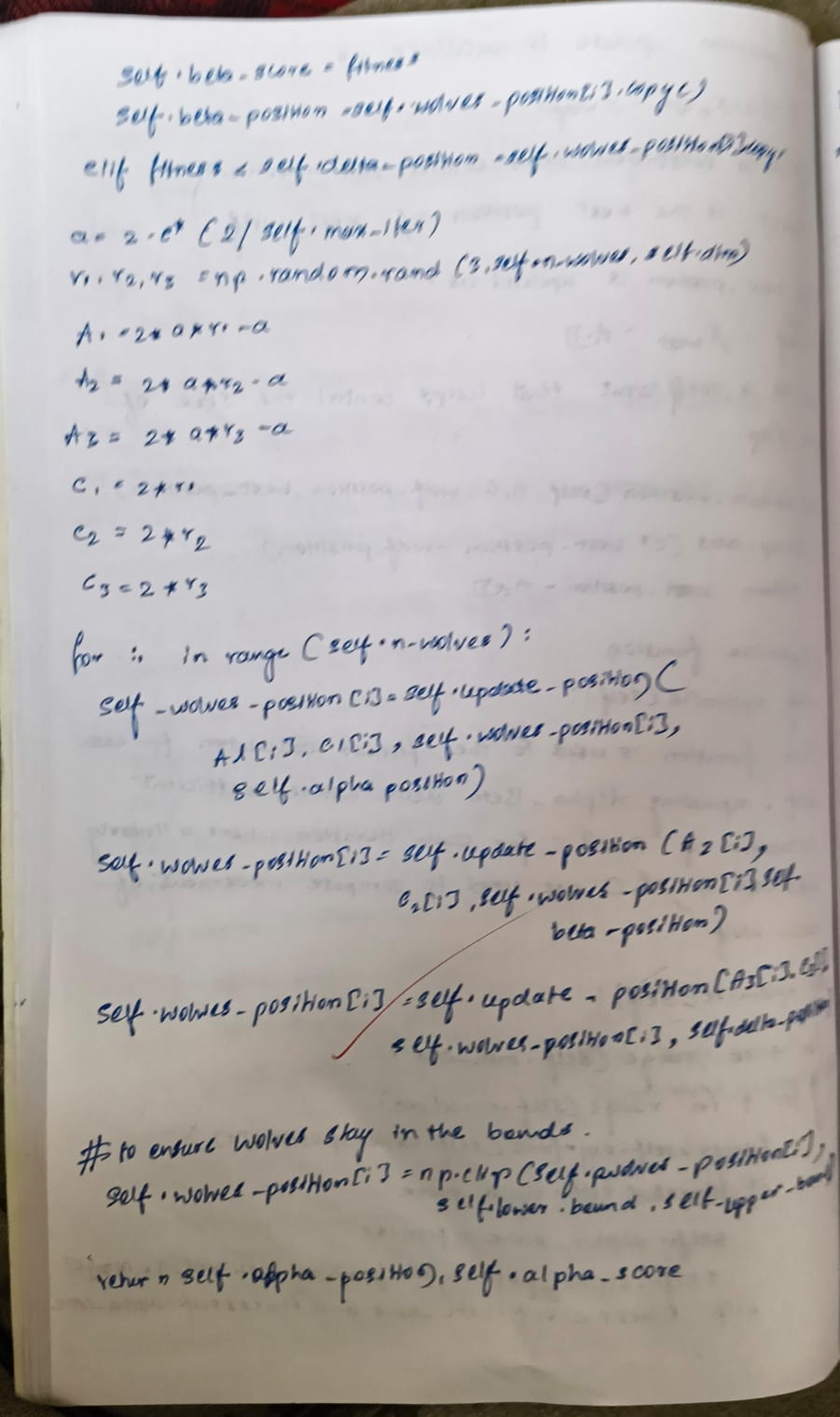
**Program 5: Grey Wolf Optimizer (GWO)**

