|  |
| --- |
| **VISVESVARAYA TECHNOLOGICAL UNIVERSITY**  **“JnanaSangama”, Belgaum -590014, Karnataka.**    **LAB RECORD**  **Bio Inspired Systems (23CS5BSBIS)**  ***Submitted by***  **Anish Budavi (1BM23CS401)**  ***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 **Anish Arjun Budavi (1BM23CS401),** 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.

|  |  |
| --- | --- |
| Prof. Surabhi S  Assistant Professor  Department of CSE, BMSCE | Dr. Kavitha Sooda  Professor & HOD  Department of CSE, BMSCE |

**Index**

|  |  |  |  |
| --- | --- | --- | --- |
| **Sl.**  **No.** | **Date** | **Experiment Title** | **Page No.** |
| 1 | 9-10-24 | Genetic Algorithm | 1-6 |
| 2 | 16-10-24 | Ant Colony Optimization | 7-16 |
| 3 | 30-10-24 | Particle Swarm Optimization | 17-21 |
| 4 | 13-11-24 | Cuckoo Search Algorithm | 22-27 |
| 5 | 20-11-24 | Grey Wolf Optimizer | 28-33 |
| 6 | 27-11-24 | Parallel Cellular Algorithm | 34-37 |
| 7 | 4-12-24 | Gene Expression Algorithm | 38-42 |

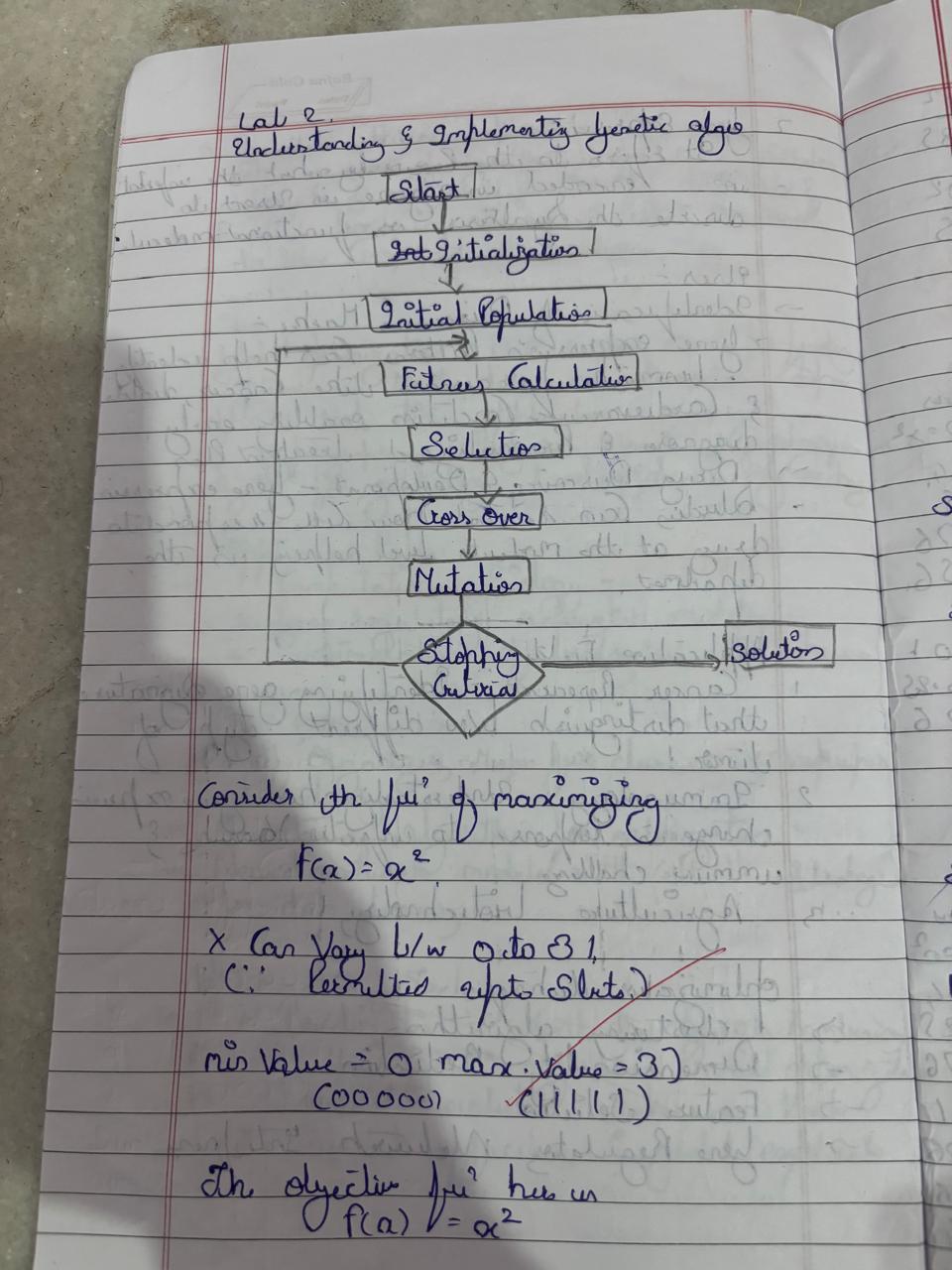
**Github Link:**

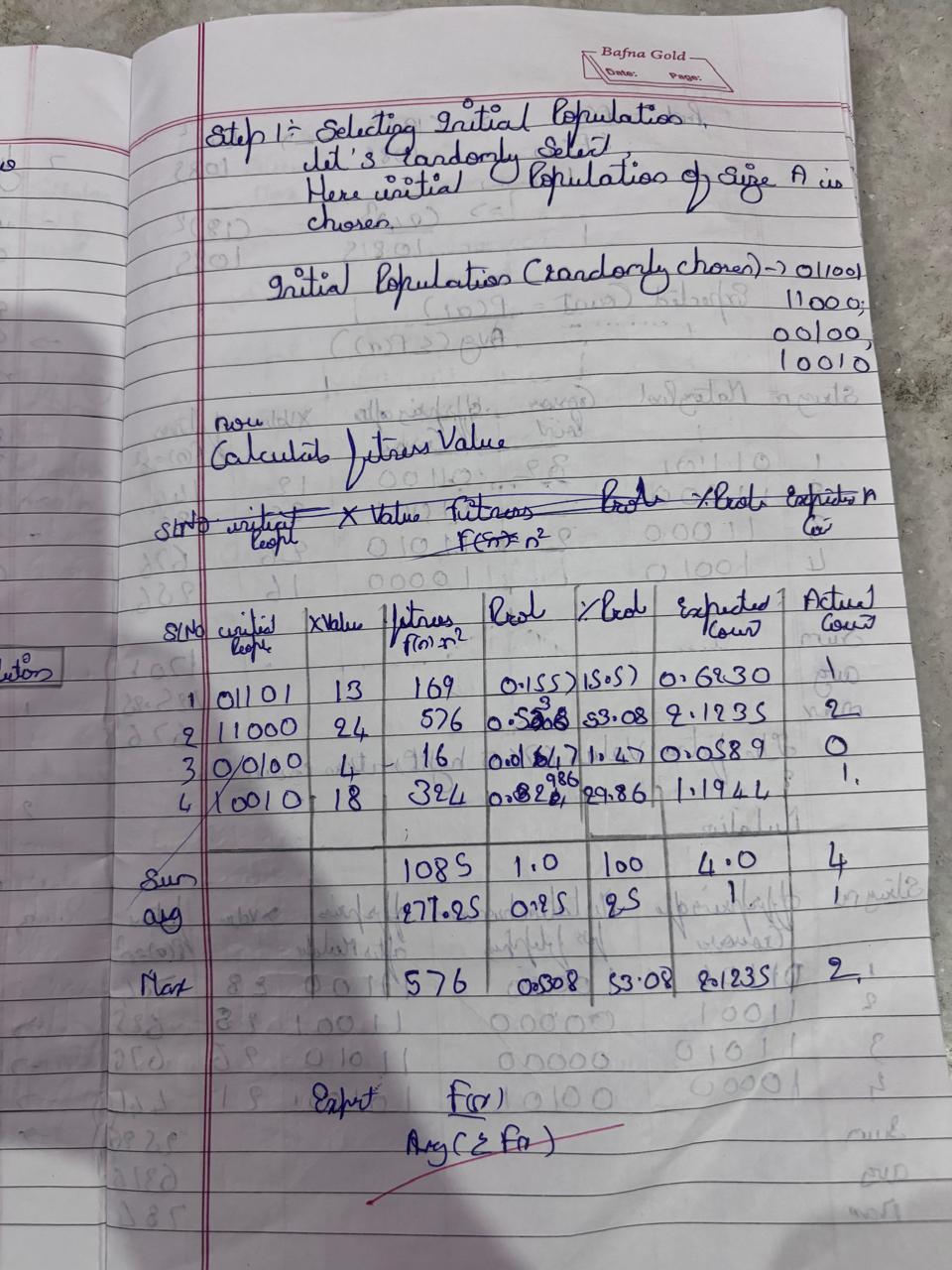
https://github.com/Anishbudavi/ANISH-BUDAVI-1BM23CS401-BIS-LAB.git

