|  |
| --- |
| **VISVESVARAYA TECHNOLOGICAL UNIVERSITY**  **“JnanaSangama”, Belgaum -590014, Karnataka.**    **LAB RECORD**  **Bio Inspired Systems (23CS5BSBIS)**  ***Submitted by***  **Shree Varna M (1BM22CS263)**  ***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 **Shree varna M (1BM22CS263),** 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.

|  |  |
| --- | --- |
| Spoorthi D M  Assistant Professor  Department of CSE, BMSCE | Dr. Jyothi S Nayak  Professor & HOD  Department of CSE, BMSCE |

**Index**

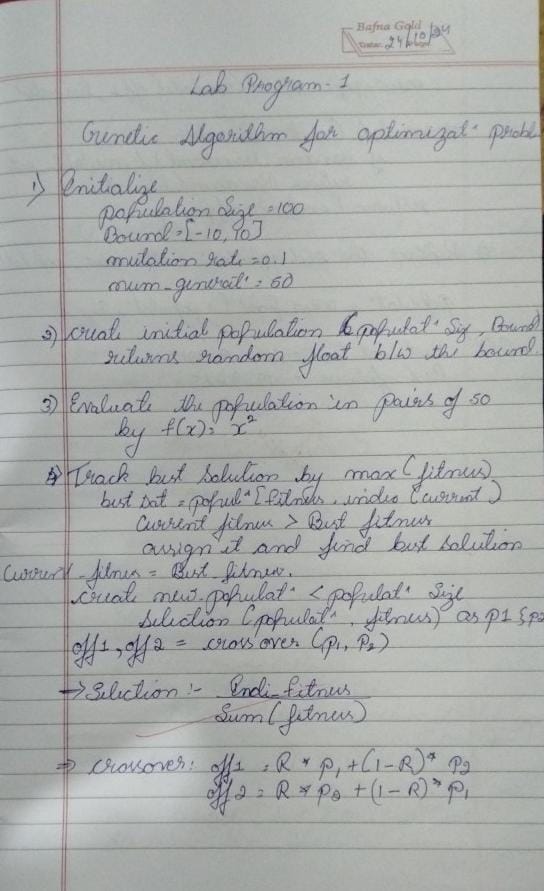
|  |  |  |  |
| --- | --- | --- | --- |
| **Sl.**  **No.** | **Date** | **Experiment Title** | **Page No.** |
| 1 | 24/10/2024 | Genetic Algorithm for Optimization Problem | 4 - 7 |
| 2 | 7/11/2024 | Particle Swam Optimization for Function Optimization | 8 - 11 |
| 3 | 14/11/2024 | Ant Colony Optimization fir the travelling Salesman Problem | 12 - 16 |
| 4 | 21/11/2024 | Cuckoo Search (CS) | 17 - 20 |
| 5 | 28/11/2024 | Grey Wolf Optimizer (GWO) | 21 - 25 |
| 6 | 05/11/2024 | Parallel Cellular Algorithms and Programs | 26 - 28 |
| 7 | 12/11/2024 | Optimization via Gene Expression Algorithm | 29 - 33 |

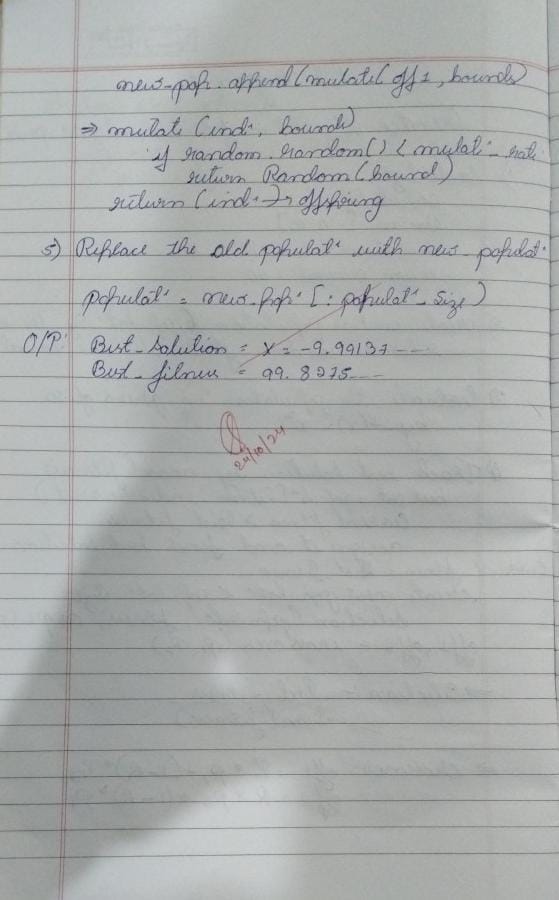
Github Link:

<https://github.com/Shree-varna/Bio_Inspires_Sys>

**Program 1**

Genetic Algorithm for Optimization Problem

Algorithm:



Code:

import numpy as np

import random

# Define the fitness function

def fitness\_function(x):

return x \*\* 2

# Initialize parameters

population\_size = 100

mutation\_rate = 0.1

num\_generations = 50

bounds = (-10, 10)

# Step 1: Create initial population

def create\_initial\_population(size, bounds):

return [random.uniform(bounds[0], bounds[1]) for \_ in

range(size)]

# Step 2: Evaluate fitness of the population

def evaluate\_population(population):

return [fitness\_function(individual) for individual in

population]

# Step 3: Selection using roulette-wheel selection

def selection(population, fitness):

total\_fitness = sum(fitness)

selection\_probs = [f / total\_fitness for f in fitness]

return np.random.choice(population, size=2, p=selection\_probs)

# Step 4: Crossover operation

def crossover(parent1, parent2):

alpha = random.uniform(0, 1)

offspring1 = alpha \* parent1 + (1 - alpha) \* parent2

offspring2 = alpha \* parent2 + (1 - alpha) \* parent1

return offspring1, offspring2

# Step 5: Mutation operationdef mutate(individual, bounds):

if random.random() < mutation\_rate:

return random.uniform(bounds[0], bounds[1])

return individual

# Main Genetic Algorithm loop

def genetic\_algorithm(bounds):

# Step 1: Create initial population

population = create\_initial\_population(population\_size, bounds)

best\_solution = None

best\_fitness = float('-inf')

for generation in range(num\_generations):

# Step 2: Evaluate fitness

fitness = evaluate\_population(population)

# Track the best solution

current\_best\_fitness = max(fitness)

if current\_best\_fitness > best\_fitness:

best\_fitness = current\_best\_fitness

best\_solution =

population[fitness.index(current\_best\_fitness)]

# Step 3: Create new population

new\_population = []

while len(new\_population) < population\_size:

parent1, parent2 = selection(population, fitness)

offspring1, offspring2 = crossover(parent1, parent2)

new\_population.append(mutate(offspring1, bounds))

new\_population.append(mutate(offspring2, bounds))