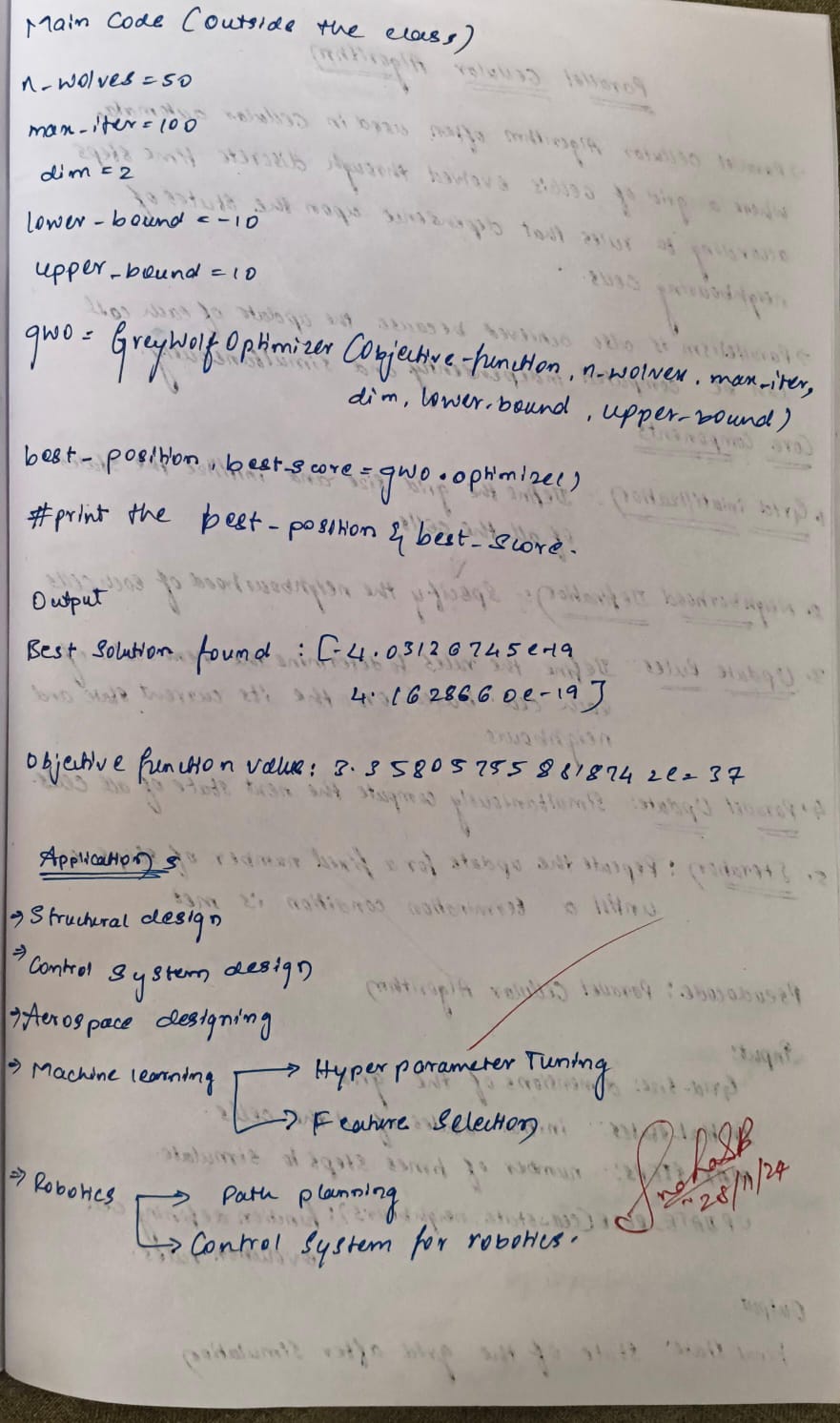
**Algorithm:**

****

****

****

****

****

**Code:**

#GWO

import numpy as np

import matplotlib.pyplot as plt

# Step 1: Define the Problem (a mathematical function to optimize)

def objective\_function(x):

    return np.sum(x\*\*2)  # Example: Sphere function (minimize sum of squares)

# Step 2: Initialize Parameters

num\_wolves = 5  # Number of wolves in the pack

num\_dimensions = 2  # Number of dimensions (for the optimization problem)

num\_iterations = 30  # Number of iterations

lb = -10  # Lower bound of search space

ub = 10  # Upper bound of search space

# Step 3: Initialize Population (Generate initial positions randomly)

wolves = np.random.uniform(lb, ub, (num\_wolves, num\_dimensions))

# Initialize alpha, beta, delta wolves

alpha\_pos = np.zeros(num\_dimensions)

beta\_pos = np.zeros(num\_dimensions)

delta\_pos = np.zeros(num\_dimensions)

alpha\_score = float('inf')  # Best (alpha) score

beta\_score = float('inf')   # Second best (beta) score

delta\_score = float('inf')  # Third best (delta) score

# To store the alpha score over iterations for graphing

alpha\_score\_history = []

# Step 4: Evaluate Fitness and assign Alpha, Beta, Delta wolves

def evaluate\_fitness():

    global alpha\_pos, beta\_pos, delta\_pos, alpha\_score, beta\_score, delta\_score

    for wolf in wolves:

        fitness = objective\_function(wolf)

        # Update Alpha, Beta, Delta wolves based on fitness

        if fitness < alpha\_score:

            delta\_score = beta\_score

            delta\_pos = beta\_pos.copy()

            beta\_score = alpha\_score

            beta\_pos = alpha\_pos.copy()

            alpha\_score = fitness

            alpha\_pos = wolf.copy()

        elif fitness < beta\_score:

            delta\_score = beta\_score

            delta\_pos = beta\_pos.copy()

            beta\_score = fitness

            beta\_pos = wolf.copy()

        elif fitness < delta\_score:

            delta\_score = fitness

            delta\_pos = wolf.copy()

# Step 5: Update Positions

def update\_positions(iteration):

    a = 2 - iteration \* (2 / num\_iterations)  # a decreases linearly from 2 to 0

    for i in range(num\_wolves):

        for j in range(num\_dimensions):

            r1 = np.random.random()

            r2 = np.random.random()

            # Position update based on alpha

            A1 = 2 \* a \* r1 - a

            C1 = 2 \* r2

            D\_alpha = abs(C1 \* alpha\_pos[j] - wolves[i, j])

            X1 = alpha\_pos[j] - A1 \* D\_alpha

            # Position update based on beta

            r1 = np.random.random()

            r2 = np.random.random()

            A2 = 2 \* a \* r1 - a

            C2 = 2 \* r2

            D\_beta = abs(C2 \* beta\_pos[j] - wolves[i, j])

            X2 = beta\_pos[j] - A2 \* D\_beta

            # Position update based on delta

            r1 = np.random.random()

            r2 = np.random.random()

            A3 = 2 \* a \* r1 - a

            C3 = 2 \* r2

            D\_delta = abs(C3 \* delta\_pos[j] - wolves[i, j])