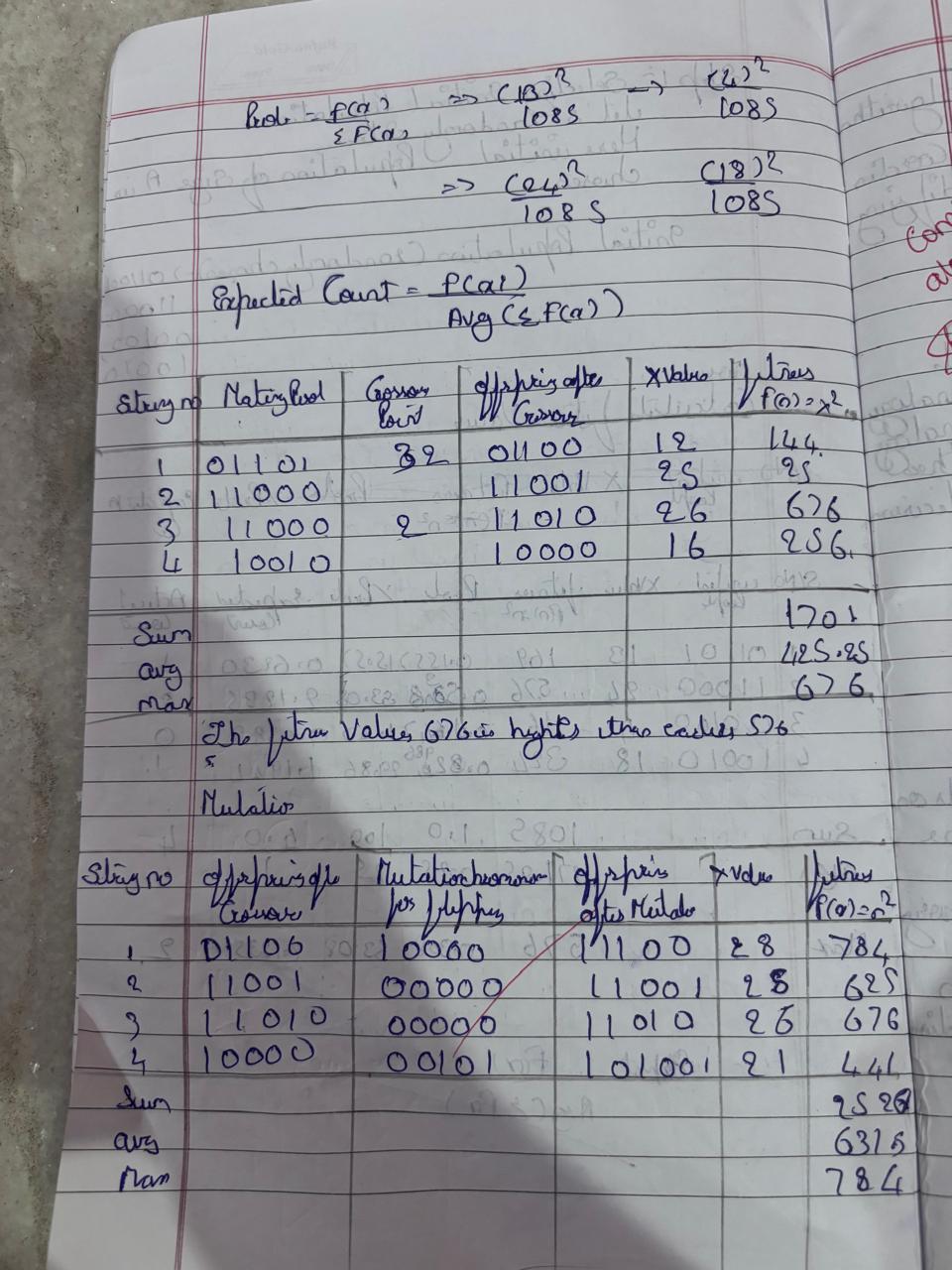
**Program 1**

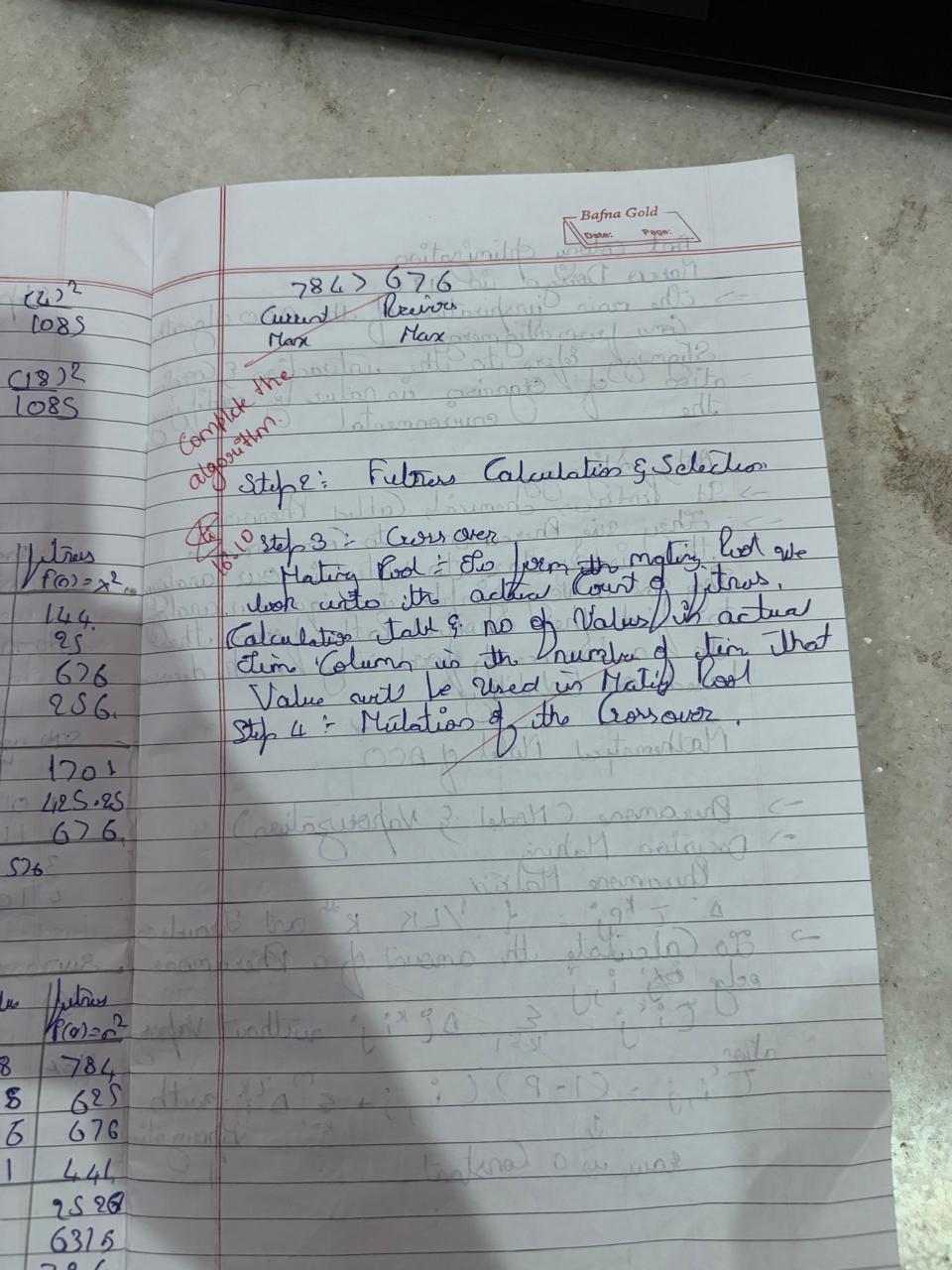
**Genetic Algorithm for Optimization Problems**

**Algorithm:**

****

****





**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]

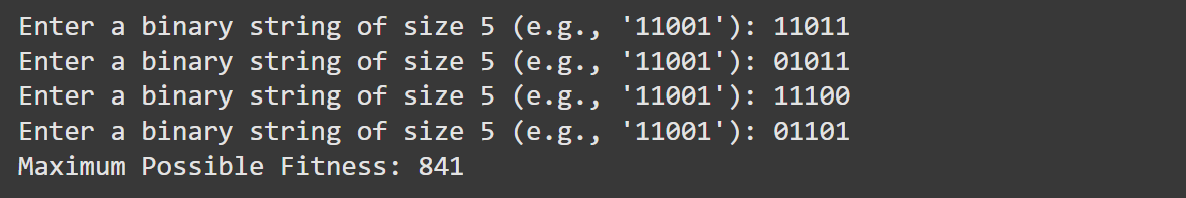
# Get the maximum fitness

fitnesses = [fitness(chromosome) for chromosome in population]

max\_fitness = max(fitnesses)

print(f"Maximum Possible Fitness: {max\_fitness}")