# Replace the old population with the new population

population = new\_population[:population\_size]

return best\_solution, best\_fitness

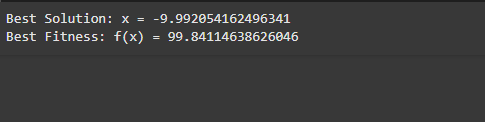
# Run the Genetic Algorithm

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

print(f"Best Solution: x = {best\_solution}")

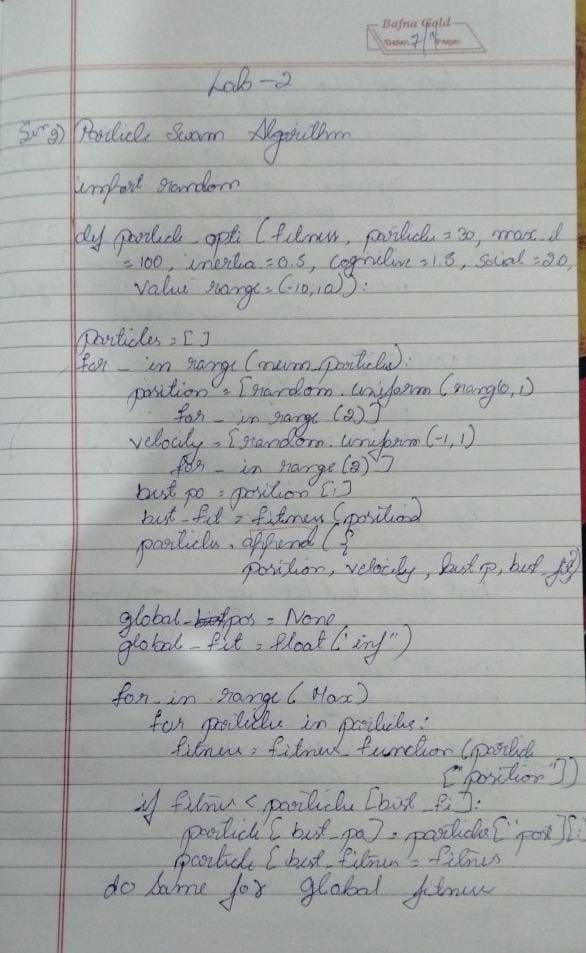
print(f"Best Fitness: f(x) = {best\_fitness}")

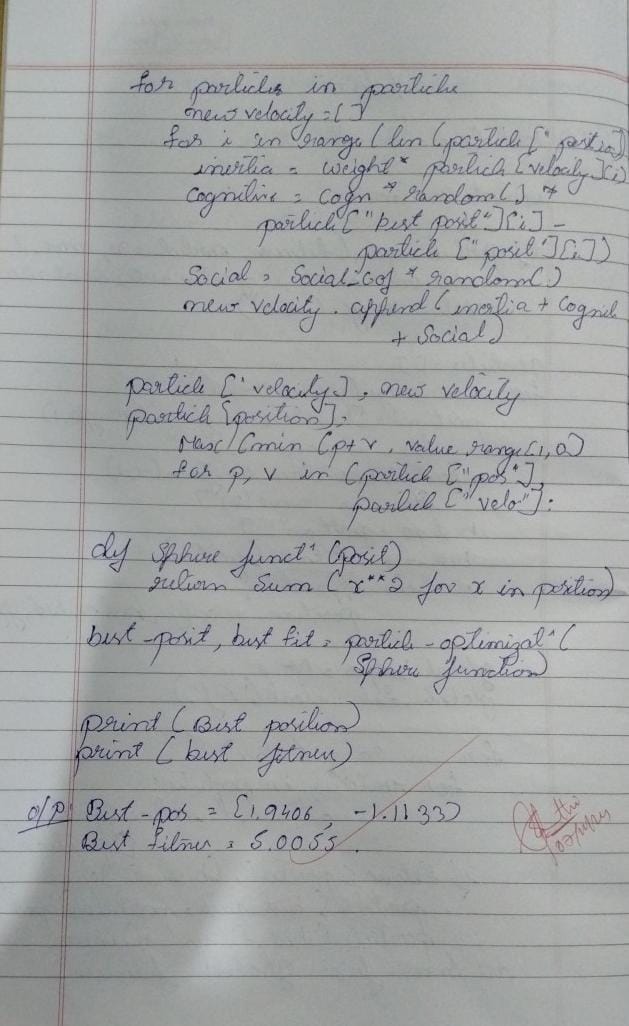
**OUTPUT :**



**Program 2**

Particle Swam Optimization for Function Optimization

Algorithm:

****

**CODE :**

import random

def particle\_swarm\_optimization(fitness\_function, num\_particles=30,

max\_iterations=100,

inertia\_weight=0.5,

cognitive\_coef=1.5, social\_coef=2.0,

value\_range=(-10, 10)):

# Initialize particles

particles = []

for \_ in range(num\_particles):

# Random position and velocity within the specified range

position = [random.uniform(value\_range[0], value\_range[1])

for \_ in range(2)] # 2D example

velocity = [random.uniform(-1, 1) for \_ in range(2)]

best\_position = position[:]

best\_fitness = fitness\_function(position)

particles.append({

"position": position,

"velocity": velocity,

"best\_position": best\_position,

"best\_fitness": best\_fitness

})

# Initialize global best

global\_best\_position = None

global\_best\_fitness = float("inf")

# Main optimization loop

for \_ in range(max\_iterations):

for particle in particles:

# Calculate fitness

fitness = fitness\_function(particle["position"])

# Update personal best

if fitness < particle["best\_fitness"]:

particle["best\_position"] = particle["position"][:]

particle["best\_fitness"] = fitness

# Update global bestif fitness < global\_best\_fitness:

global\_best\_position = particle["position"][:]

global\_best\_fitness = fitness

# Update velocities and positions

for particle in particles:

new\_velocity = []

for i in range(len(particle["position"])):

inertia = inertia\_weight \* particle["velocity"][i]

cognitive = cognitive\_coef \* random.random() \*

(particle["best\_position"][i] - particle["position"][i])

social = social\_coef \* random.random() \*

(global\_best\_position[i] - particle["position"][i])

new\_velocity.append(inertia + cognitive + social)

particle["velocity"] = new\_velocity

particle["position"] = [

max(min(p + v, value\_range[1]), value\_range[0]) #

Clip to value\_range

for p, v in zip(particle["position"],

particle["velocity"])

]

return global\_best\_position, global\_best\_fitness

# Example usage with a function to optimize (Sphere function)

def sphere\_function(position):

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

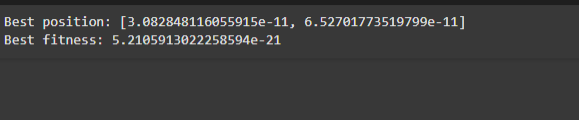
best\_position, best\_fitness =

particle\_swarm\_optimization(sphere\_function)

print("Best position:", best\_position)