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

            # Update wolf position

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

            # Apply boundary constraints

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

# Step 6: Iterate (repeat evaluation and position updating)

for iteration in range(num\_iterations):

    evaluate\_fitness()  # Evaluate fitness of each wolf

    update\_positions(iteration)  # Update positions based on alpha, beta, delta

    # Record the alpha score for this iteration

    alpha\_score\_history.append(alpha\_score)

    # Optional: Print current best score

    print(f"Iteration {iteration+1}/{num\_iterations}, Alpha Score: {alpha\_score}")

# Step 7: Output the Best Solution

print("Best Solution:", alpha\_pos)

print("Best Solution Fitness:", alpha\_score)

# Plotting the convergence graph

plt.plot(alpha\_score\_history)

plt.title('Convergence of Grey Wolf Optimizer')

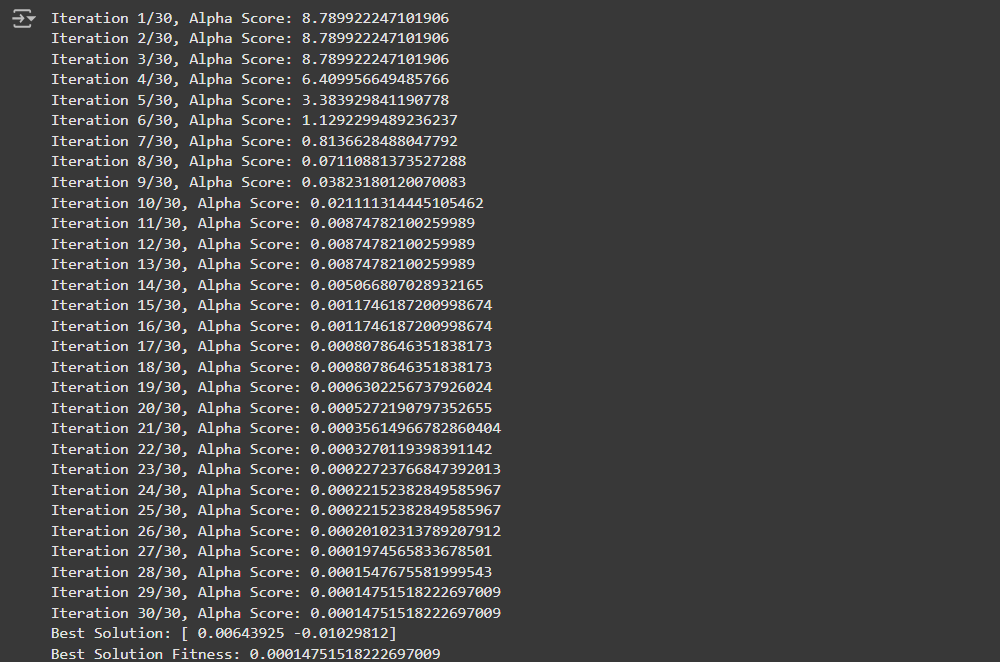
plt.xlabel('Iteration')

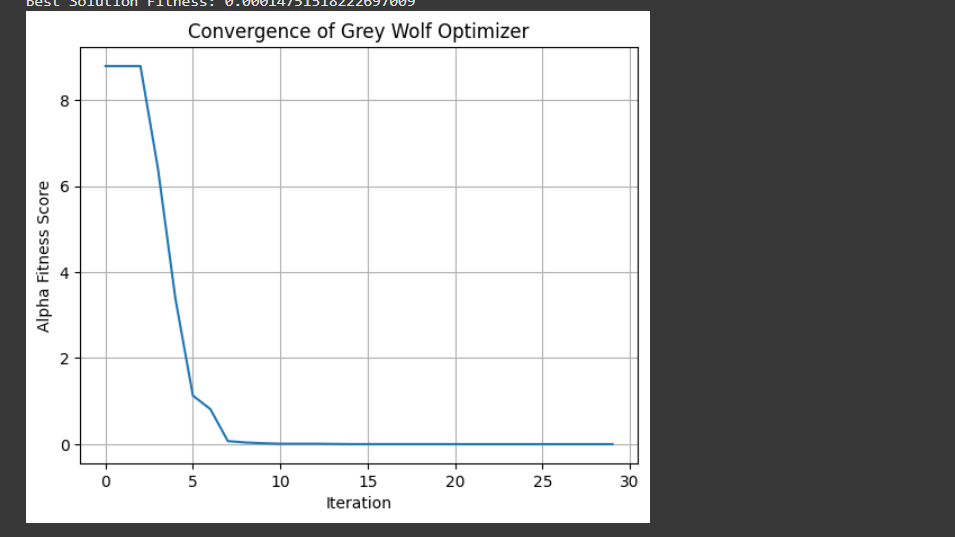
plt.ylabel('Alpha Fitness Score')

plt.grid(True)

plt.show()