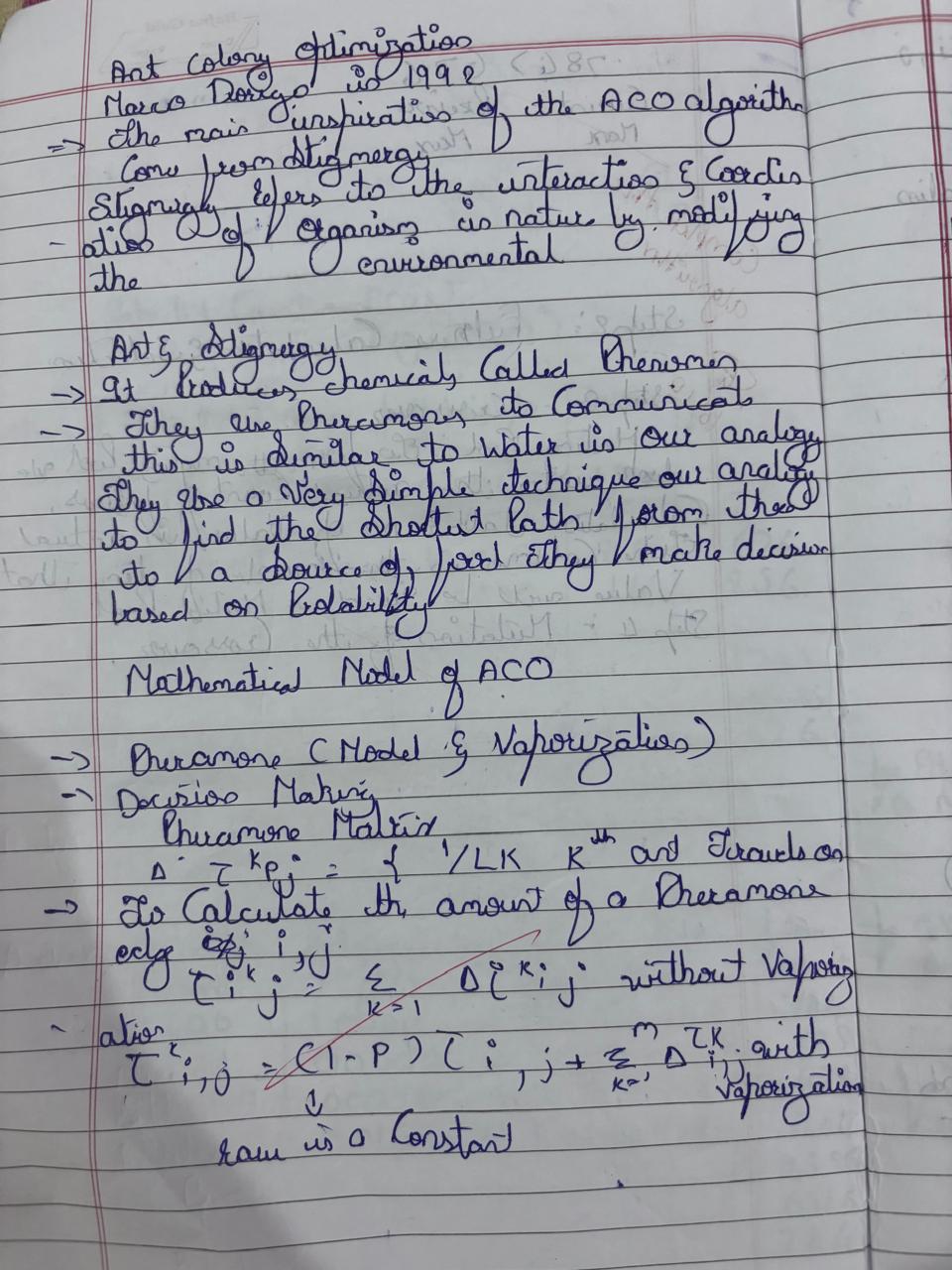
**Output:**

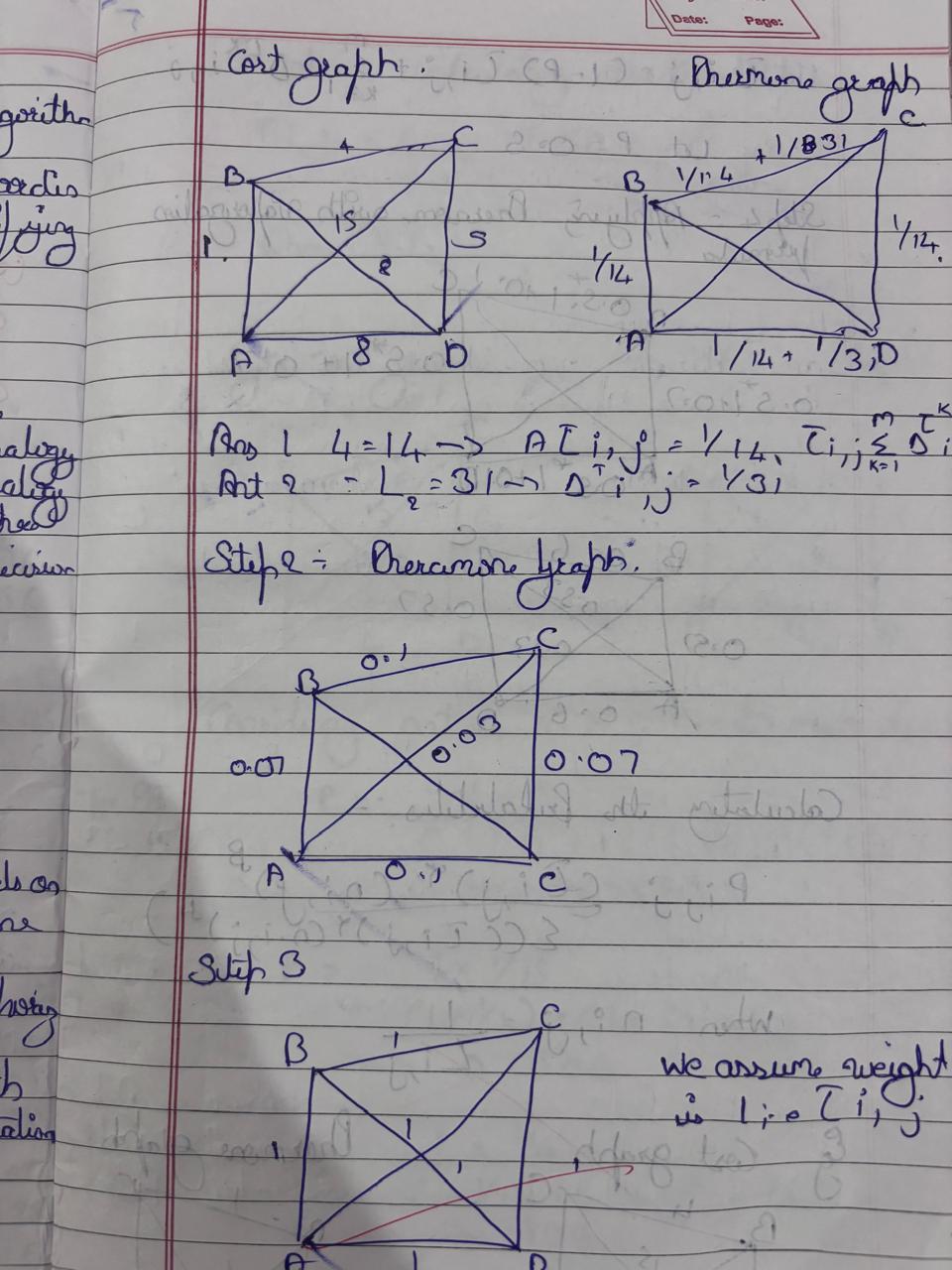


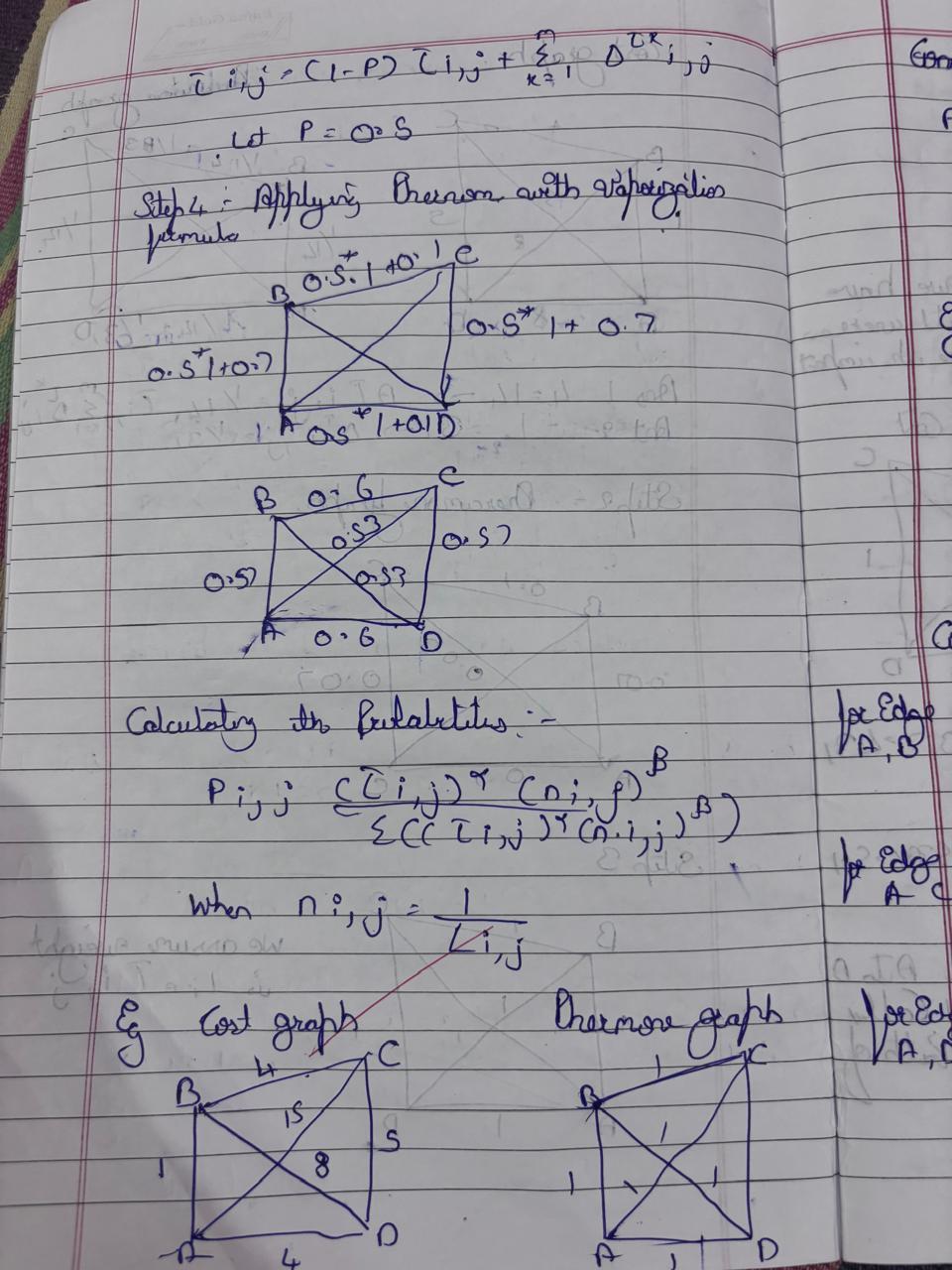
**Program 2**

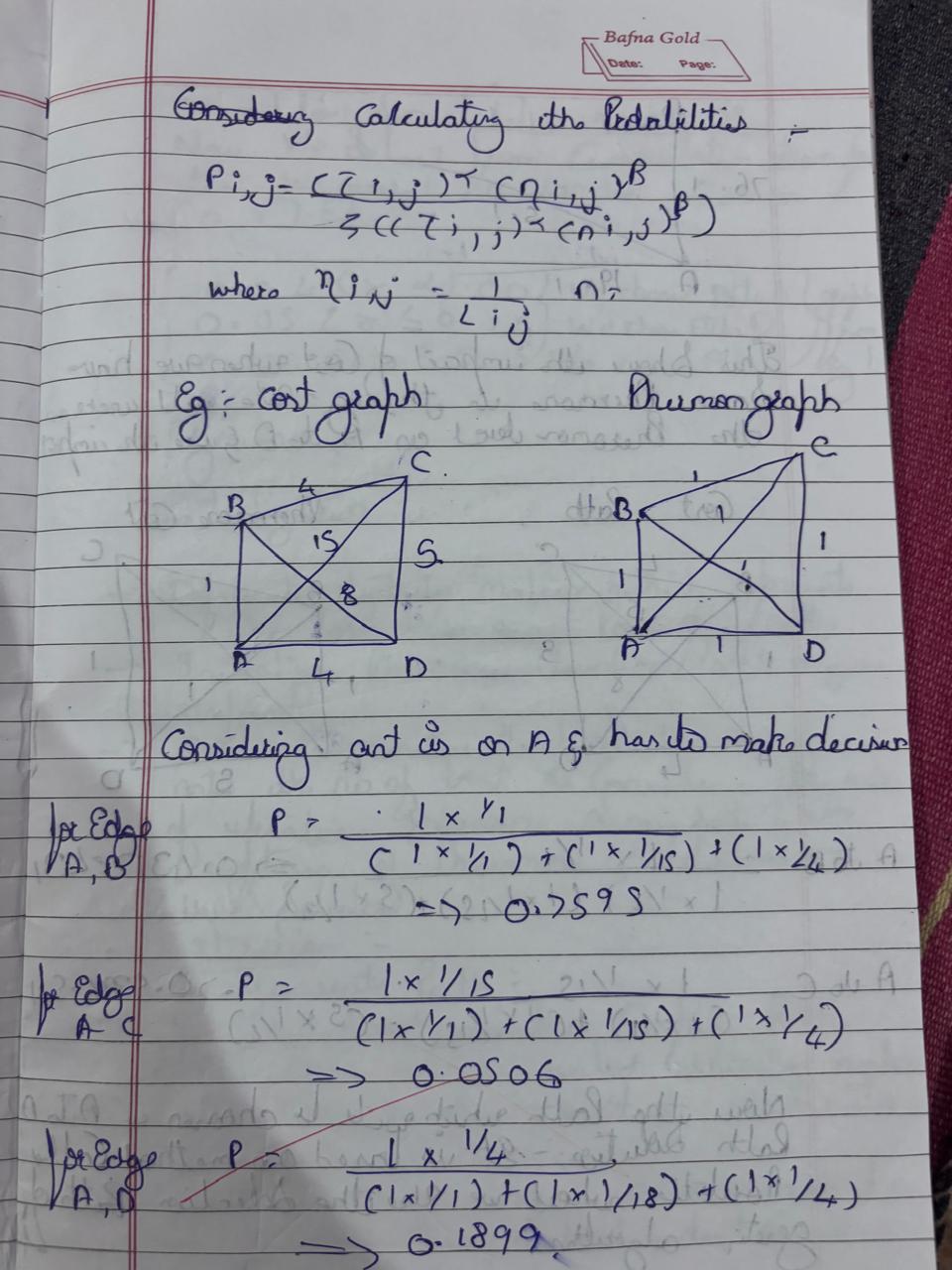
**Ant Colony Optimization**

**Algorithm:**

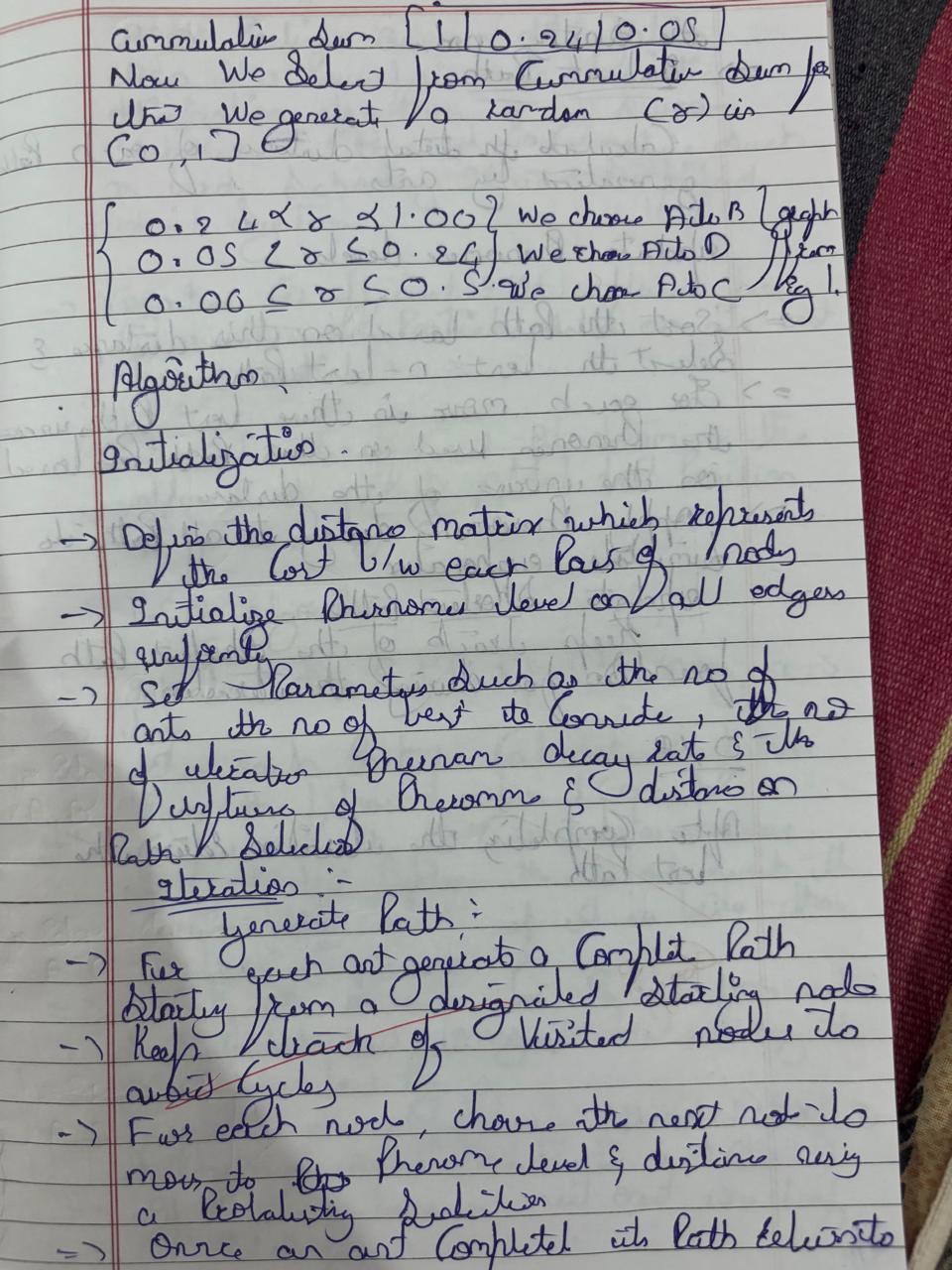
****

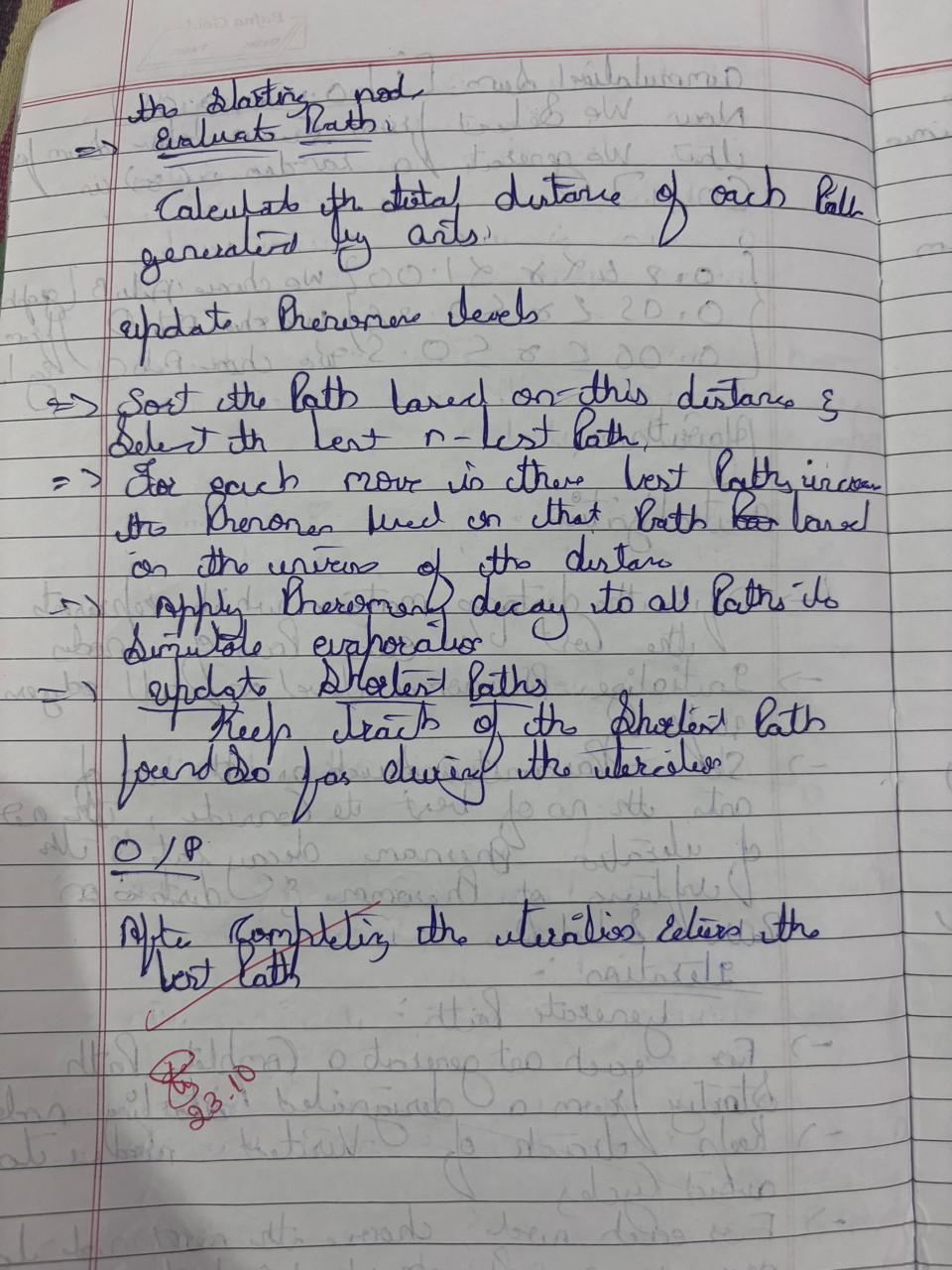
****

****

****

****

****

****

**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],

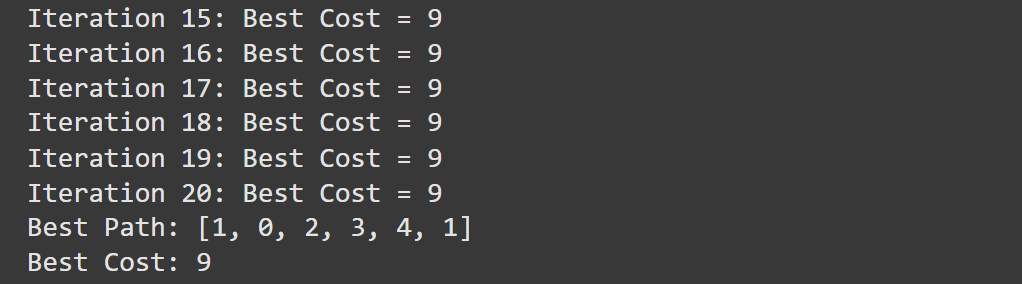
[5, 8, 1, 0, 2],

[7, 2, 3, 2, 0]

]

ant\_colony\_optimization(cost\_matrix, n\_ants=5, n\_iterations=20)

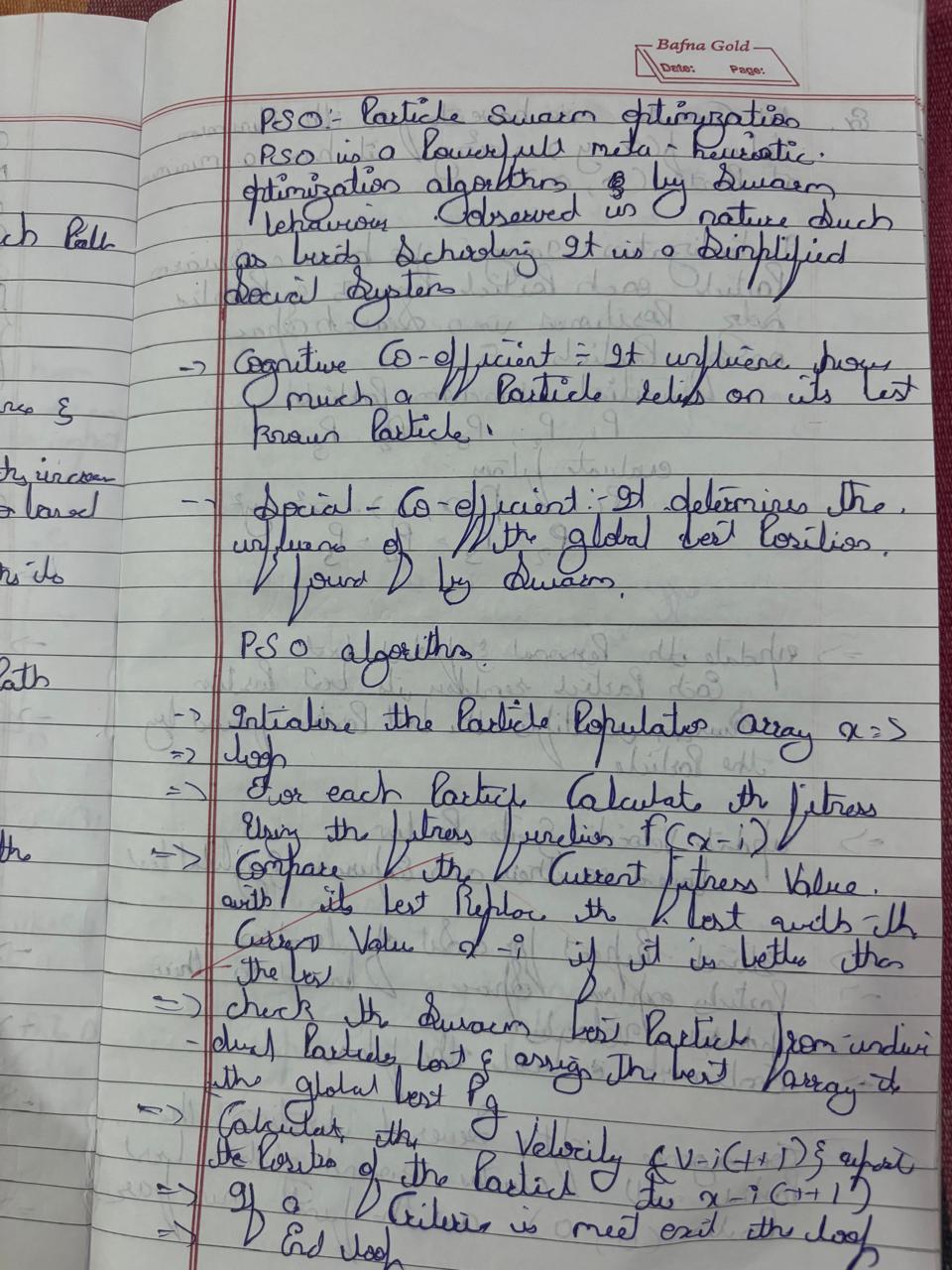
**Output:**

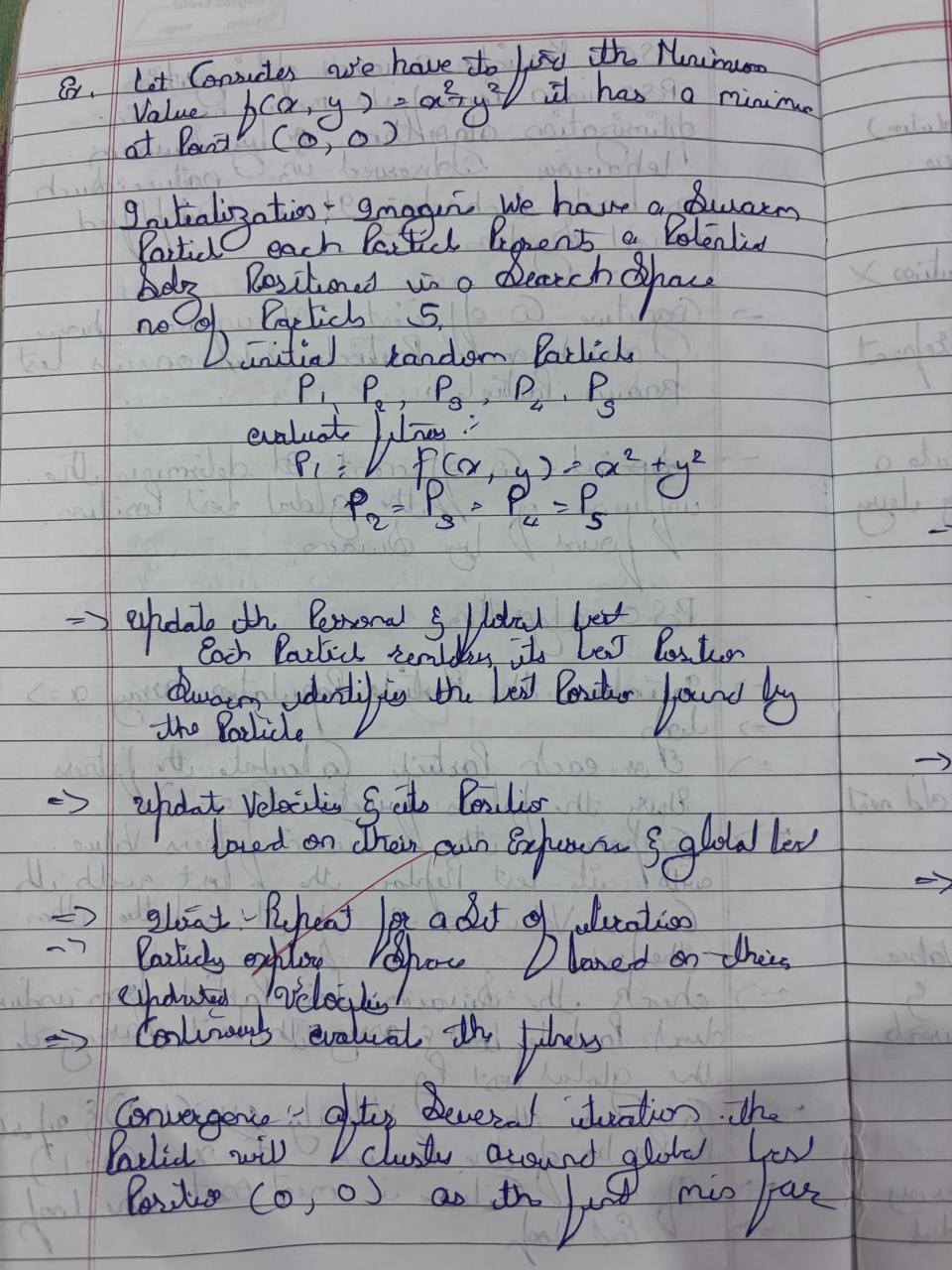
****

**Program 3**

**Particle Swarm Optimization**

**Algorithm:**

****

****

**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))

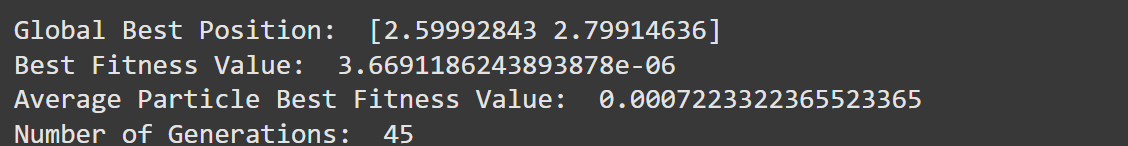
print('Average Particle Best Fitness Value: ', np.average(pbest\_fitness))

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)

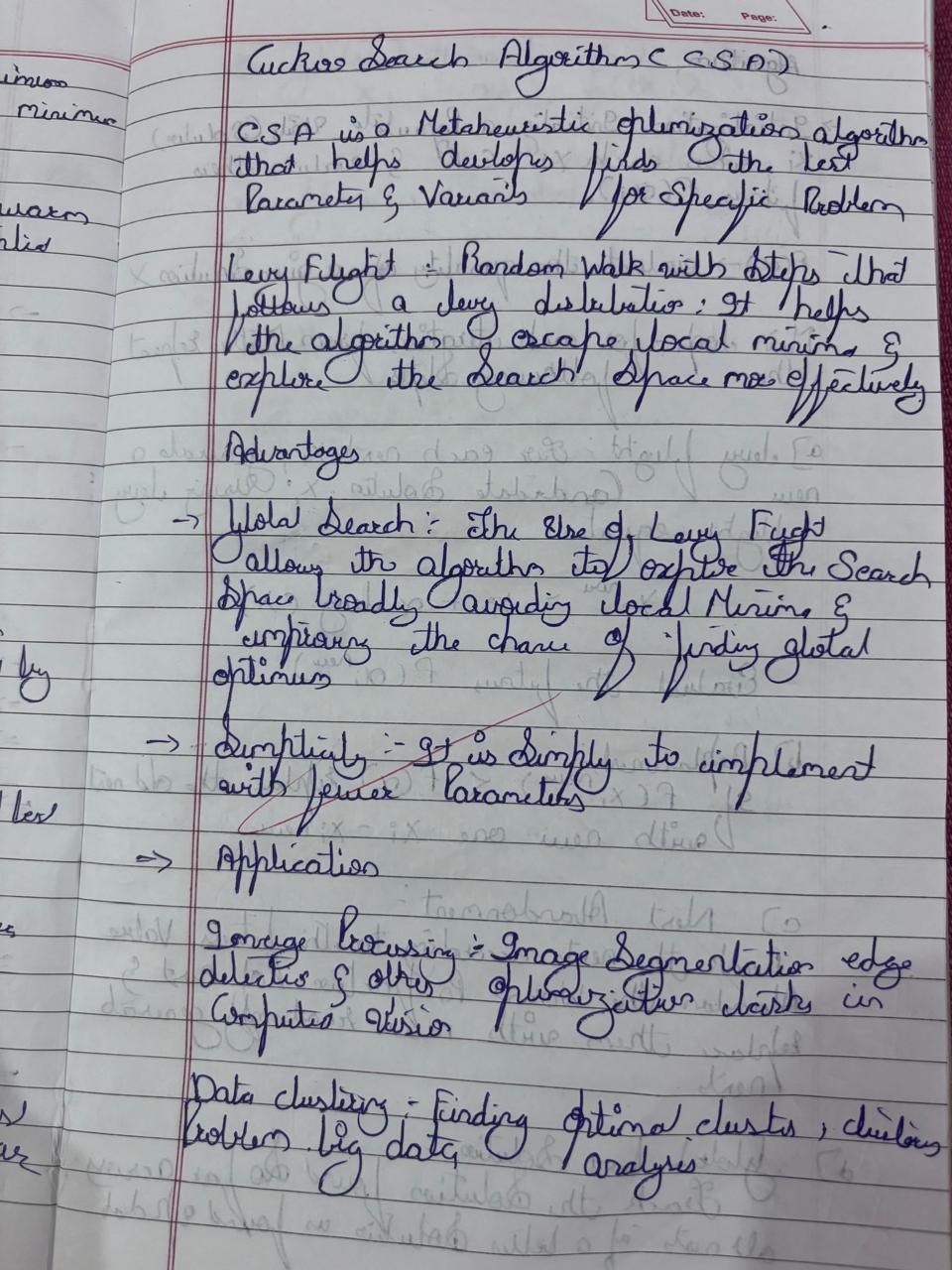
**Output:**

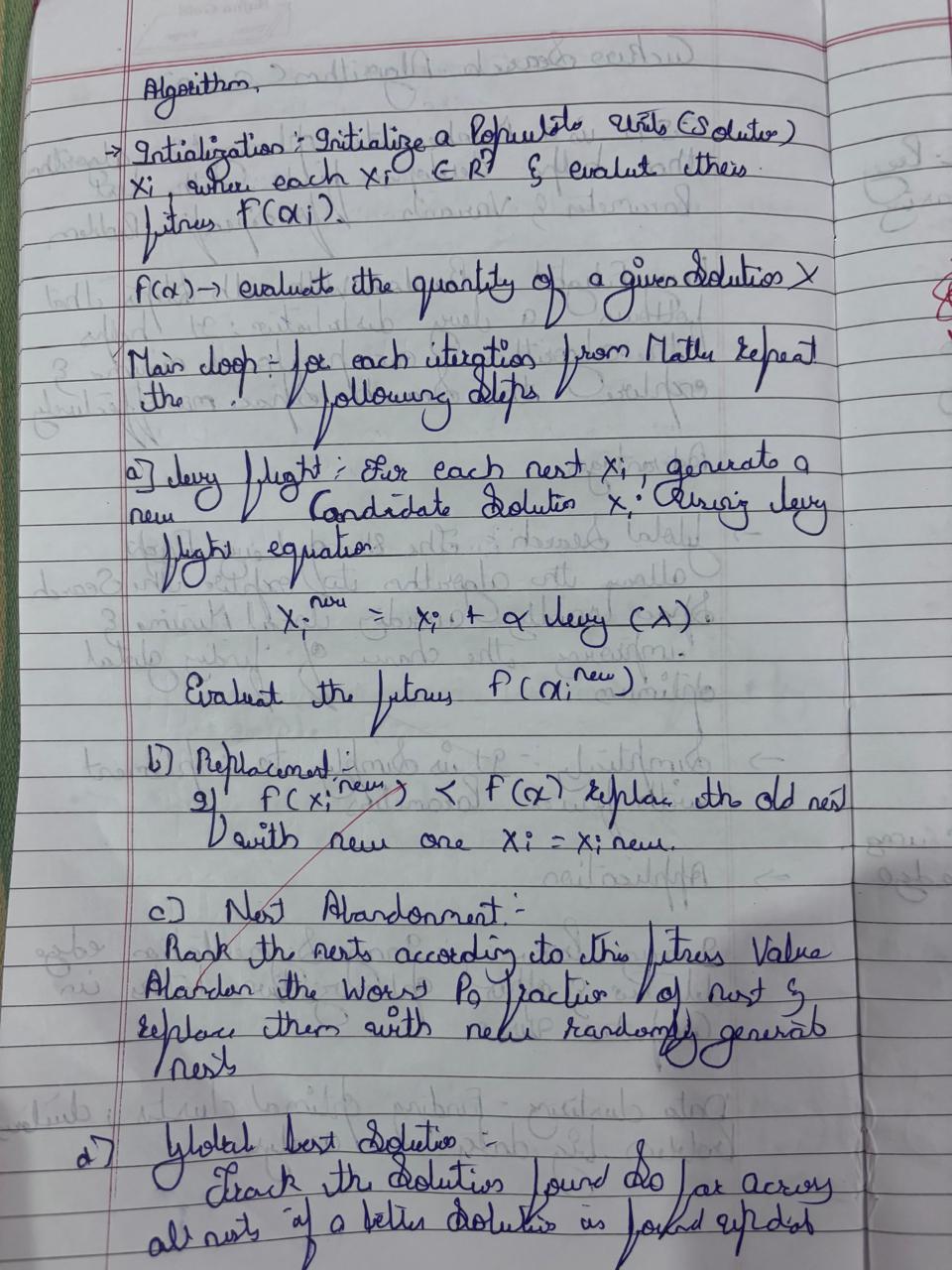
****

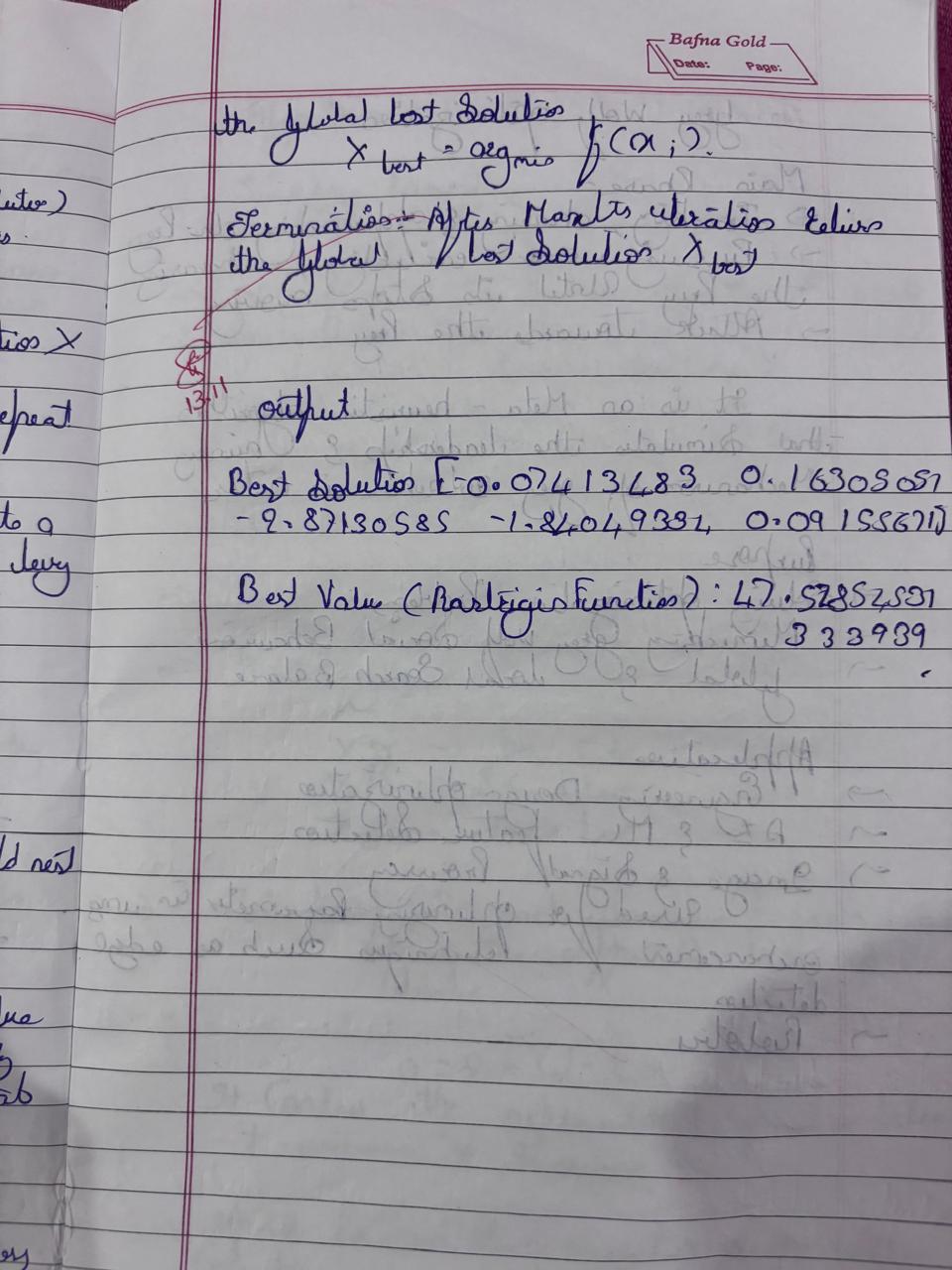
**Program 4**

**Cuckoo Search Algorithm**

**Algorithm:**

****

****

****

**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()

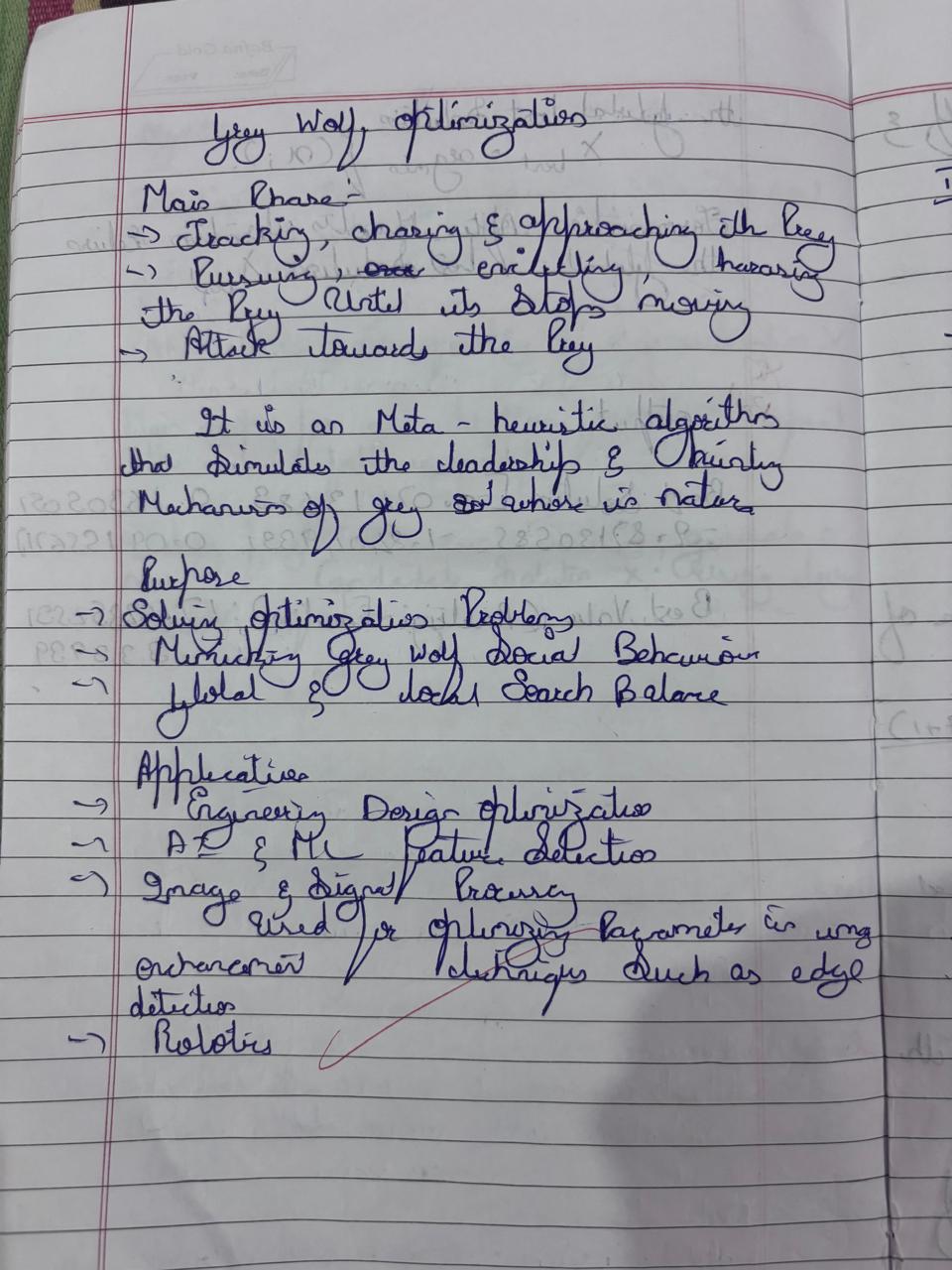
**Output:**

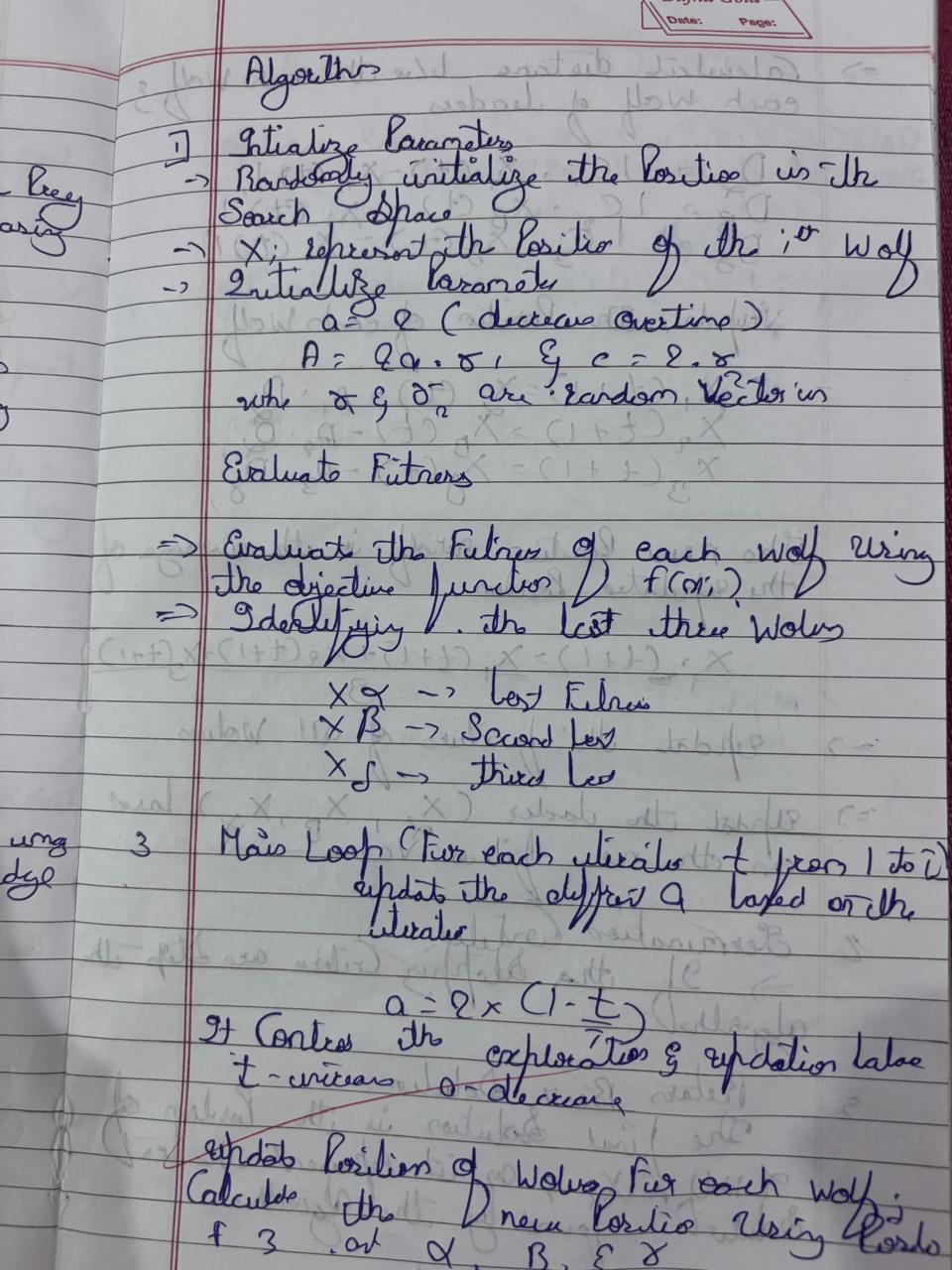
****

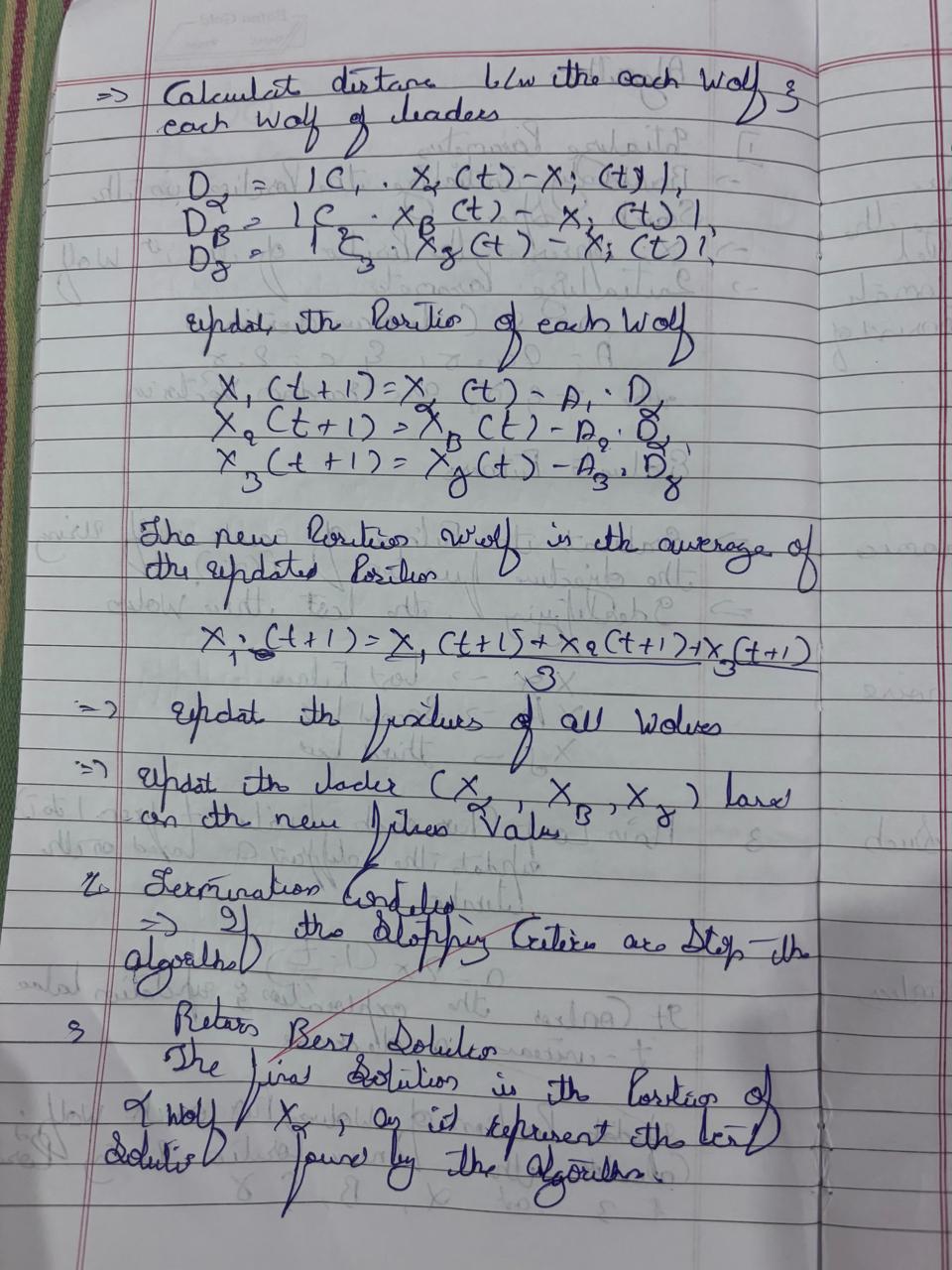
**Program 5**

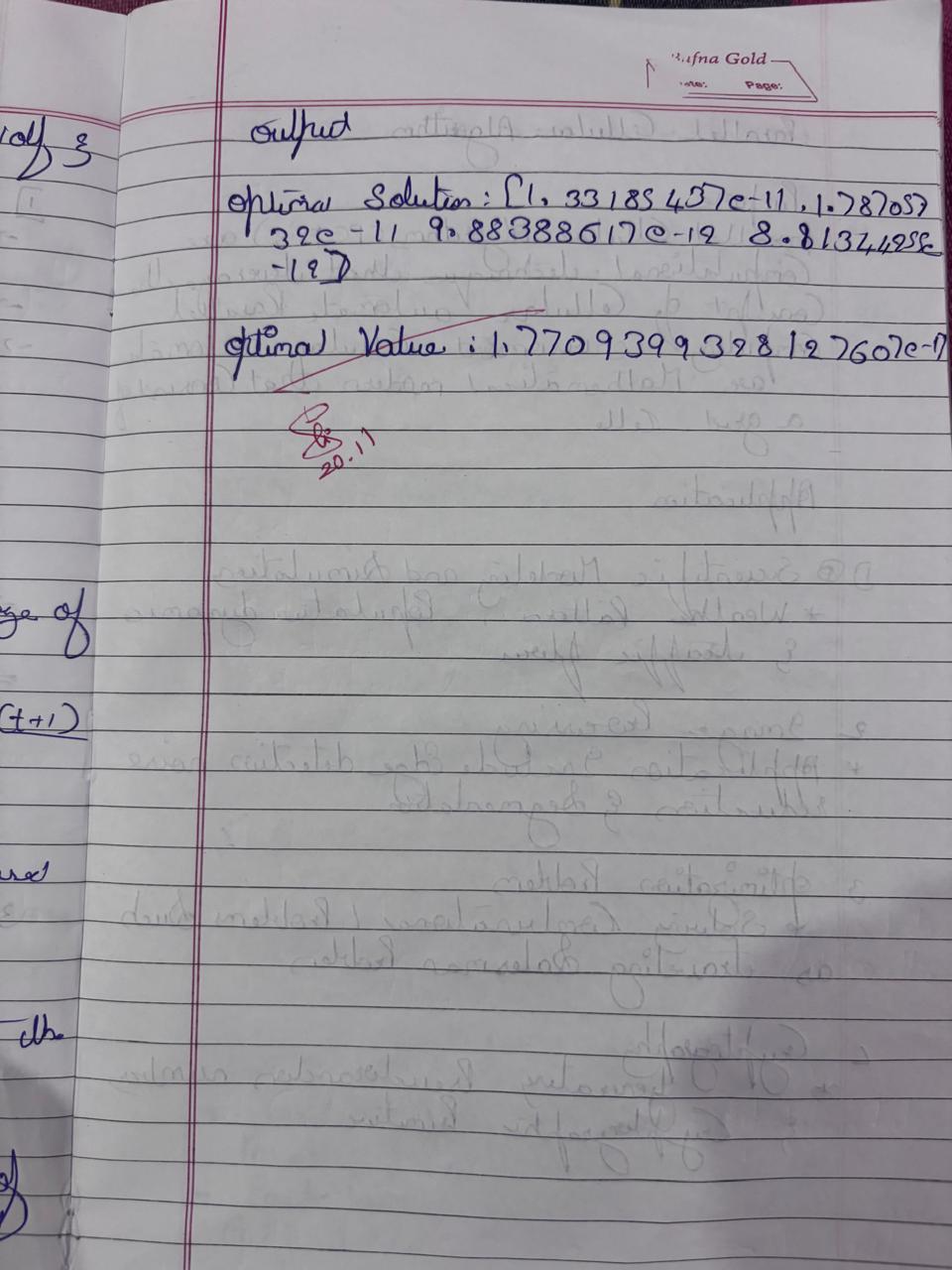
**Grey Wolf Optimizer**

**Algorithm:**

****

****

****

****

**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

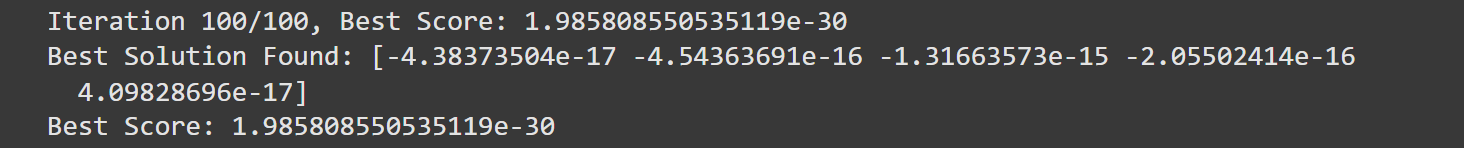
gwo = GreyWolfOptimizer(objective\_function, n\_wolves, n\_variables, max\_iter, lb, ub)

best\_solution, best\_score = gwo.optimize()

print("Best Solution Found:", best\_solution)

print("Best Score:", best\_score)

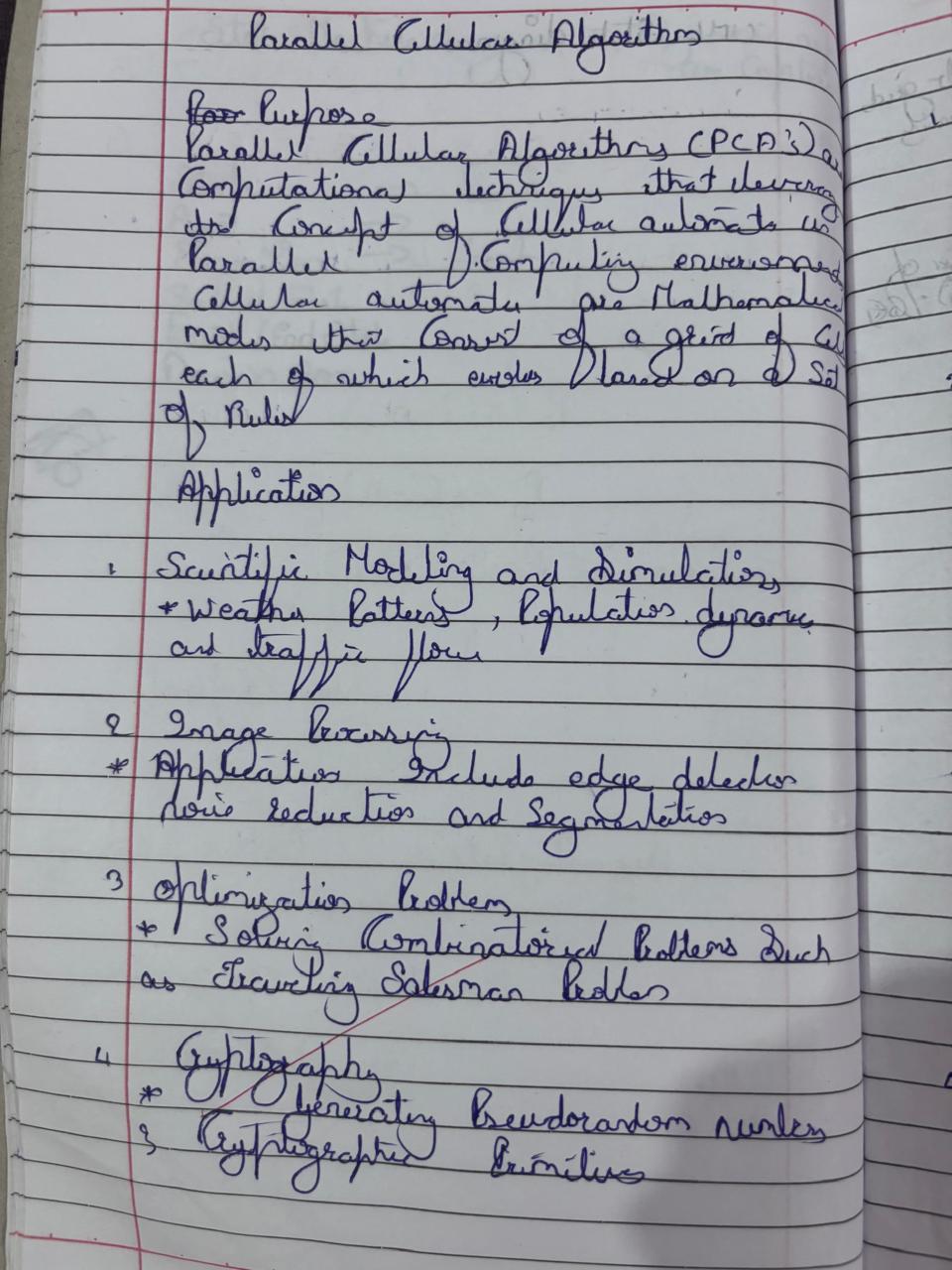
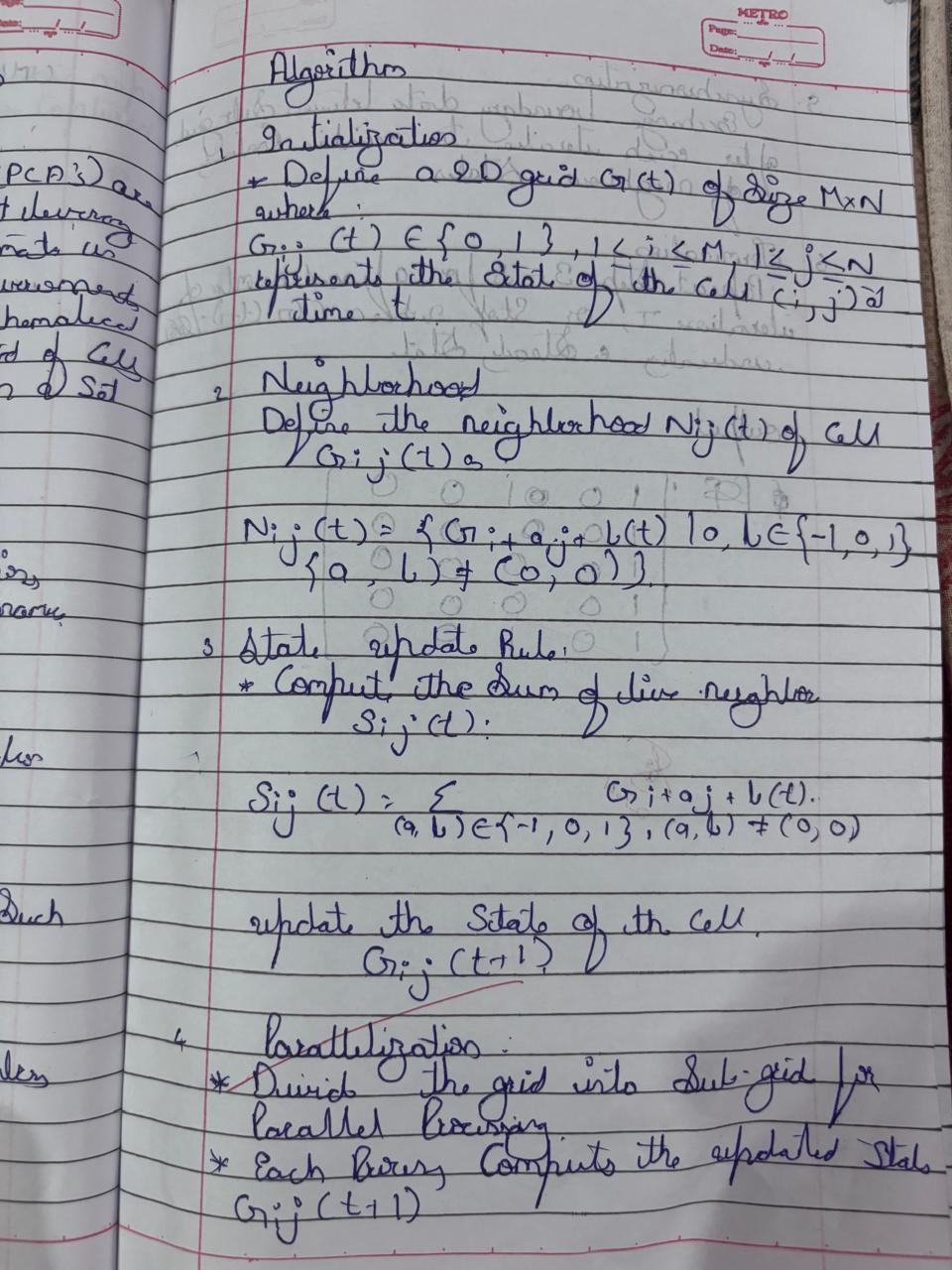
**Output:**

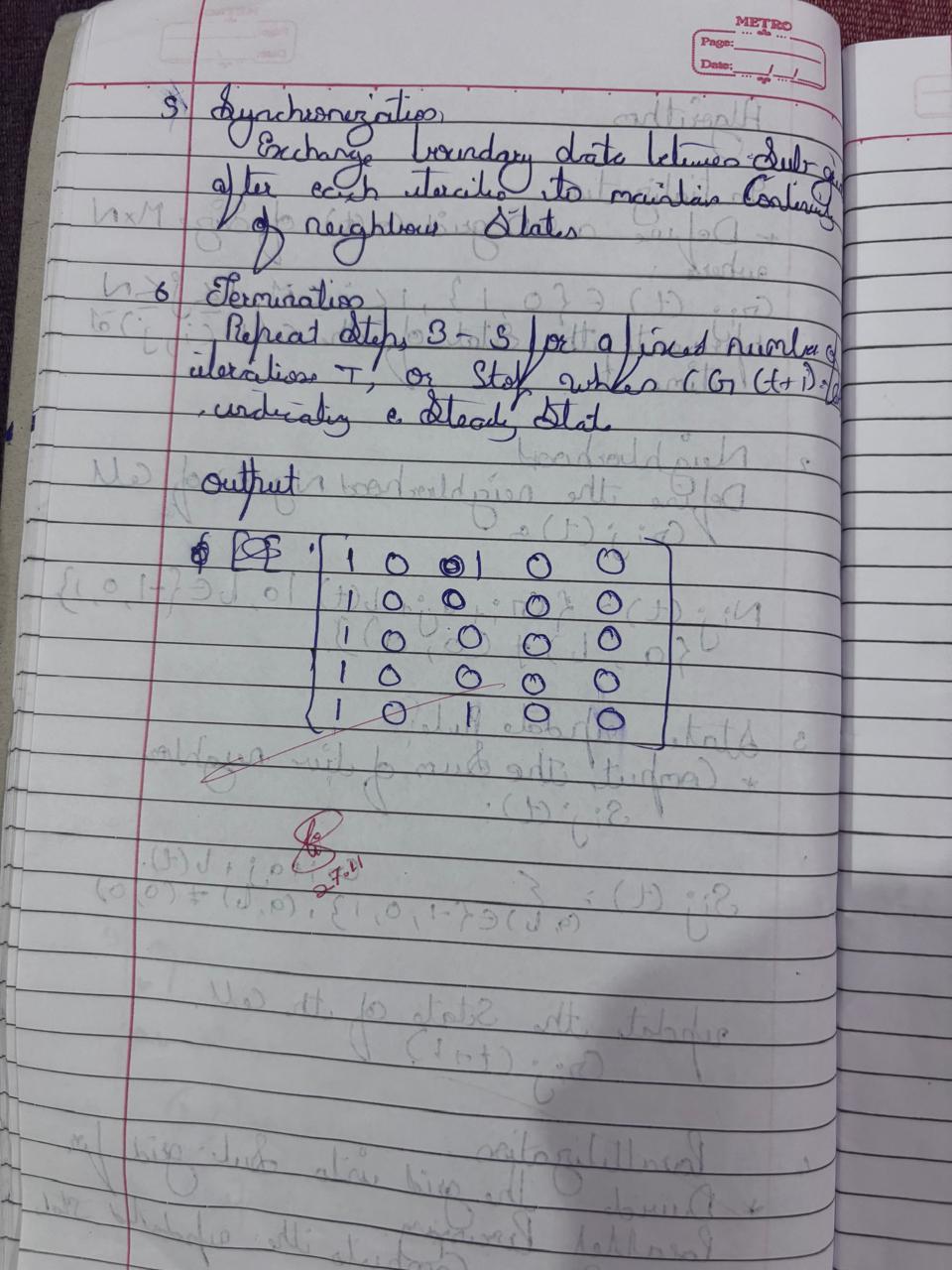


**Program 6**

**Parallel Cellular Algorithm**

**Algorithm:**

****

****

**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))

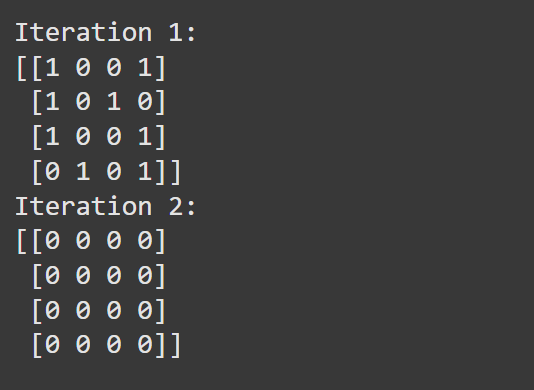
print("Initial state:")

print(grid)

num\_iterations = 2

updated\_grid = parallel\_update(grid, grid\_size, num\_iterations)

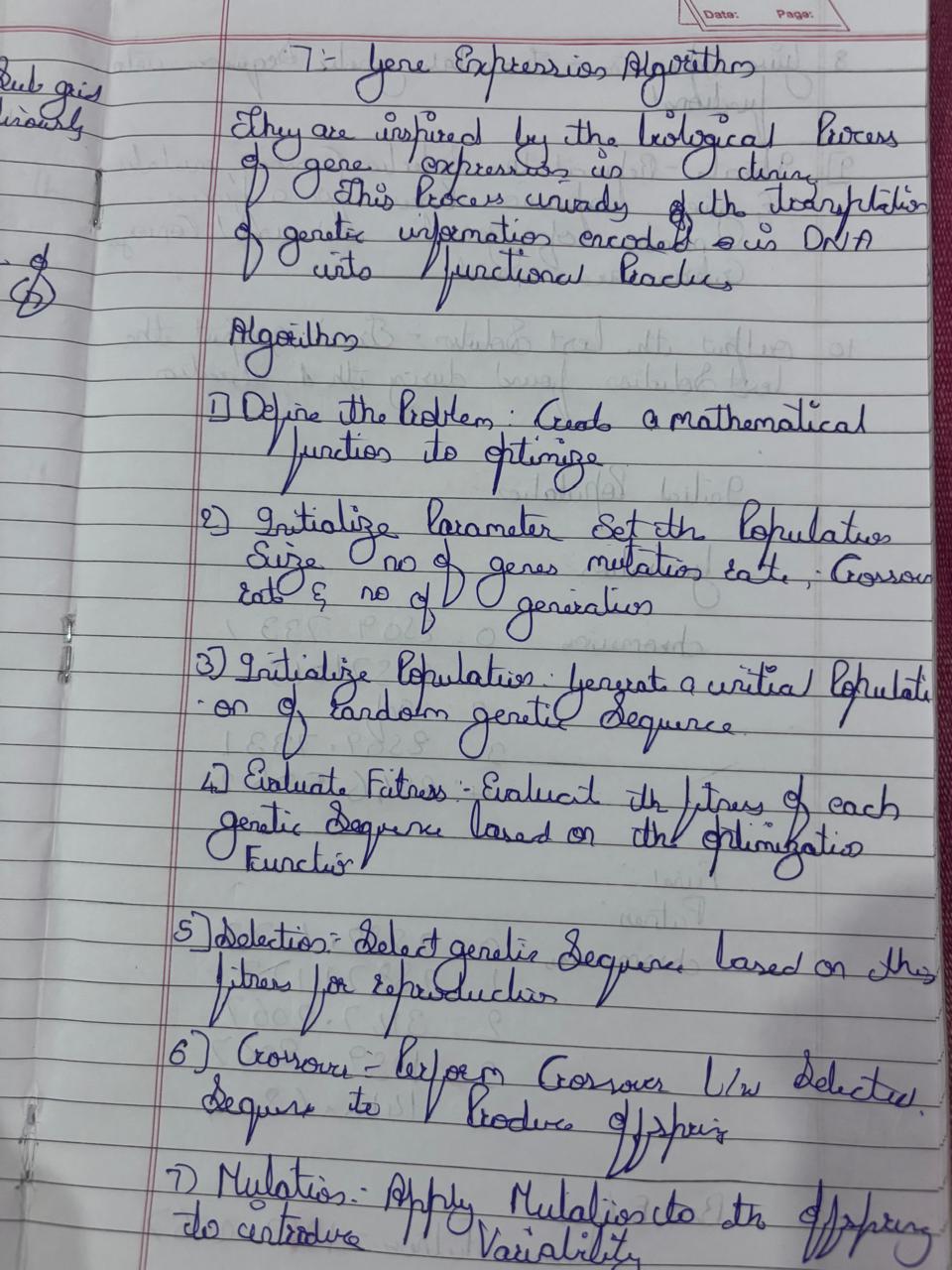
**Output:**

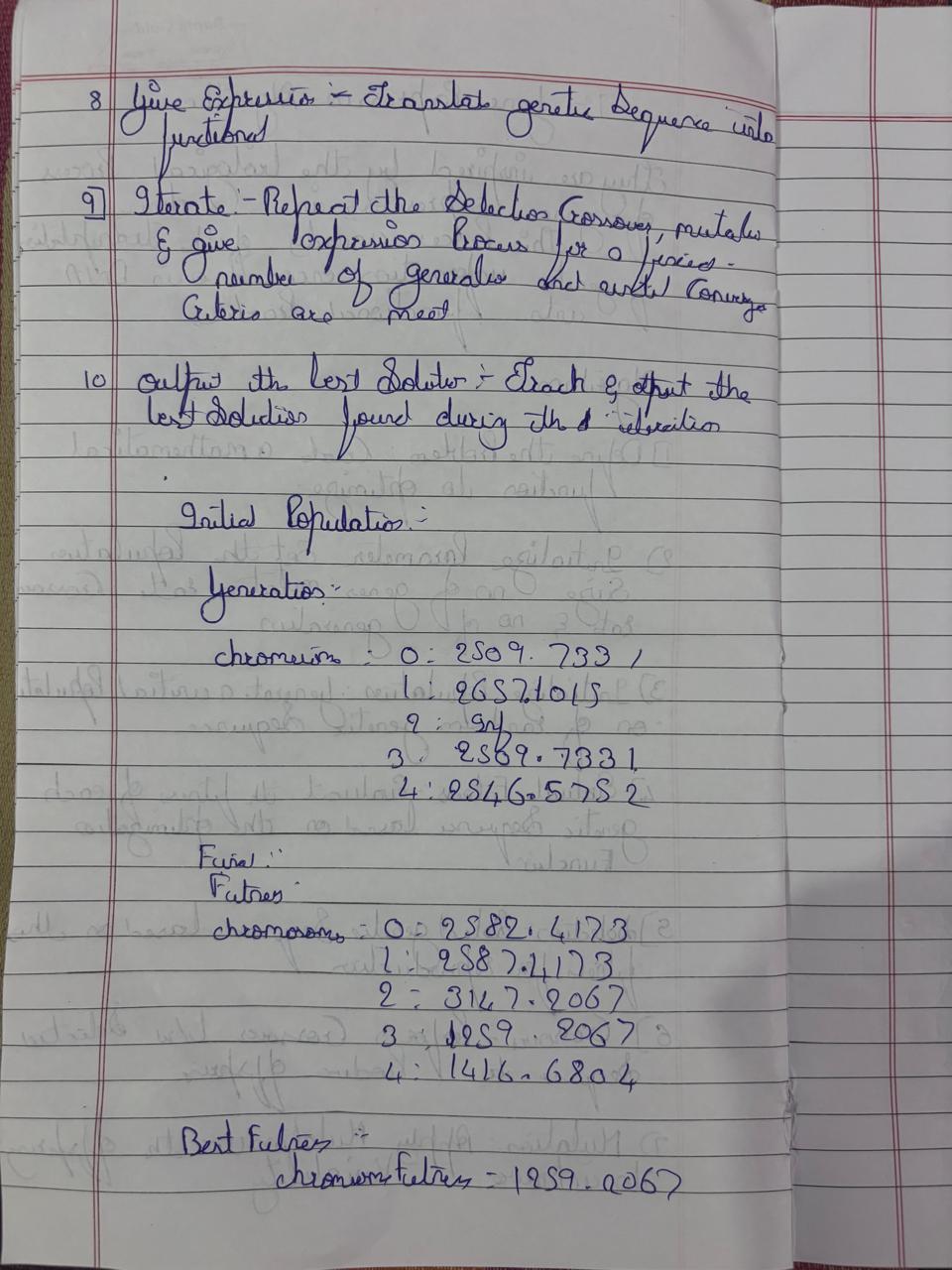
****

**Program 7**

**Gene Expression Algorithm**

**Algorithm:**

****

****

**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

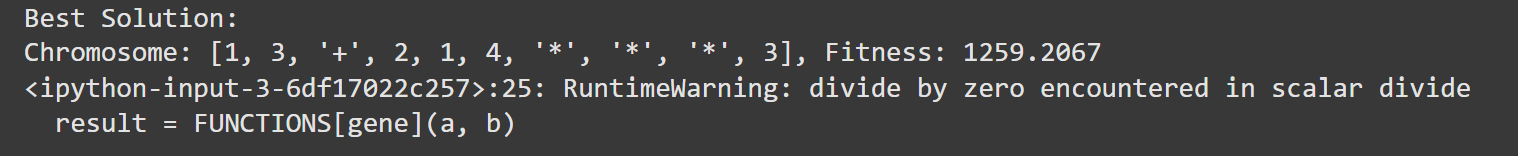
# Input values

x\_values = np.linspace(-10, 10, 20)

# Run the algorithm

gene\_expression\_algorithm(target\_function, x\_values, population\_size=10, generations=10)

**Output:**

****