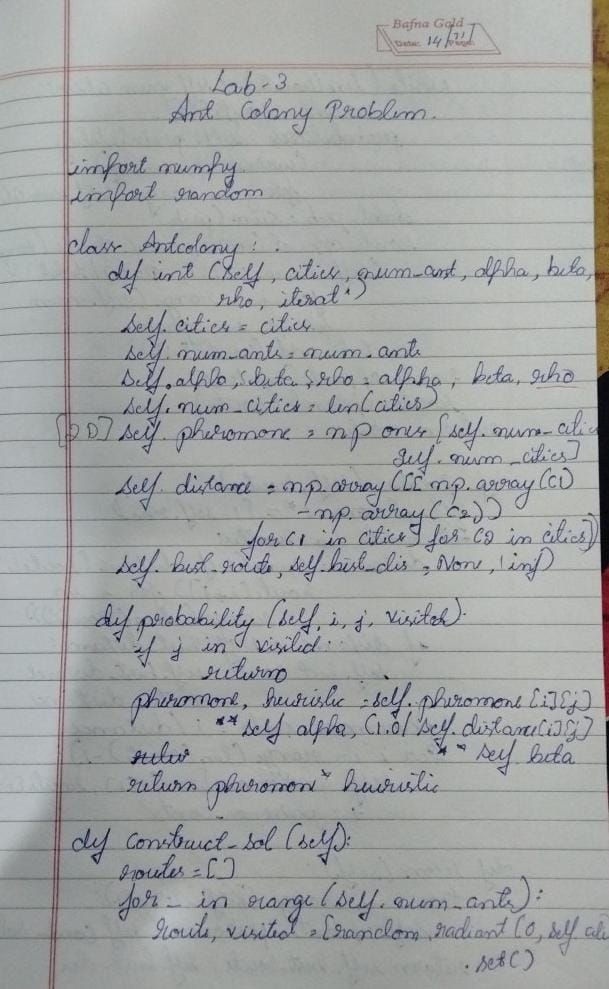
print("Best fitness:", best\_fitness)

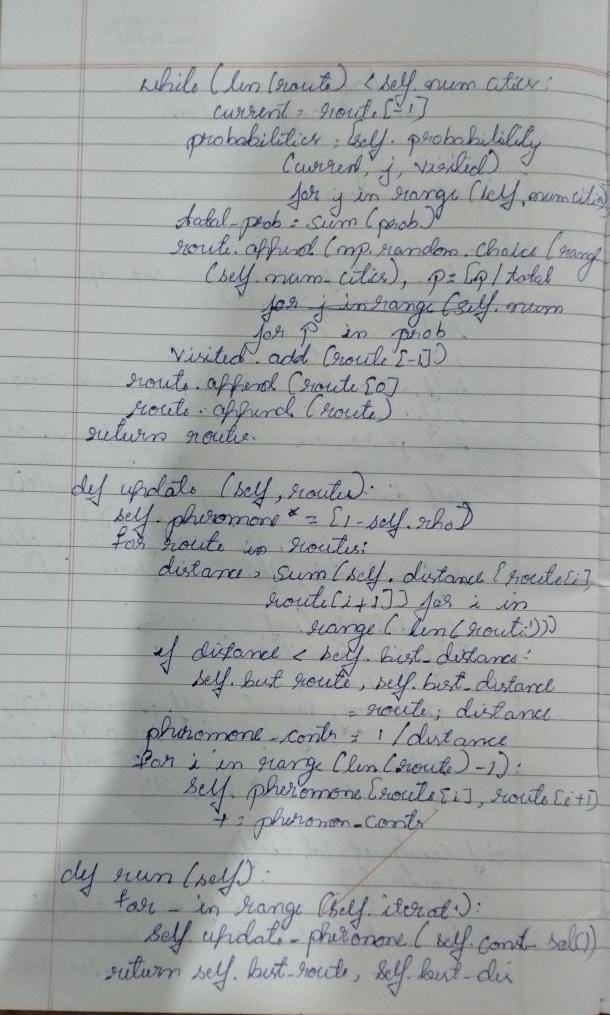
**OUTPUT :**

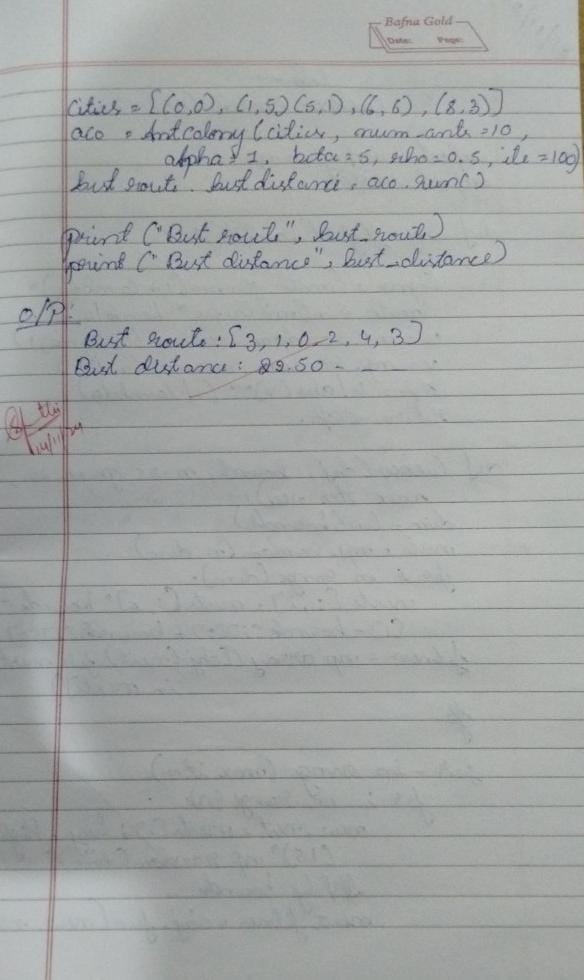


**Program 3**

Ant Colony Optimization fir the travelling Salesman Problem

Algorithm:



****

**CODE :**

import numpy as np

import random

class AntColonyOptimization:

def \_\_init\_\_(self, cities, num\_ants, alpha, beta, rho, iterations):

self.cities = cities

self.num\_ants = num\_ants

self.alpha = alpha # importance of pheromone

self.beta = beta # importance of heuristic (inverse distance)

self.rho = rho # pheromone evaporation rate

self.iterations = iterations

self.num\_cities = len(cities)

# Initialize pheromone levels

self.pheromone = np.ones((self.num\_cities, self.num\_cities))

self.distances = self.calculate\_distances()

# Track the best route found

self.best\_route = None

self.best\_distance = float('inf')

def calculate\_distances(self):

""" Calculate distance matrix between cities """

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

for i in range(self.num\_cities):

for j in range(self.num\_cities):

if i != j:

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

return distances

def probability(self, i, j, visited):

""" Calculate probability of moving from city i to city j """

if j in visited:

return 0

pheromone = self.pheromone[i][j] \*\* self.alpha

heuristic = (1.0 / self.distances[i][j]) \*\* self.beta if self.distances[i][j] != 0 else 0

return pheromone \* heuristic

def construct\_solution(self):