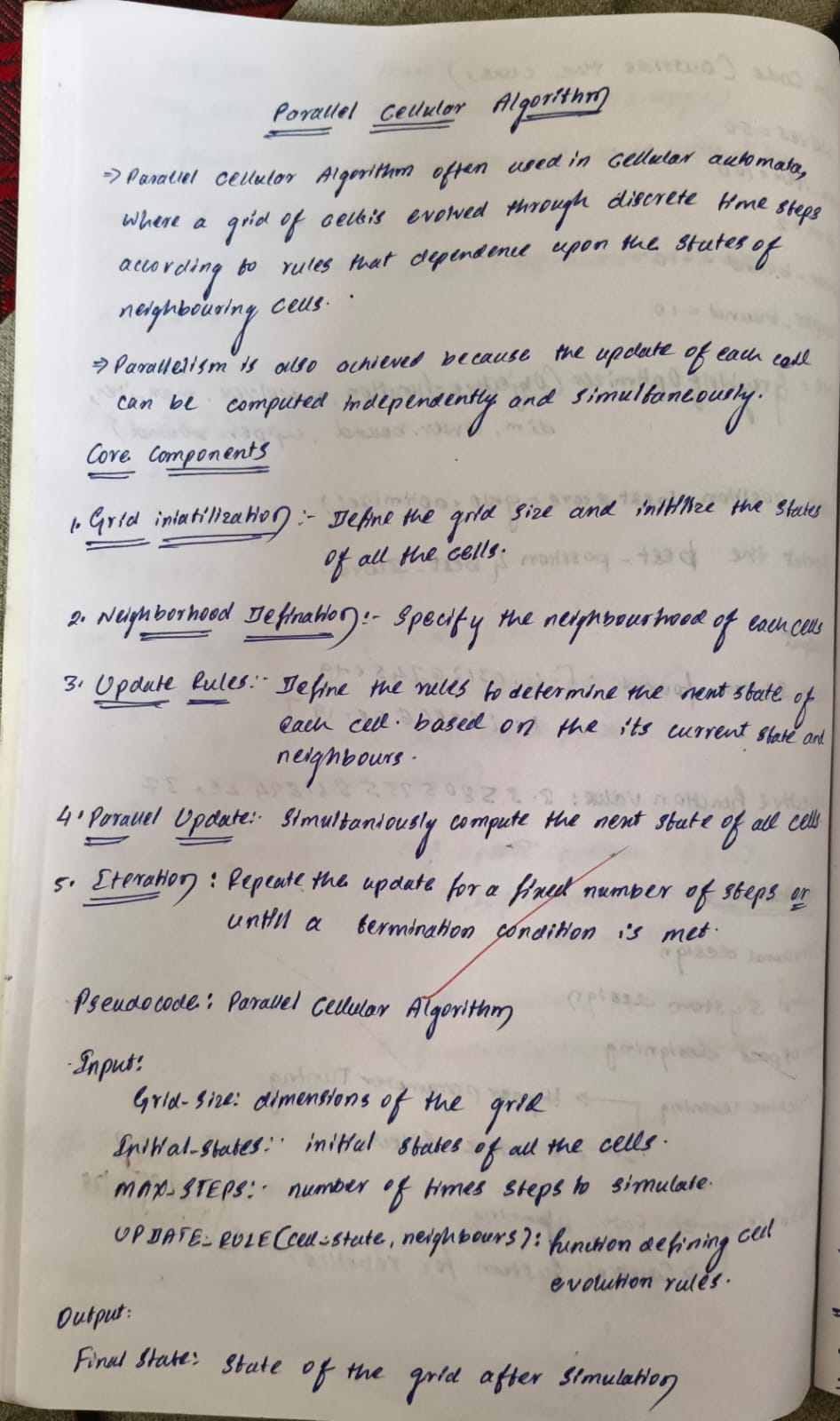
**OUTPUT:**

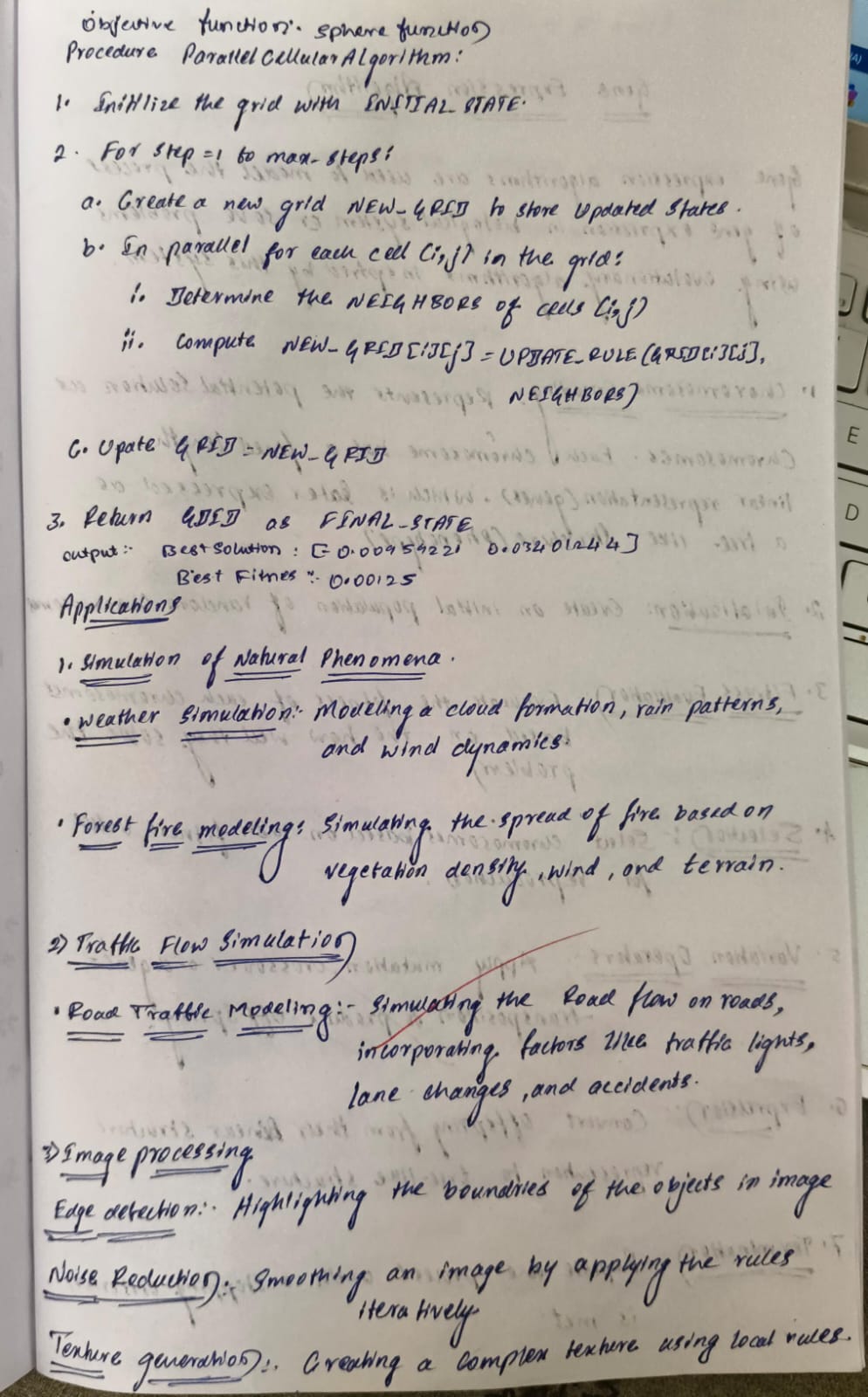




**Program 6: Parallel Cellular Algorithms and Programs**

**Algorithm:**





**Code:**

#pcap

import numpy as np

# Define the problem: A simple optimization function (e.g., Sphere Function)

def optimization\_function(position):

    """Example: Sphere Function for minimization."""

    return sum(x\*\*2 for x in position)

# Initialize Parameters

GRID\_SIZE = (10, 10)  # Grid size (rows, columns)

NEIGHBORHOOD\_RADIUS = 1  # Moore neighborhood radius

DIMENSIONS = 2  # Number of dimensions in the solution space

ITERATIONS = 30  # Number of iterations

# Initialize Population

def initialize\_population(grid\_size, dimensions):

    """Initialize a grid with random positions."""

    population = np.random.uniform(-10, 10, size=(grid\_size[0], grid\_size[1], dimensions))

    return population

# Evaluate Fitness

def evaluate\_fitness(population):

    """Calculate the fitness of all cells."""

    fitness = np.zeros((population.shape[0], population.shape[1]))

    for i in range(population.shape[0]):

        for j in range(population.shape[1]):

            fitness[i, j] = optimization\_function(population[i, j])

    return fitness

# Get Neighborhood

def get\_neighborhood(grid, x, y, radius):

    """Get the neighbors of a cell within the specified radius."""

    neighbors = []

    for i in range(-radius, radius + 1):

        for j in range(-radius, radius + 1):

            if i == 0 and j == 0:

                continue  # Skip the current cell

            ni, nj = x + i, y + j

            if 0 <= ni < grid.shape[0] and 0 <= nj < grid.shape[1]:

                neighbors.append((ni, nj))

    return neighbors

# Update States

def update\_states(population, fitness):

    """Update the state of each cell based on its neighbors."""

    new\_population = np.copy(population)

    for i in range(population.shape[0]):

        for j in range(population.shape[1]):

            neighbors = get\_neighborhood(population, i, j, NEIGHBORHOOD\_RADIUS)

            best\_neighbor = population[i, j]

            best\_fitness = fitness[i, j]

            # Find the best position among neighbors

            for ni, nj in neighbors:

                if fitness[ni, nj] < best\_fitness:

                    best\_fitness = fitness[ni, nj]

                    best\_neighbor = population[ni, nj]

            # Update the cell state (move towards the best neighbor)

            new\_population[i, j] = (population[i, j] + best\_neighbor) / 2  # Average position

    return new\_population

# Main Algorithm

def parallel\_cellular\_algorithm():

    """Implementation of the Parallel Cellular Algorithm."""

    population = initialize\_population(GRID\_SIZE, DIMENSIONS)

    best\_solution = None

    best\_fitness = float('inf')

    for iteration in range(ITERATIONS):

        # Evaluate fitness

        fitness = evaluate\_fitness(population)

        # Track the best solution

        min\_fitness = np.min(fitness)

        if min\_fitness < best\_fitness:

            best\_fitness = min\_fitness

            best\_solution = population[np.unravel\_index(np.argmin(fitness), fitness.shape)]

        # Update states based on neighbors

        population = update\_states(population, fitness)

        # Print progress

        print(f"Iteration {iteration + 1}: Best Fitness = {best\_fitness}")

    print("\nBest Solution Found:")

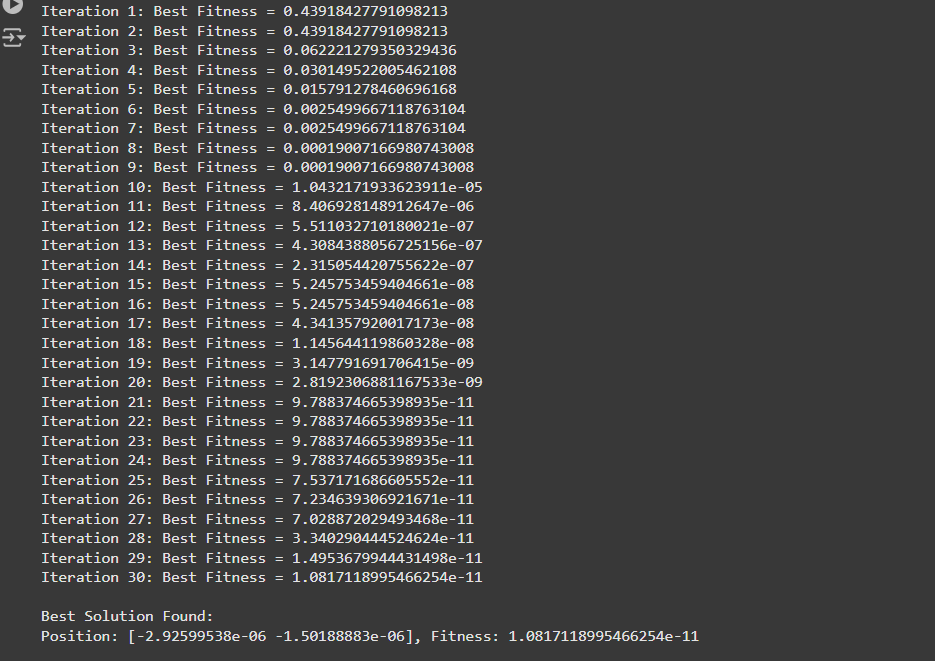
    print(f"Position: {best\_solution}, Fitness: {best\_fitness}")