""" Construct a route for each ant based on probabilities """all\_routes = []

for \_ in range(self.num\_ants):

route = [random.randint(0, self.num\_cities - 1)]

while len(route) < self.num\_cities:

current\_city = route[-1]

probabilities = [self.probability(current\_city, j, route) for j in

range(self.num\_cities)]

total\_prob = sum(probabilities)

probabilities = [p / total\_prob if total\_prob > 0 else 0 for p in probabilities]

next\_city = np.random.choice(range(self.num\_cities), p=probabilities)

route.append(next\_city)

route.append(route[0]) # return to starting city

all\_routes.append(route)

return all\_routes

def route\_distance(self, route):

""" Calculate total distance of a route """

return sum([self.distances[route[i]][route[i + 1]] for i in range(len(route) - 1)])

def update\_pheromones(self, all\_routes):

""" Update pheromone levels based on routes found """

self.pheromone \*= (1 - self.rho) # evaporate some pheromone

for route in all\_routes:

distance = self.route\_distance(route)

if distance < self.best\_distance:

self.best\_distance = distance

self.best\_route = route

pheromone\_contribution = 1 / distance

for i in range(len(route) - 1):

self.pheromone[route[i]][route[i + 1]] += pheromone\_contribution

def run(self):

""" Execute the ACO algorithm """

for \_ in range(self.iterations):

all\_routes = self.construct\_solution()

self.update\_pheromones(all\_routes)

return self.best\_route, self.best\_distance

# Example usage

cities = [(0, 0), (1, 5), (5, 1), (6, 6), (8, 3)] # example set of city coordinates

aco = AntColonyOptimization(cities, num\_ants=10, alpha=1, beta=5, rho=0.5,

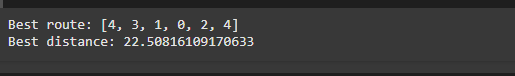
iterations=100)

best\_route, best\_distance = aco.run()

print("Best route:", best\_route)

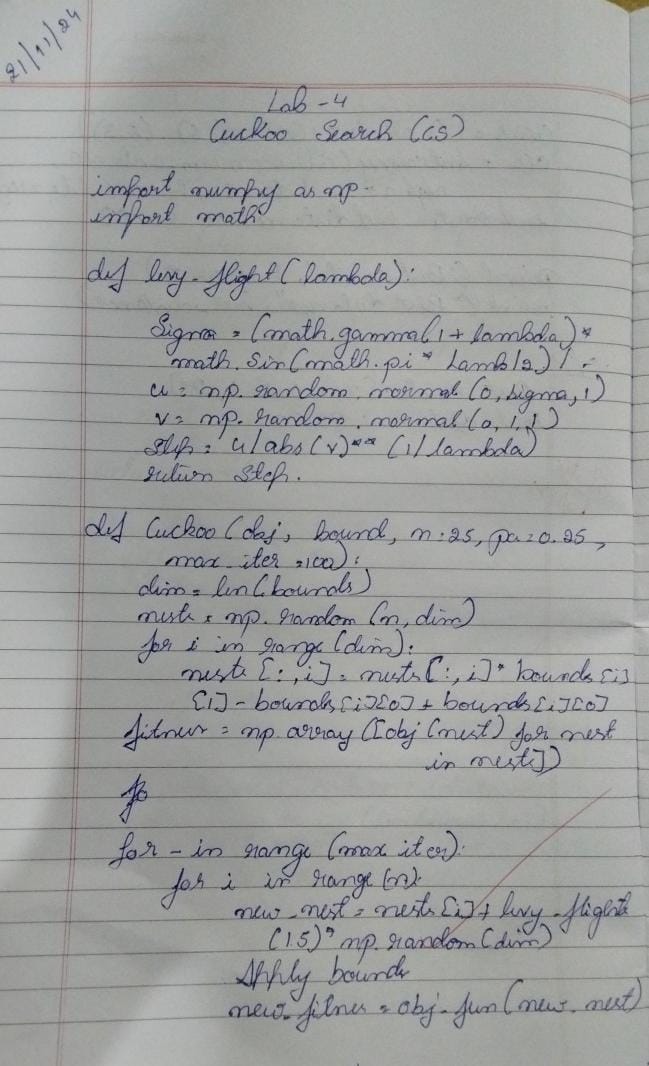
print("Best distance:", best\_distance)

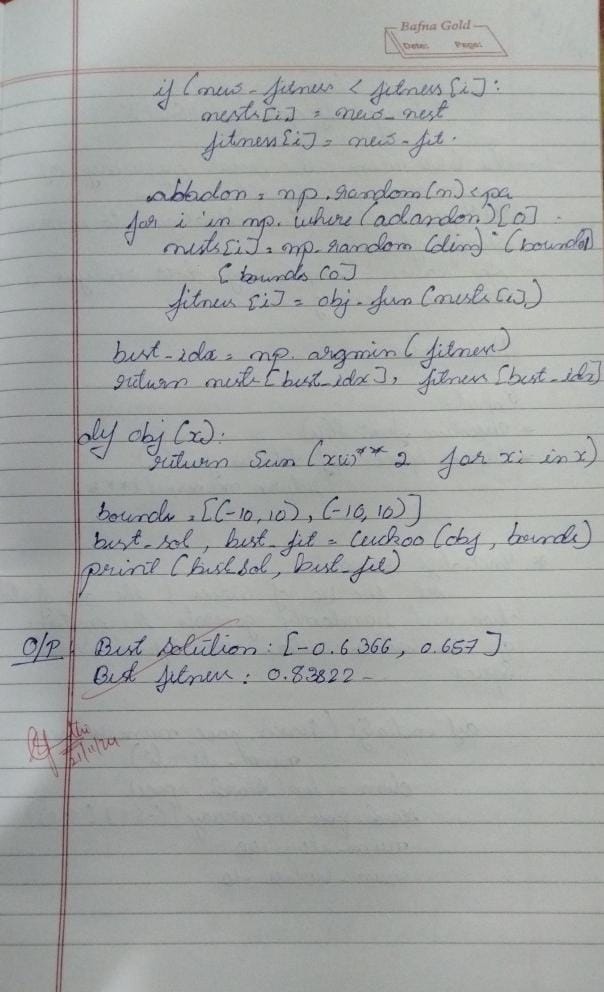
**OUTPUT :**



**Program 4**

Cuckoo Search (CS)

Algorithm:



**CODE :**

import numpy as np

import math # Import the standard math module

def levy\_flight(Lambda):

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

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

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

v = np.random.normal(0, 1, 1)

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

return step

def cuckoo\_search(obj\_function, bounds, n=25, pa=0.25, max\_iter=100):

# Initialize nests

dim = len(bounds)

nests = np.random.rand(n, dim)

for i in range(dim):

nests[:, i] = nests[:, i] \* (bounds[i][1] - bounds[i][0]) + bounds[i][0]

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

# Start optimization

for \_ in range(max\_iter):

for i in range(n):

# Generate a new solution via Levy flight

new\_nest = nests[i] + levy\_flight(1.5) \* np.random.randn(dim)

# Apply bounds

new\_nest = np.clip(new\_nest, [b[0] for b in bounds], [b[1] for b in bounds])

new\_fitness = obj\_function(new\_nest)

# Update if new solution is better

if new\_fitness < fitness[i]:

nests[i] = new\_nest

fitness[i] = new\_fitness

# Abandon some nests and create new ones

abandon\_idx = np.random.rand(n) < pa

for i in np.where(abandon\_idx)[0]:

nests[i] = np.random.rand(dim) \* (np.array([b[1] for b in bounds]) - np.array([b[0] for b in bounds])) + np.array([b[0] for b in bounds])

fitness[i] = obj\_function(nests[i])

# Return the best solution

best\_idx = np.argmin(fitness)

return nests[best\_idx], fitness[best\_idx]

# Example usage: Minimize f(x) = x^2

def objective(x):

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

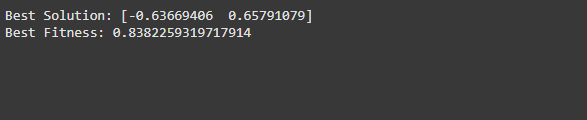
bounds = [(-10, 10), (-10, 10)] # 2D problem

best\_solution, best\_fitness = cuckoo\_search(objective, bounds)

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

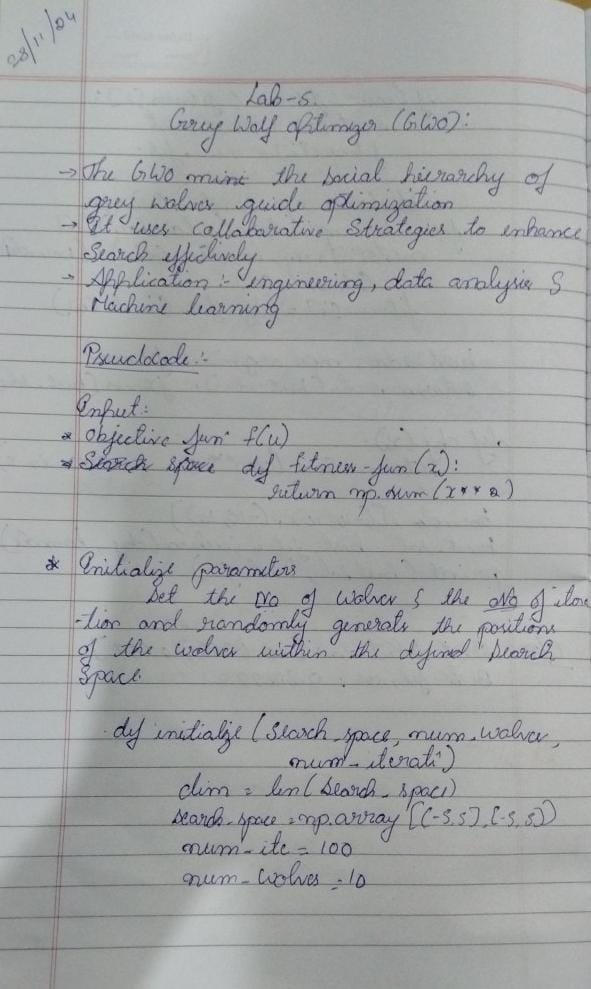
print("Best Fitness:", best\_fitness)

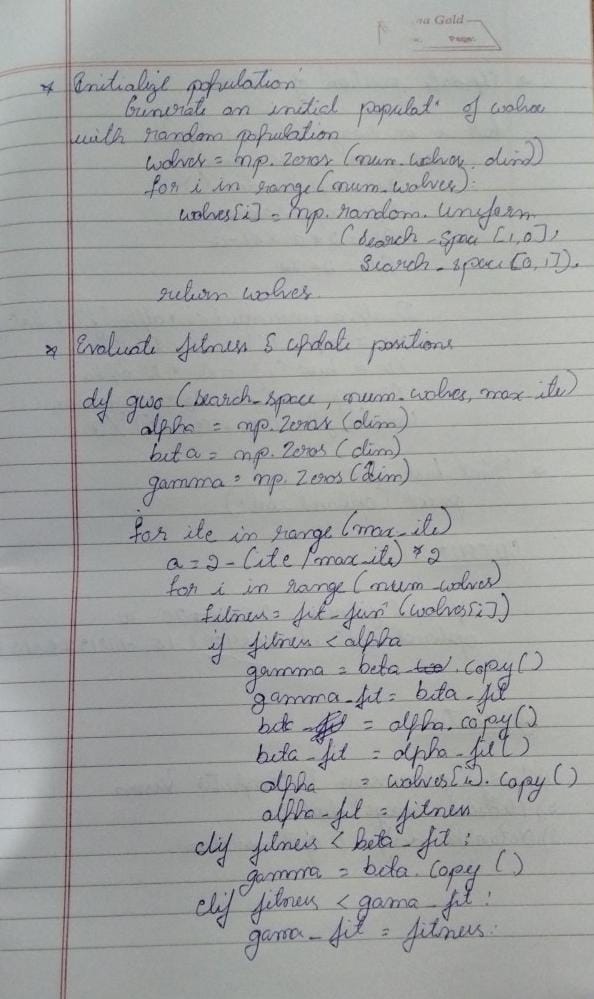
OUTPUT :

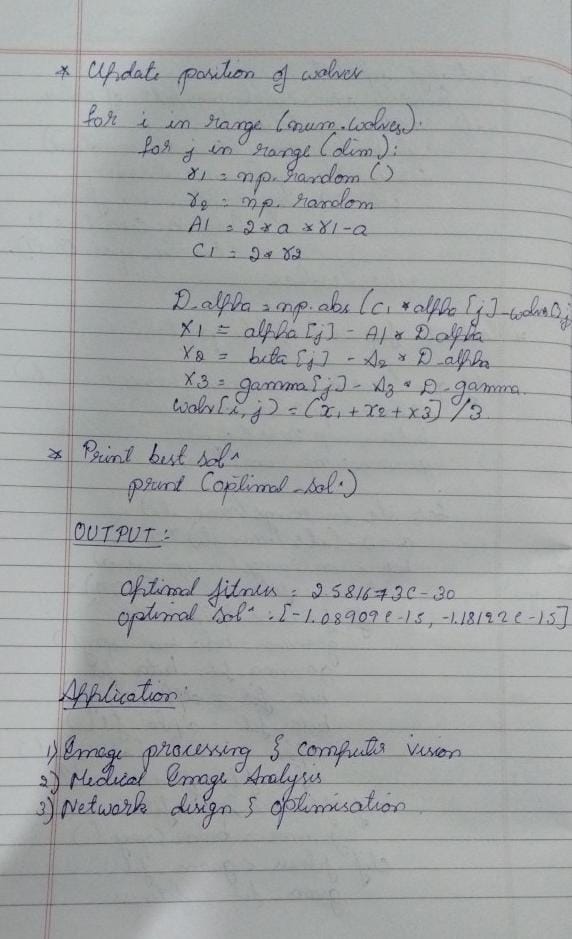


**Program 5**

Grey Wolf Optimizer (GWO)

Algorithm:



****

**CODE :**

import numpy as np

# Define the problem (a simple function to minimize)

def objective\_function(x):

return np.sum(x\*\*2) # Sphere function

# GWO Algorithm

def gwo(num\_wolves, num\_iterations, dim):

# Step 1: Initialize wolves' positions randomly

wolves = np.random.uniform(-10, 10, (num\_wolves, dim))

# Step 2: Initialize alpha, beta, delta wolves' positions and fitness values

alpha\_pos = np.zeros(dim)

beta\_pos = np.zeros(dim)

delta\_pos = np.zeros(dim)

alpha\_score = float('inf')

beta\_score = float('inf')

delta\_score = float('inf')

# Step 3: Iterate for given number of iterations

for \_ in range(num\_iterations):

for i in range(num\_wolves): #valuates each wolf's position and updates their fitness

fitness = objective\_function(wolves[i])

# Update alpha, beta, and delta wolves

if fitness < alpha\_score:

delta\_score = beta\_score

delta\_pos = beta\_pos

beta\_score = alpha\_score

beta\_pos = alpha\_pos

alpha\_score = fitness

alpha\_pos = wolves[i]

elif fitness < beta\_score:

delta\_score = beta\_score

delta\_pos = beta\_pos

beta\_score = fitness

beta\_pos = wolves[i]

elif fitness < delta\_score:

delta\_score = fitness

delta\_pos = wolves[i]

# Step 4: Update wolf positions

a = 2 - 2 \* (\_ / num\_iterations) # Decreasing factor

for i in range(num\_wolves):

r1, r2 = np.random.rand(2)

A = 2 \* a \* r1 - a

C = 2 \* r2

D\_alpha = abs(C \* alpha\_pos - wolves[i])

D\_beta = abs(C \* beta\_pos - wolves[i])

D\_delta = abs(C \* delta\_pos - wolves[i])

wolves[i] = wolves[i] - A \* D\_alpha - A \* D\_beta - A \* D\_delta

return alpha\_pos, alpha\_score

# Run GWO

best\_pos, best\_score = gwo(30, 100, 2)

print("Best Position:", best\_pos)

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

OUTPUT:

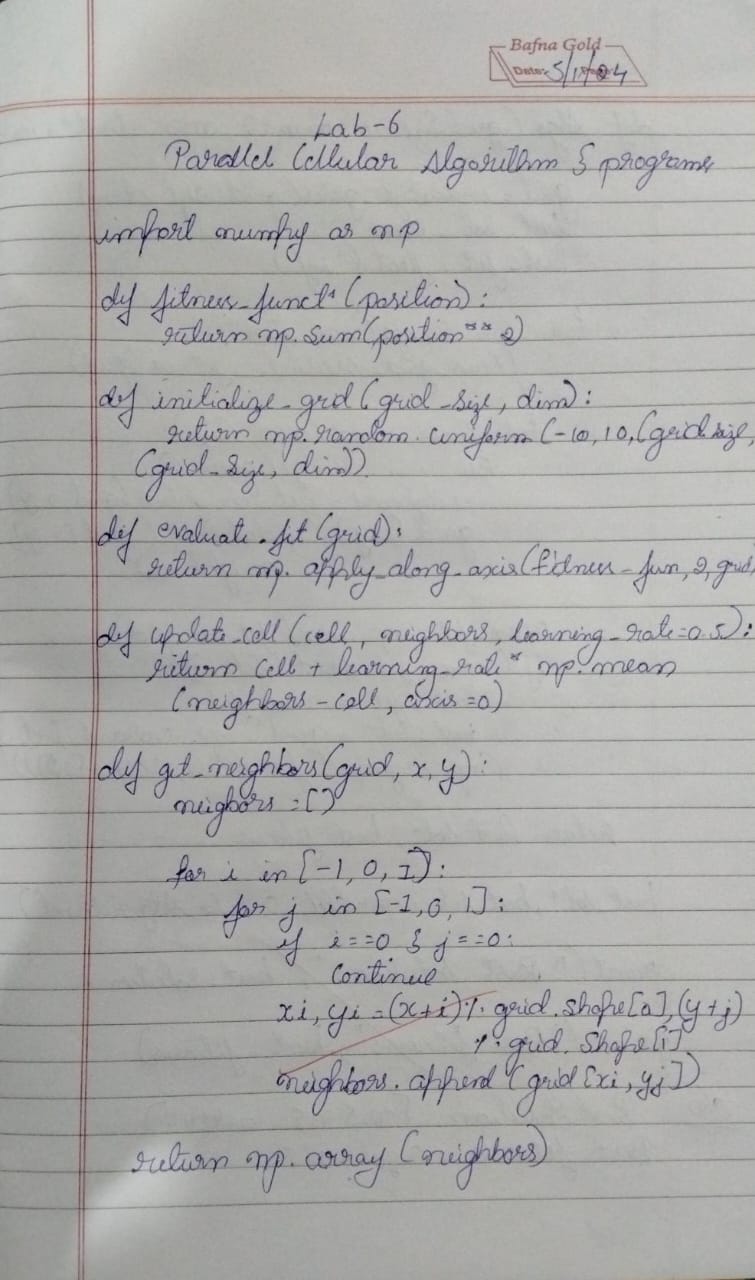
Best Position: [3.35381149e+14 4.81097512e+13]

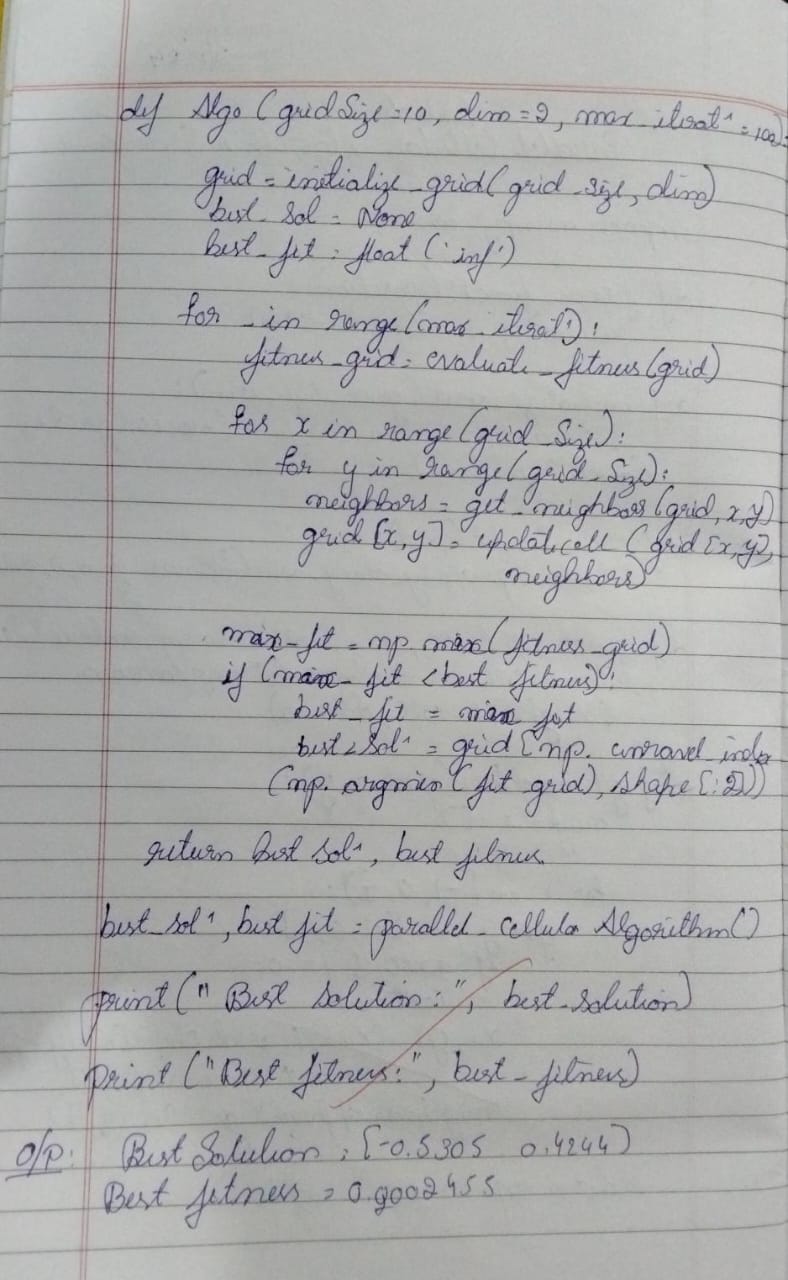
Best Score: 4.108979235848214

**Program 6**

Parallel cellular Algorithm and Programs

Algorithm :



Code:

#Parallel Cellular Algorithms and Programs

import numpy as np

# Define the optimization function

def fitness\_function(x):

return x\*\*2

# Initialize parameters

num\_cells = 10

grid\_size = 1.0

iterations = 100

neighborhood\_size = 1

# Initialize population

cells = np.random.uniform(-grid\_size, grid\_size, num\_cells)

# Main loop

for \_ in range(iterations):

# Evaluate fitness

fitness = np.array([fitness\_function(cell) for cell in cells])

# Update states

new\_cells = np.copy(cells)

for i in range(num\_cells):

# Get neighbors

neighbors = cells[max(0, i-neighborhood\_size):min(num\_cells, i+neighborhood\_size+1)]

# Update cell based on neighbors

new\_cells[i] = np.mean(neighbors) + np.random.uniform(-0.1, 0.1) # Add some noise

cells = new\_cells

# Output the best solution

best\_cell = cells[np.argmin(fitness)]

print(f"Best solution found: {best\_cell}")

print(f"Fitness: {fitness\_function(best\_cell)}")

OUTPUT:

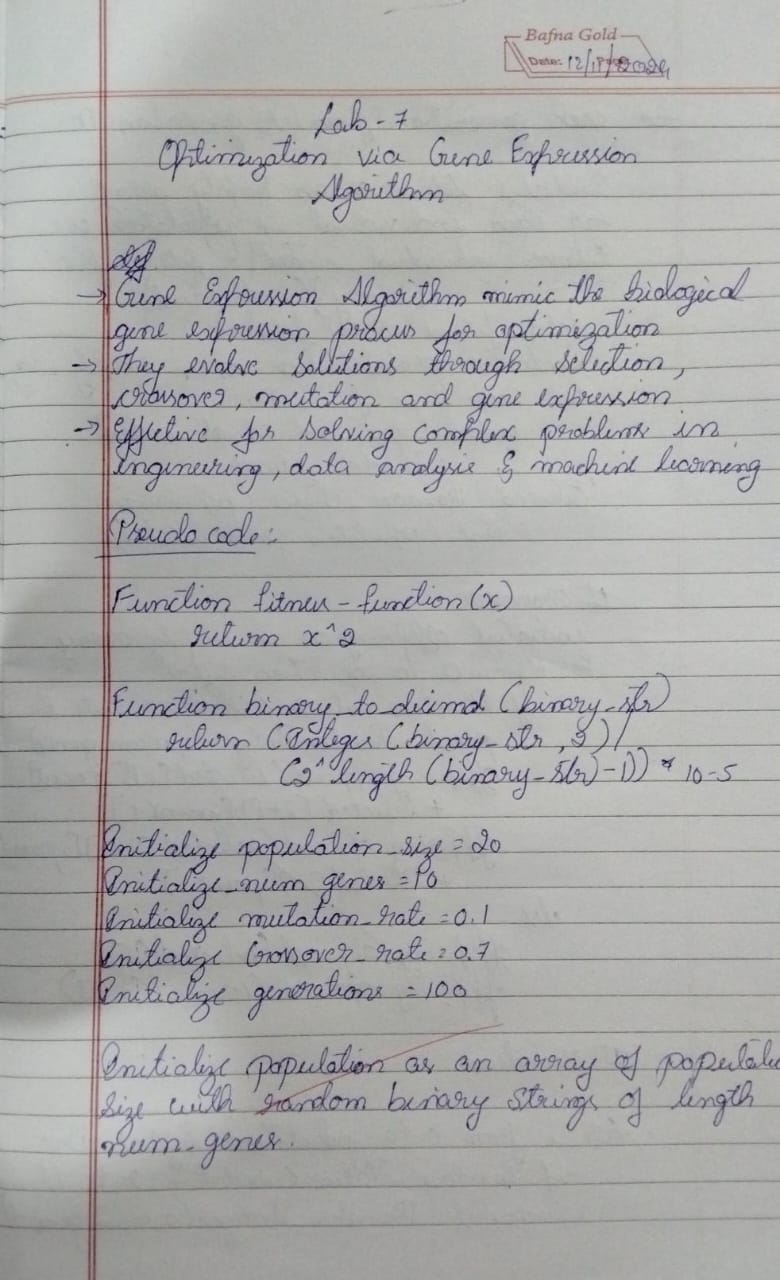
Best solution found: 0.07514167362240409

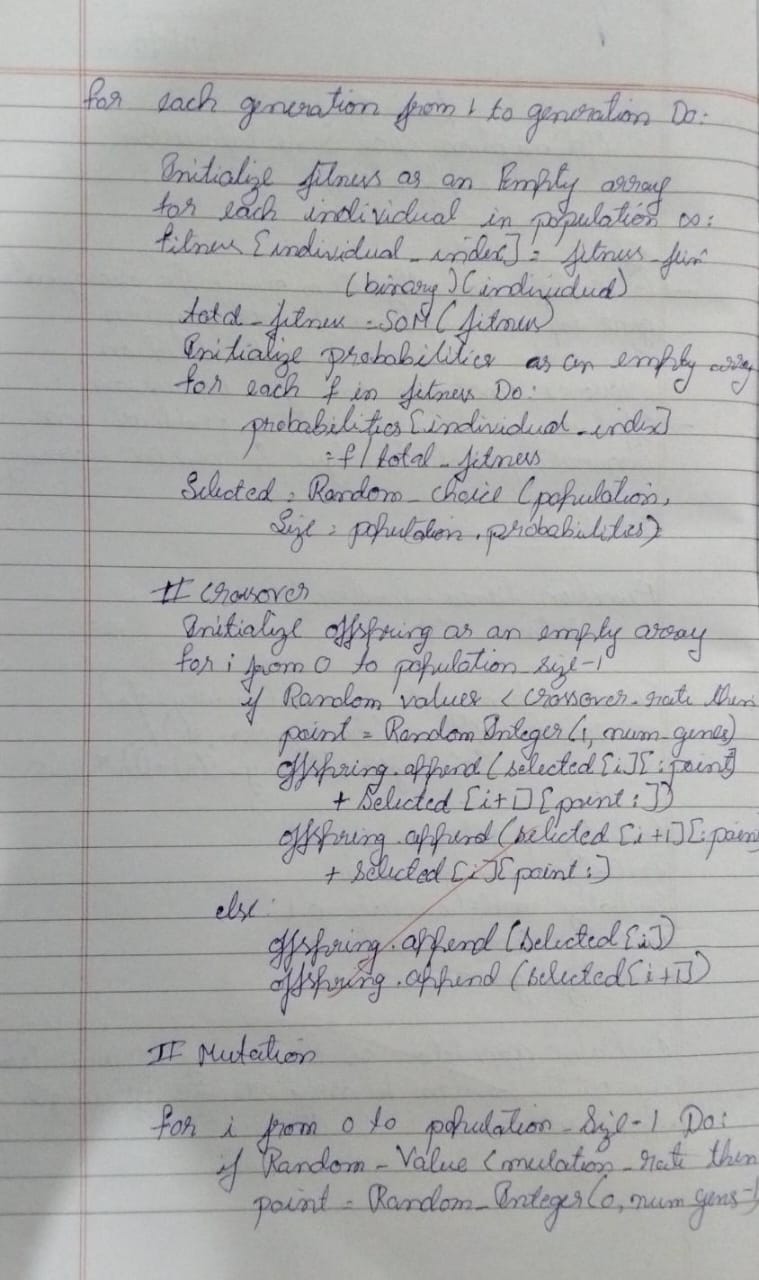
Fitness: 0.005646271114775899

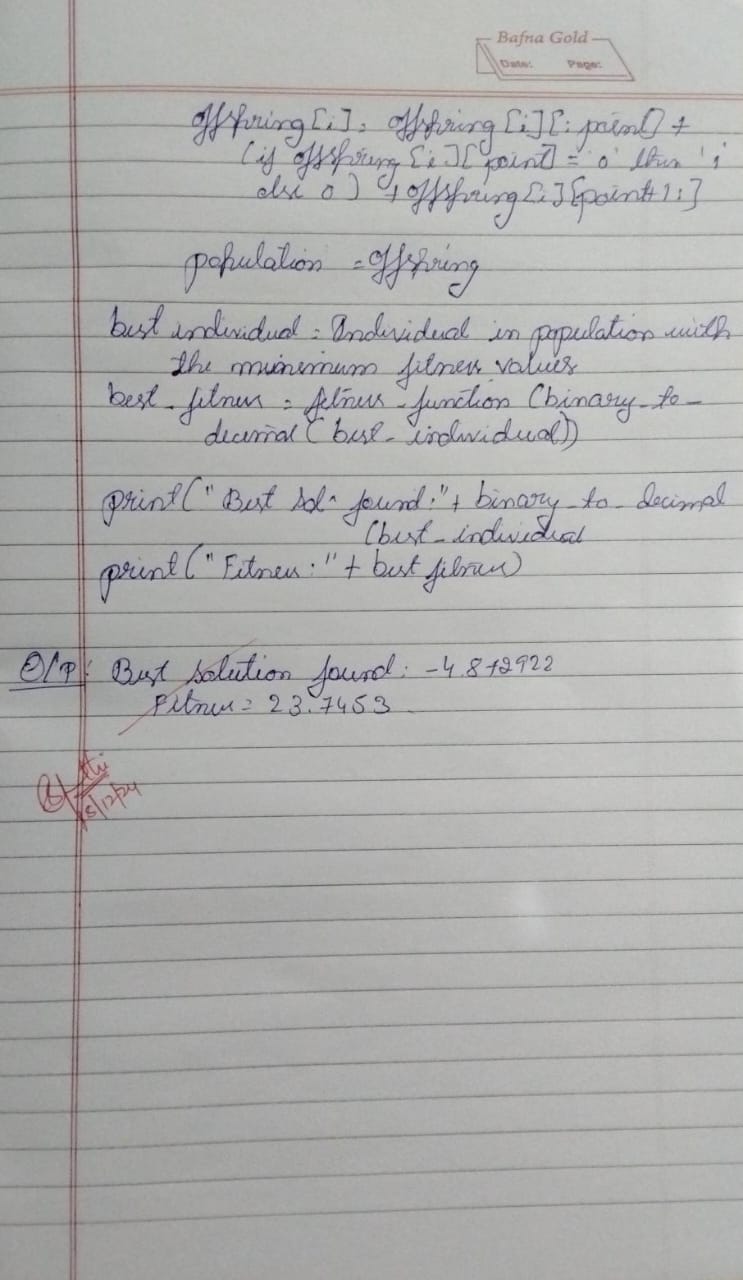
**Program 7**

Optimization via Gene Expression

Algorithm :







Code:

#Optimization via Gene Expression Algorithms

import numpy as np

# Define the optimization function

def fitness\_function(x):

return x\*\*2

# Convert binary string to decimal

def binary\_to\_decimal(binary\_str):

return int(binary\_str, 2) / (2\*\*len(binary\_str) - 1) \* 10 - 5 # Scale to [-5, 5]

# Initialize parameters

population\_size = 20

num\_genes = 10

mutation\_rate = 0.1

crossover\_rate = 0.7

generations = 100

# Initialize population

population = [''.join(np.random.choice(['0', '1'], num\_genes)) for \_ in range(population\_size)]

# Main loop

for \_ in range(generations):

# Evaluate fitness

fitness = [fitness\_function(binary\_to\_decimal(ind)) for ind in population]

# Selection (roulette wheel)

total\_fitness = sum(fitness)

probabilities = [f / total\_fitness for f in fitness]

selected = np.random.choice(population, size=population\_size, p=probabilities)

# Crossover

offspring = []

for i in range(0, population\_size, 2):

if np.random.rand() < crossover\_rate:

point = np.random.randint(1, num\_genes)

offspring.append(selected[i][:point] + selected[i+1][point:])

offspring.append(selected[i+1][:point] + selected[i][point:])

else:

offspring.append(selected[i])

offspring.append(selected[i+1])

# Mutation

for i in range(population\_size):

if np.random.rand() < mutation\_rate:

point = np.random.randint(num\_genes)

offspring[i] = offspring[i][:point] + ('1' if offspring[i][point] == '0' else '0') + offspring[i][point+1:]

population = offspring

# Output the best solution

best\_individual = min(population, key=lambda ind: fitness\_function(binary\_to\_decimal(ind)))

best\_fitness = fitness\_function(binary\_to\_decimal(best\_individual))

print(f"Best solution found: {binary\_to\_decimal(best\_individual)}")

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

OUTPUT :

Best solution found: 4.247311827956988

Fitness: 18.03965776390333