# Run the algorithm

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

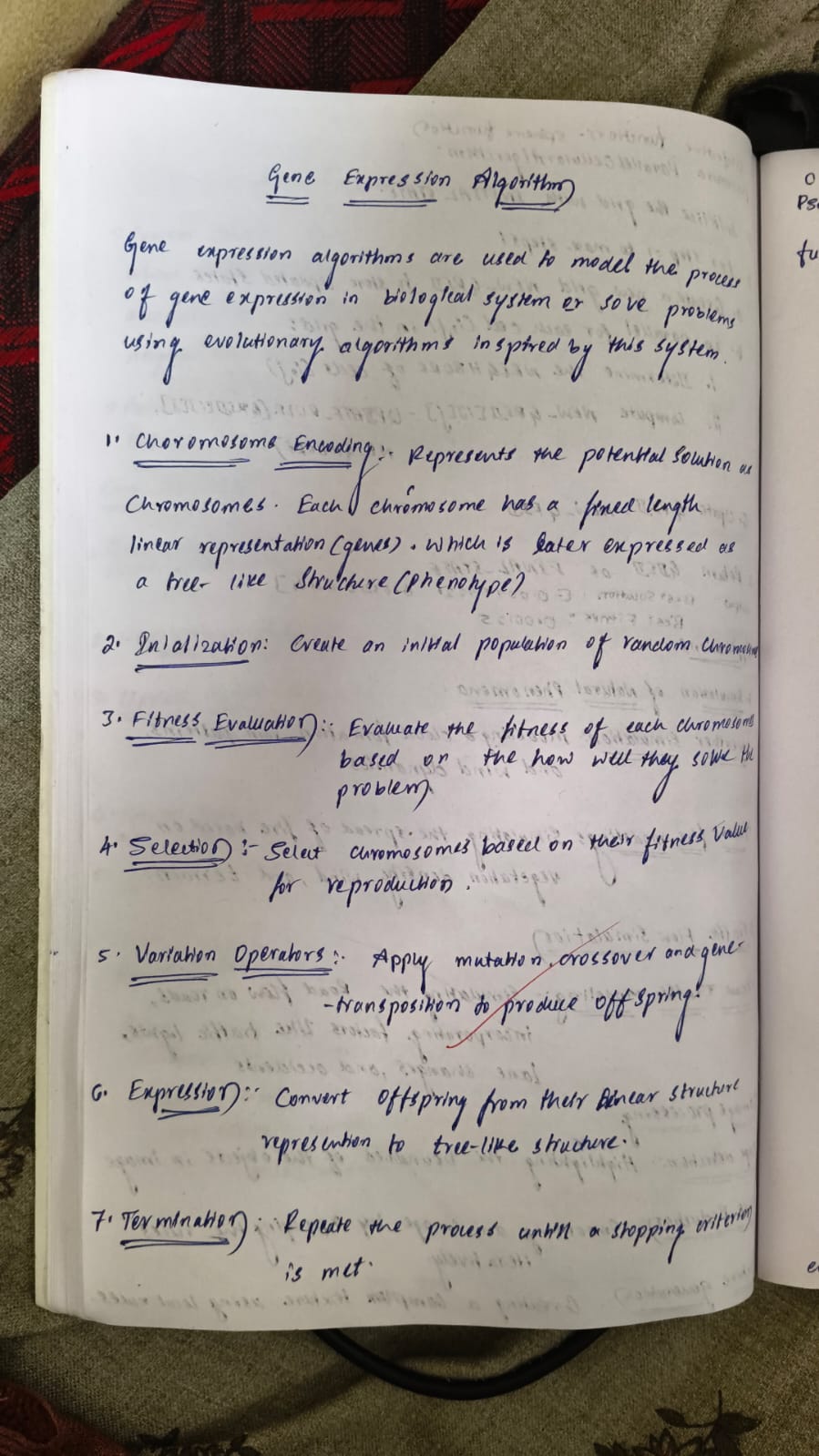
    parallel\_cellular\_algorithm()

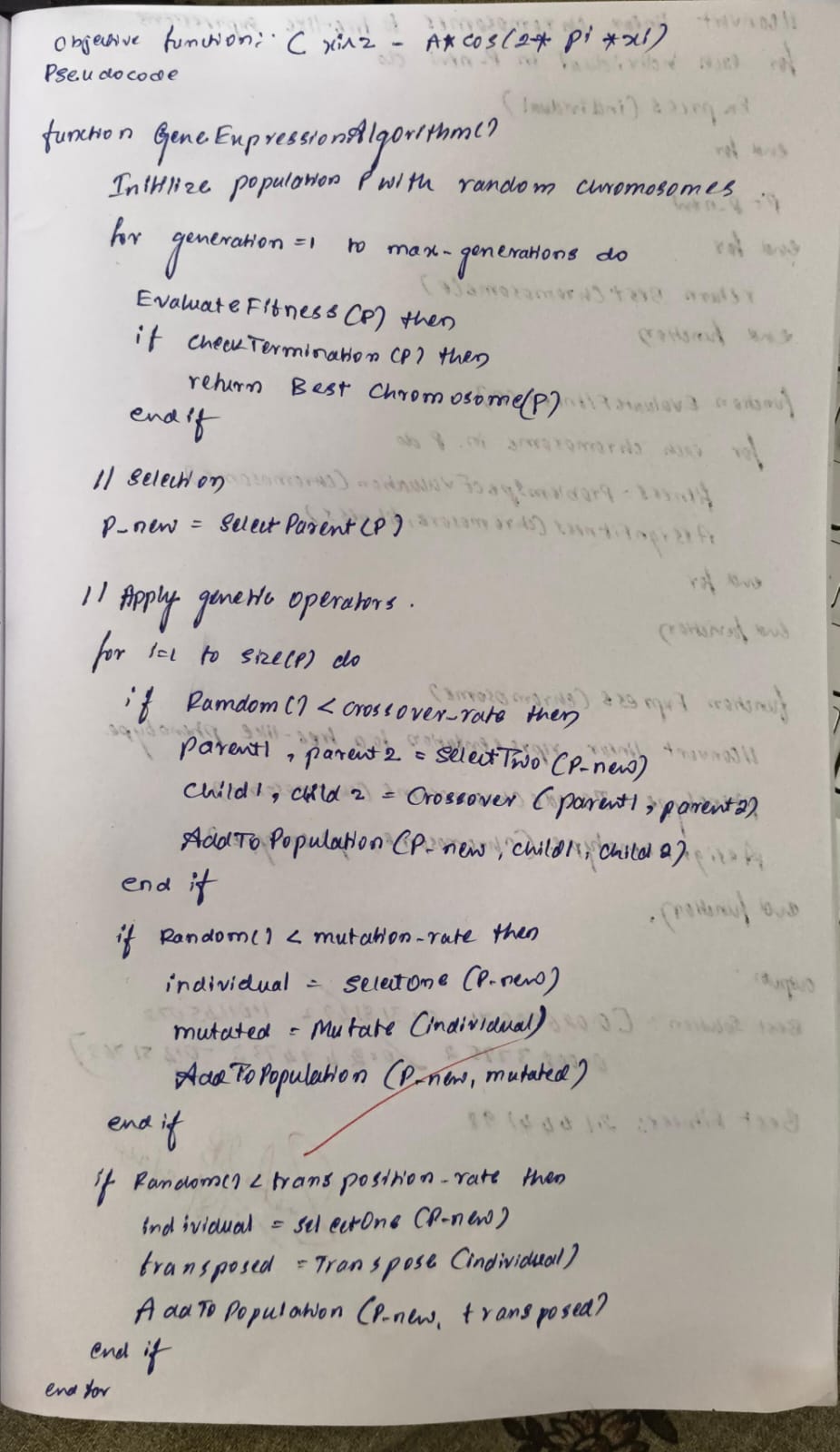
OUTPUT:

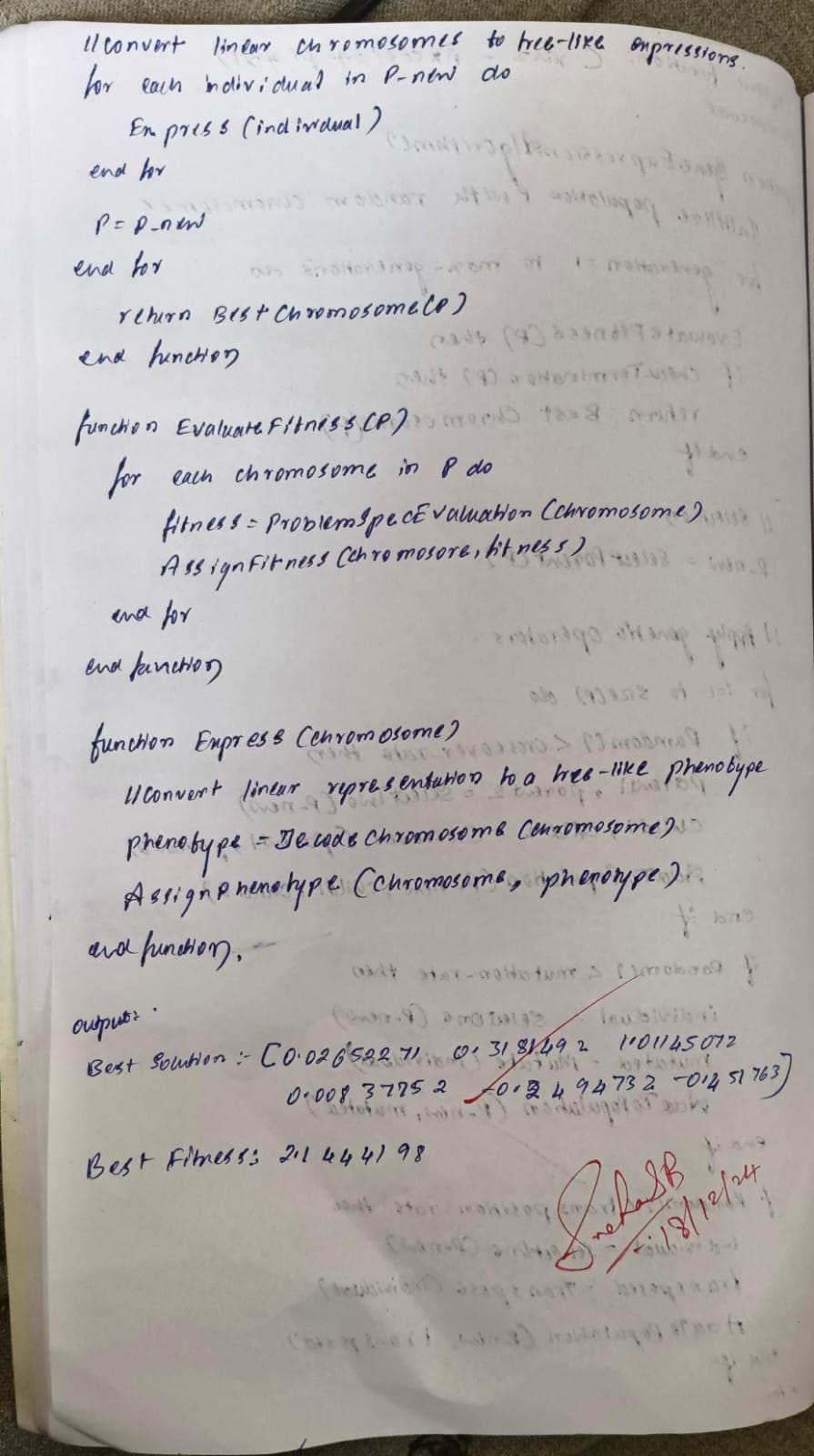


**Program 7: Optimization via Gene Expression Algorithms**

**Algorithm:**





****

**Code:**

import numpy as np

import random

# 1. Define the Problem: Optimization Function (e.g., Sphere Function)

def optimization\_function(solution):

    """Sphere Function for minimization (fitness evaluation)."""

    return sum(x\*\*2 for x in solution)

# 2. Initialize Parameters

POPULATION\_SIZE = 50  # Number of genetic sequences (solutions)

GENES = 5  # Number of genes per solution

MUTATION\_RATE = 0.1  # Probability of mutation

CROSSOVER\_RATE = 0.7  # Probability of crossover

GENERATIONS = 30  # Number of generations to evolve

# 3. Initialize Population

def initialize\_population(pop\_size, genes):

    """Generate initial population of random genetic sequences."""

    return np.random.uniform(-10, 10, (pop\_size, genes))

# 4. Evaluate Fitness

def evaluate\_fitness(population):

    """Evaluate the fitness of each genetic sequence."""

    fitness = [optimization\_function(solution) for solution in population]

    return np.array(fitness)

# 5. Selection: Tournament Selection

def select\_parents(population, fitness, num\_parents):

    """Select parents using tournament selection."""

    parents = []

    for \_ in range(num\_parents):

        tournament = random.sample(range(len(population)), 3)  # Randomly select 3 candidates

        best = min(tournament, key=lambda idx: fitness[idx])

        parents.append(population[best])

    return np.array(parents)

# 6. Crossover: Single-Point Crossover

def crossover(parents, crossover\_rate):

    """Perform crossover between pairs of parents."""

    offspring = []

    for i in range(0, len(parents), 2):

        if i + 1 >= len(parents):

            break

        parent1, parent2 = parents[i], parents[i + 1]

        if random.random() < crossover\_rate:

            point = random.randint(1, len(parent1) - 1)  # Single crossover point

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

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

        else:

            child1, child2 = parent1, parent2  # No crossover

        offspring.extend([child1, child2])

    return np.array(offspring)

# 7. Mutation

def mutate(offspring, mutation\_rate):

    """Apply mutation to introduce variability."""

    for i in range(len(offspring)):

        for j in range(len(offspring[i])):

            if random.random() < mutation\_rate:

                offspring[i][j] += np.random.uniform(-1, 1)  # Random small change

    return offspring

# 8. Gene Expression: Functional Solution (No transformation needed for this case)

def gene\_expression(population):

    """Translate genetic sequences into functional solutions."""

    return population  # Genetic sequences directly represent solutions here.

# 9. Main Function: Gene Expression Algorithm

def gene\_expression\_algorithm():

    """Implementation of Gene Expression Algorithm for optimization."""

    # Initialize population

    population = initialize\_population(POPULATION\_SIZE, GENES)

    best\_solution = None

    best\_fitness = float('inf')

    for generation in range(GENERATIONS):

        # Evaluate fitness

        fitness = evaluate\_fitness(population)

        # Track the best solution

        min\_fitness\_idx = np.argmin(fitness)

        if fitness[min\_fitness\_idx] < best\_fitness:

            best\_fitness = fitness[min\_fitness\_idx]

            best\_solution = population[min\_fitness\_idx]

        # Selection

        parents = select\_parents(population, fitness, POPULATION\_SIZE // 2)

        # Crossover

        offspring = crossover(parents, CROSSOVER\_RATE)

        # Mutation

        offspring = mutate(offspring, MUTATION\_RATE)

        # Gene Expression

        population = gene\_expression(offspring)

        # Print progress

        print(f"Generation {generation + 1}: Best Fitness = {best\_fitness}")

    # Output the best solution

    print("\nBest Solution Found:")

    print(f"Position: {best\_solution}, Fitness: {best\_fitness}")

# 10. Run the Algorithm

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

    gene\_expression\_algorithm()

OUTPUT:

