|  |
| --- |
| **VISVESVARAYA TECHNOLOGICAL UNIVERSITY**  **“JnanaSangama”, Belgaum -590014, Karnataka.**    **LAB RECORD**  **Bio Inspired Systems (23CS5BSBIS)**  ***Submitted by***  **SOHAN T SANJEEV (1BM23CS421)**  ***in partial fulfilment 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 **SOHAN T SANJEEV (1BM23CS421),** 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.

|  |  |
| --- | --- |
| Saritha AN  Assistant Professor  Department of CSE, BMSCE | Dr. Kavitha Sooda Professor & HOD  Department of CSE, BMSCE |

**Index**

|  |  |  |  |
| --- | --- | --- | --- |
| **Sl.**  **No.** | **Date** | **Experiment Title** | **Page No.** |
| 1 | 3-10-24 | Implementation of Genetic Algorithms using Optimization | 4-7 |
| 2 | 25-10-24 | Implementation of Particle Swarm Optimization for Function Optimization | 8-11 |
| 3 | 7-11-24 | Implementation of Ant Colony Algorithm | 12-15 |
| 4 | 14-11-24 | Implementation of Cuckoo Search Algorithm | 16-21 |
| 5 | 21-11-24 | Implementation of Grey Wolf Search Optimization Algorithm | 22-25 |
| 6 | 28-11-24 | Implementation of Parallel Cellular Algorithms and Programs | 26-30 |
| 7 | 18-12-24 | Implementation of **Gene Expression Algorithms (GEA)** | 31-36 |

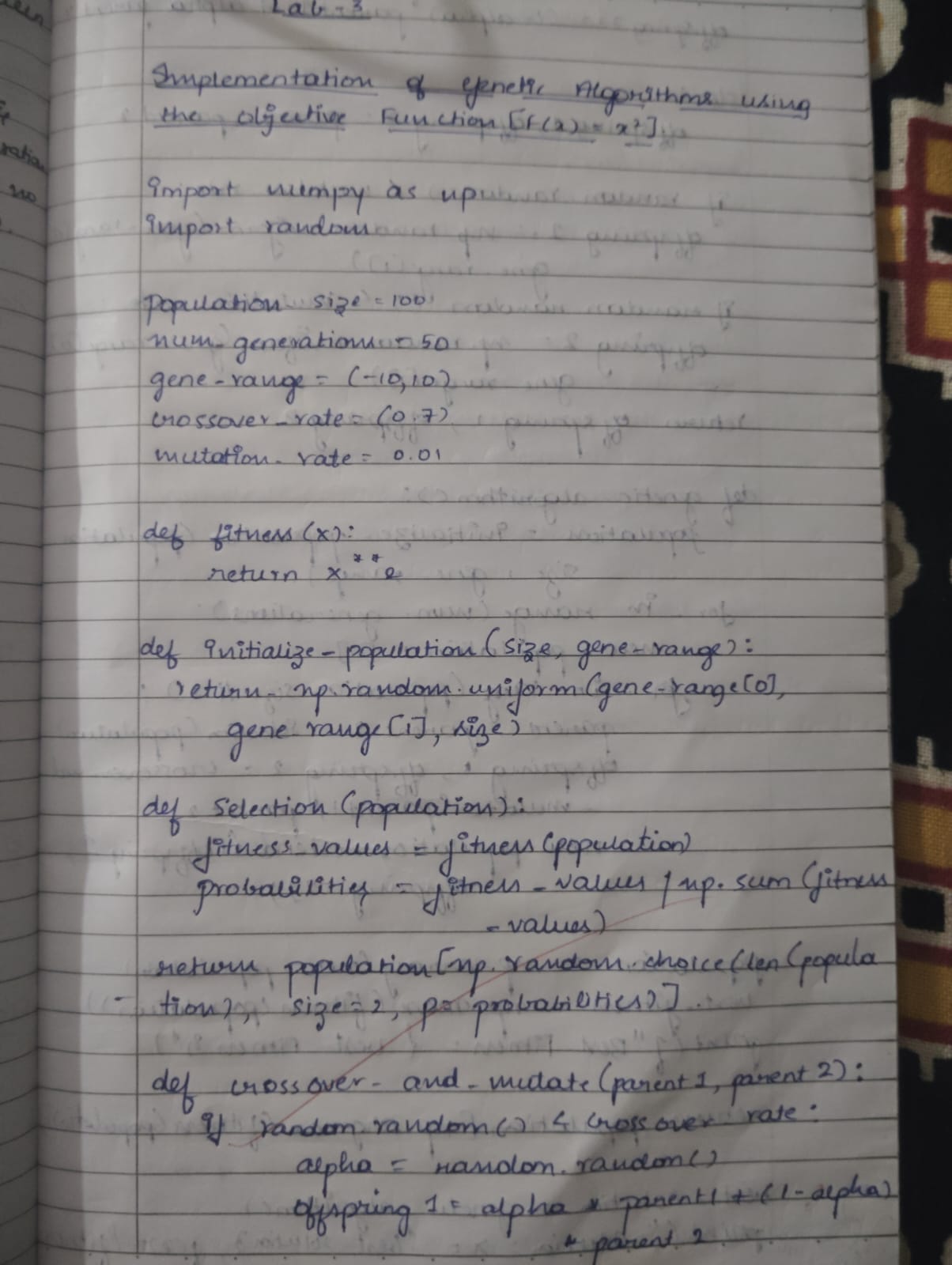
GitHub Link:

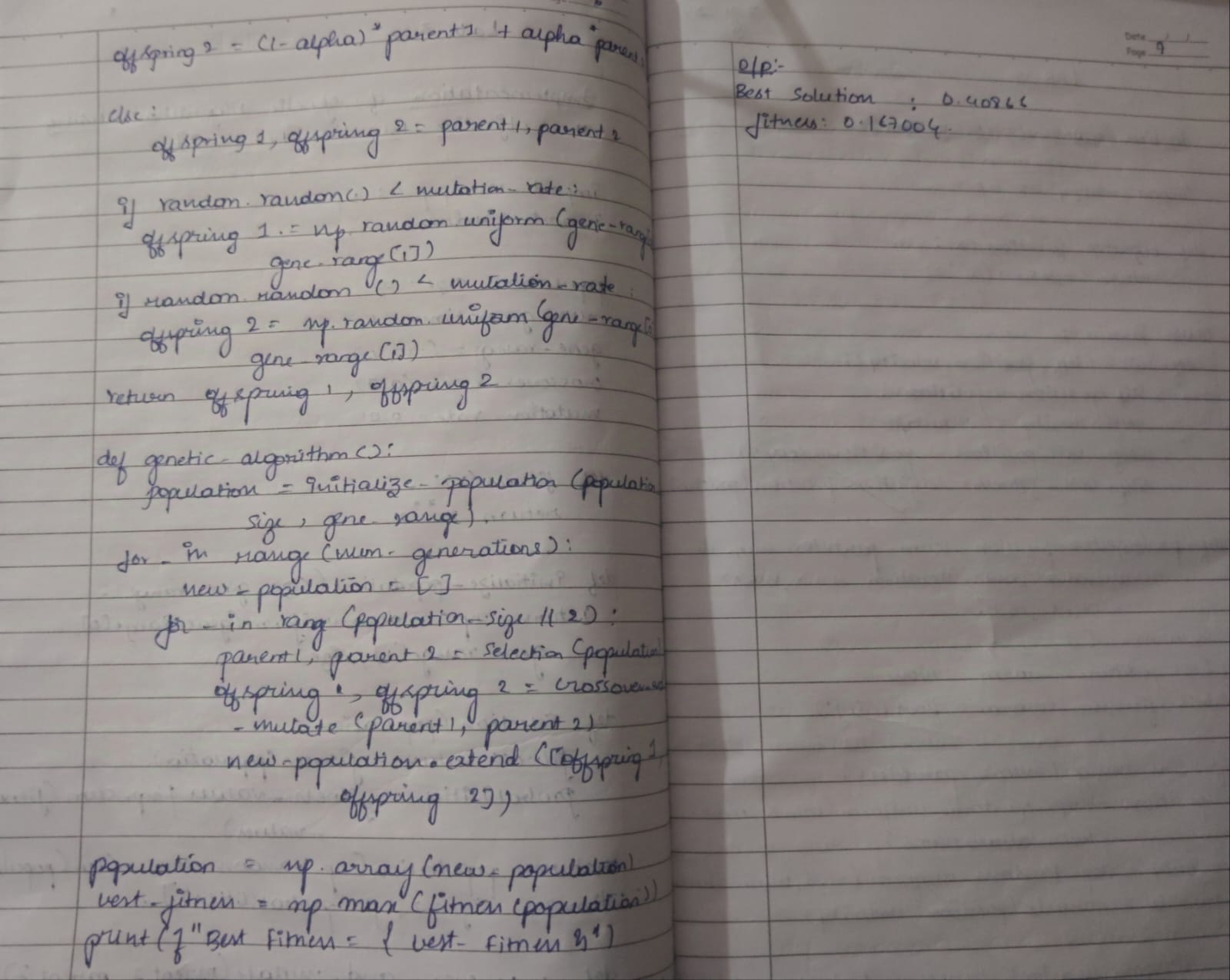
<https://github.com/SohanTbmsce/5E---BIS>

**Program 1**

Implementation of Genetic Algorithms using Optimization

Algorithm:





Code:

#LAB-3:GENETIC ALGORITHM USING OPTIMIZATION

import numpy as np

import random

population\_size = 50

num\_generations = 50

gene\_range = (-10, 10)

crossover\_rate = 0.7

mutation\_rate = 0.01

def fitness(x):

    return x \*\* 2

def initialize\_population(size, gene\_range):

    return np.random.uniform(gene\_range[0], gene\_range[1], size)

def selection(population):

    fitness\_values = fitness(population)

    probabilities = fitness\_values / np.sum(fitness\_values)

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

def crossover\_and\_mutate(parent1, parent2):

    if random.random() < crossover\_rate:

        alpha = random.random()

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

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

    else:

        offspring1, offspring2 = parent1, parent2

    if random.random() < mutation\_rate:

        offspring1 = np.random.uniform(gene\_range[0], gene\_range[1])

    if random.random() < mutation\_rate:

        offspring2 = np.random.uniform(gene\_range[0], gene\_range[1])

    return offspring1, offspring2

def genetic\_algorithm():

    population = initialize\_population(population\_size, gene\_range)

    for \_ in range(num\_generations):

        new\_population = []

        for \_ in range(population\_size // 2):

            parent1, parent2 = selection(population)

            offspring1, offspring2 = crossover\_and\_mutate(parent1, parent2)

            new\_population.extend([offspring1, offspring2])

        population = np.array(new\_population)

        best\_fitness = np.max(fitness(population))

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

    return population[np.argmax(fitness(population))]

best\_solution = genetic\_algorithm()print(f"The best solution found: x = {best\_solution}, f(x) = {fitness(best\_solution)}")

Output:

Best Fitness = 96.41554825031078

Best Fitness = 96.41554825031078

Best Fitness = 93.77197105788775

Best Fitness = 93.77197105788775

Best Fitness = 89.75890201273582

Best Fitness = 89.75890201273582

Best Fitness = 89.75890201273582

Best Fitness = 86.28500889461726

Best Fitness = 85.46826382132502

Best Fitness = 85.46826382132502

Best Fitness = 83.92324509711497

Best Fitness = 82.56253766252124

Best Fitness = 81.54296033544576

Best Fitness = 80.93166255477712

Best Fitness = 80.93166255477712

Best Fitness = 98.63638625600349

Best Fitness = 95.71847971648907

Best Fitness = 95.71847971648907

Best Fitness = 78.67688660267774

Best Fitness = 78.42074885686151

Best Fitness = 78.40955357287604

Best Fitness = 78.2163163886333

Best Fitness = 78.2163163886333

Best Fitness = 77.87139603278503

Best Fitness = 77.83602785214866

Best Fitness = 77.73510449497564

Best Fitness = 77.5393503439129

Best Fitness = 80.56823420832076

Best Fitness = 77.46782583009356

Best Fitness = 77.46782583009356

Best Fitness = 77.46782583009356

Best Fitness = 77.4087588279005

Best Fitness = 77.38033782912058

Best Fitness = 77.38033782912058

Best Fitness = 77.37443058909962

Best Fitness = 77.36238137273801

Best Fitness = 77.35009480792148

Best Fitness = 77.33394495335988

Best Fitness = 77.33246445407919

Best Fitness = 77.33246445407919

Best Fitness = 77.32984862750179

Best Fitness = 77.30828605125325

Best Fitness = 77.26925723673783

Best Fitness = 77.17293761811123

Best Fitness = 77.13725577017766

Best Fitness = 77.13725577017766

Best Fitness = 77.1084152222153

Best Fitness = 77.1084152222153

Best Fitness = 76.9472934808921

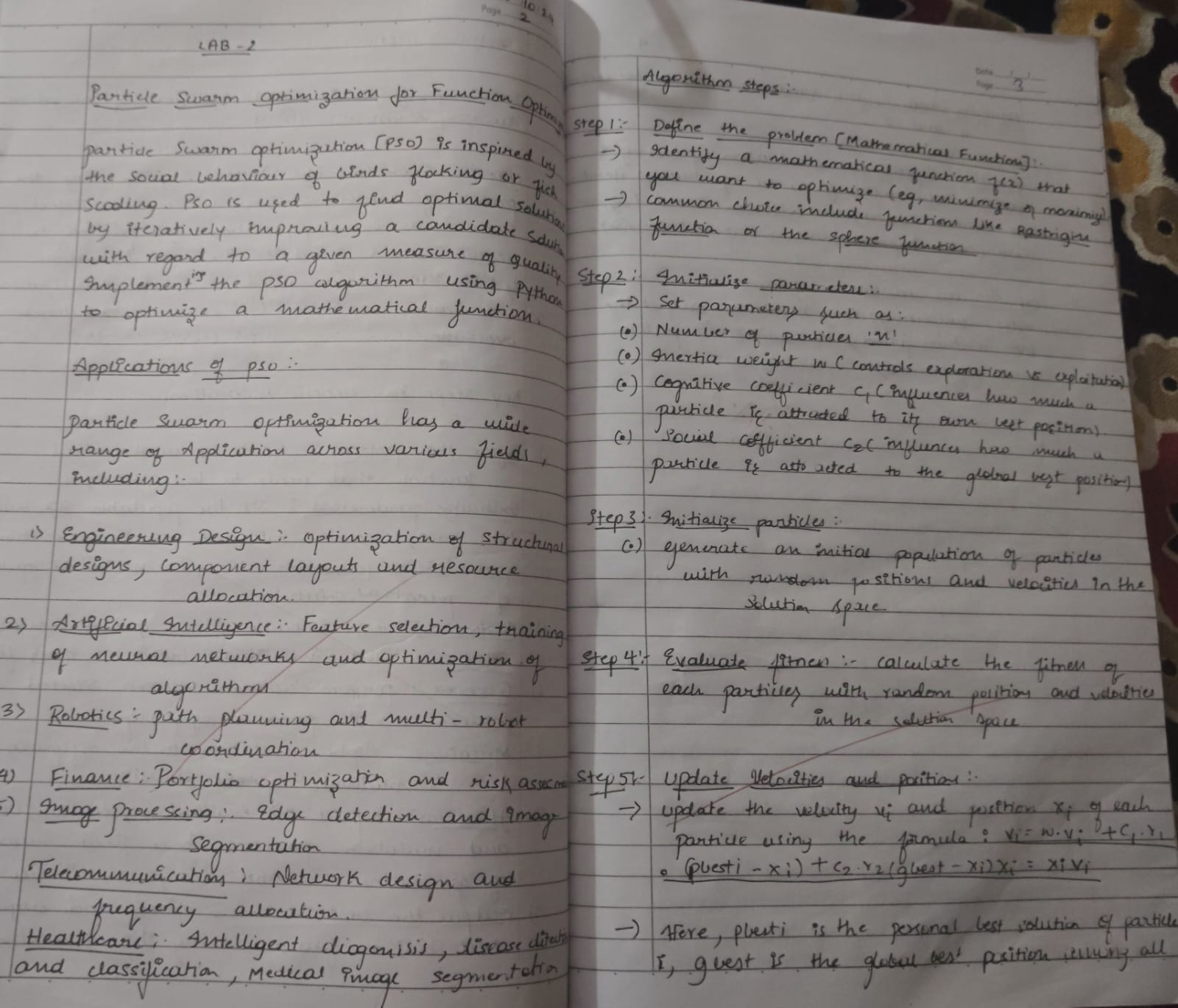
Best Fitness = 76.93707640654297

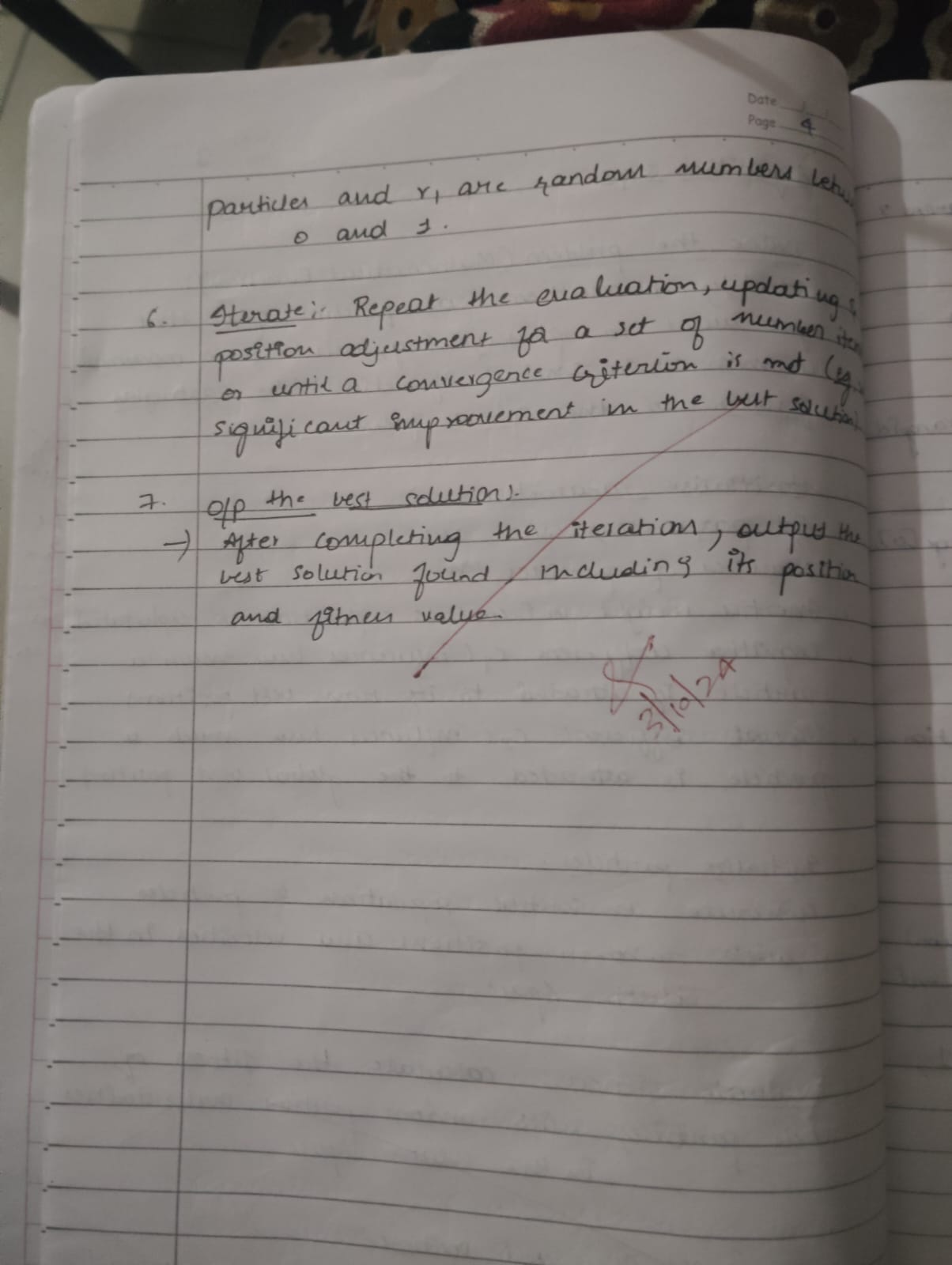
The best solution found: x = 8.771378250112292, f(x) = 76.93707640654297

**Program 2**

Implementation of Particle Swarm Optimization for Function Optimization

Algorithm:





Code:

#LAB-4:PARTICLE SWARM OPTIMIZATION

import numpy as np

# Objective function (example: Sphere function)

def objective\_function(x):

    return np.sum(x\*\*2)

# Particle class

class Particle:

    def \_\_init\_\_(self, position, velocity):

        self.position = position

        self.velocity = velocity

        self.best\_position = position.copy()

        self.best\_fitness = objective\_function(position)

# Particle Swarm Optimization (PSO) function

def pso(objective\_function, num\_particles, num\_dimensions, max\_iter, minx, maxx, w, c1, c2):

    # Initialize the swarm

    swarm = []

    best\_position\_swarm = None

    best\_fitness\_swarm = float('inf')

    # Create particles

    for \_ in range(num\_particles):

        position = np.random.uniform(minx, maxx, num\_dimensions)

        velocity = np.random.uniform(-1, 1, num\_dimensions)

        particle = Particle(position, velocity)

        # Update global best if this particle has a better fitness

        if particle.best\_fitness < best\_fitness\_swarm:

            best\_fitness\_swarm = particle.best\_fitness

            best\_position\_swarm = particle.best\_position.copy()

        swarm.append(particle)

    # PSO main loop

    for iter in range(max\_iter):

        for particle in swarm:

            # Calculate random factors

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

            # Update velocity

            inertia = w \* particle.velocity

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

            social = c2 \* r2 \* (best\_position\_swarm - particle.position)

            particle.velocity = inertia + cognitive + social

            # Update position

            particle.position += particle.velocity

            # Apply position boundaries

            particle.position = np.clip(particle.position, minx, maxx)

            # Evaluate fitness of the new position

            fitness = objective\_function(particle.position)

            # Update the particle's best fitness and position if needed

            if fitness < particle.best\_fitness:

                particle.best\_fitness = fitness

                particle.best\_position = particle.position.copy()

            # Update global best fitness and position if needed

            if fitness < best\_fitness\_swarm:

                best\_fitness\_swarm = fitness

                best\_position\_swarm = particle.position.copy()

    # Return the best particle of the swarm

    return best\_position\_swarm, best\_fitness\_swarm

# PSO Hyperparameters

num\_particles = 30

num\_dimensions = 2

max\_iter = 100

minx, maxx = -10, 10

w = 0.5      # Inertia weight

c1 = 1.5     # Cognitive (particle's own best) weight

c2 = 1.5     # Social (swarm's best) weight

# Run PSO

best\_position, best\_fitness = pso(objective\_function, num\_particles, num\_dimensions, max\_iter, minx, maxx, w, c1, c2)

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

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

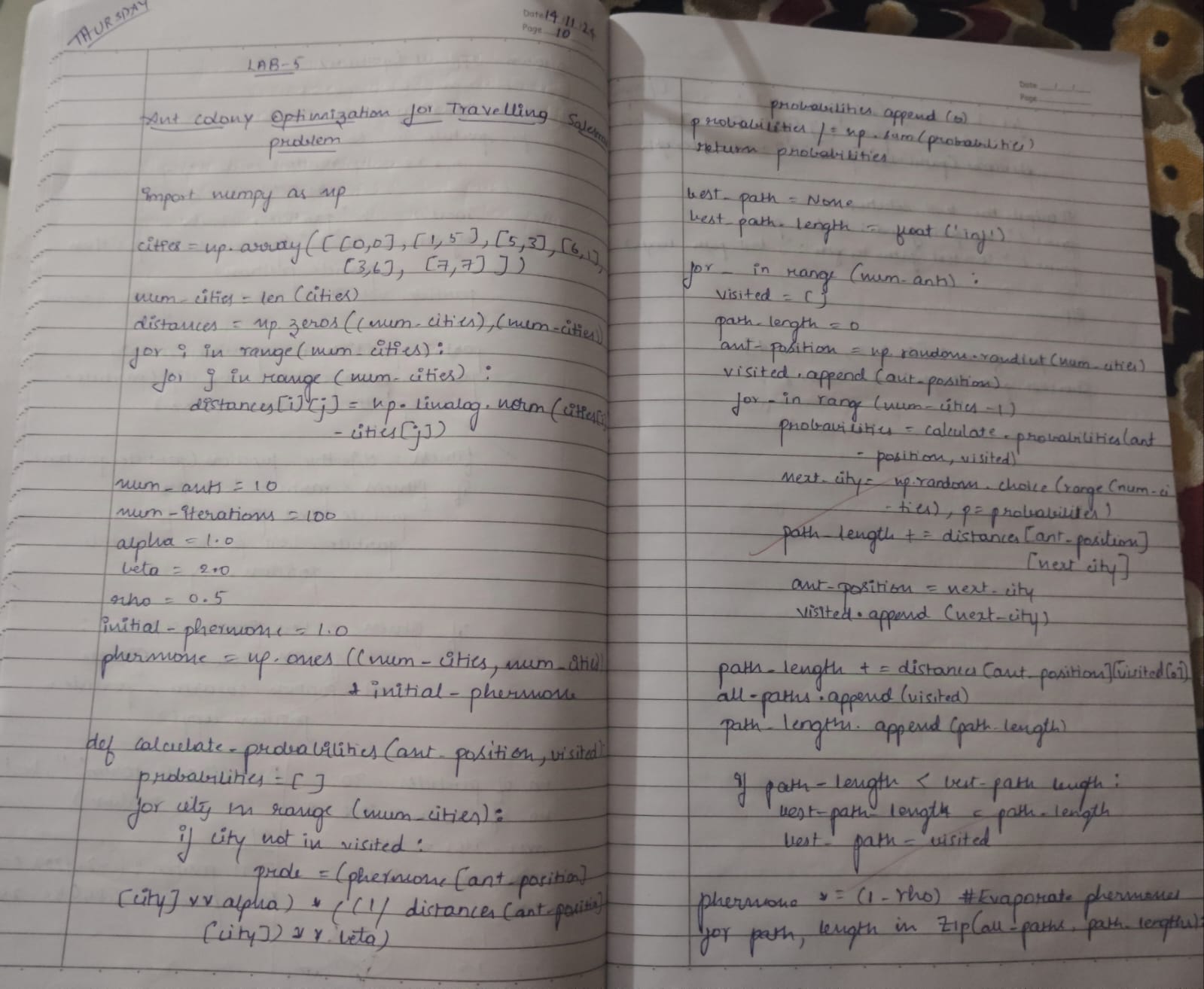
Output:

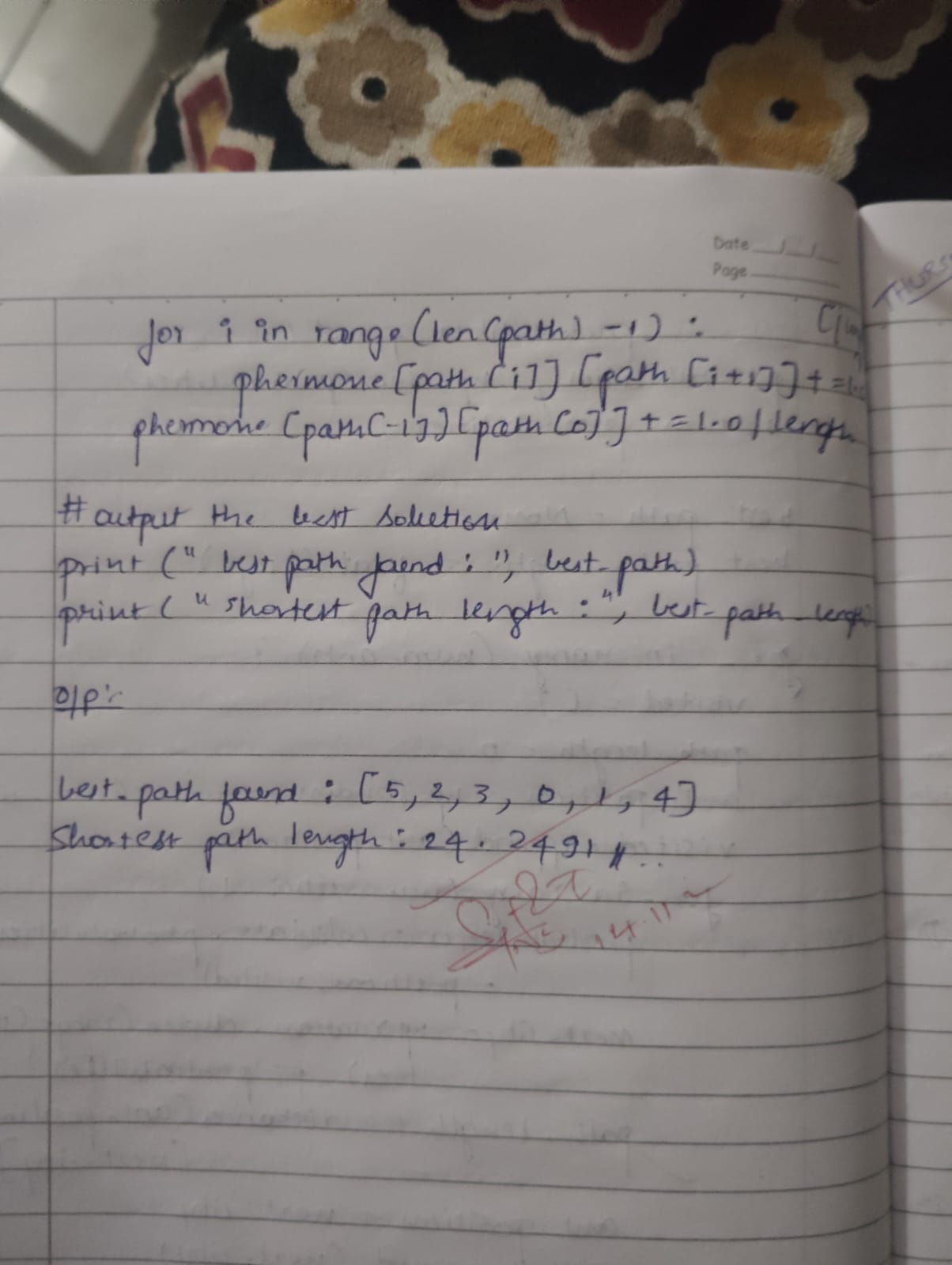
Best Position: [-7.11248266e-13 -6.05931834e-13]BestFitnesss8.730274834800181e

**Program 3**

Implementation of Ant Colony Algorithm

Algorithm:





Code:

 #LAB-5:ANT COLONY ORGANIZATION

import numpy as np

cities = np.array([

    [0, 0], [1, 5], [5, 3], [6, 1], [3, 6], [7, 7]

])

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])

num\_ants = 10

num\_iterations = 100

alpha = 1.0

beta = 2.0

rho = 0.5

initial\_pheromone = 1.0

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

def calculate\_probabilities(ant\_position, visited):

    probabilities = []

    for city in range(num\_cities):

        if city not in visited:

            prob = (pheromone[ant\_position][city] \*\* alpha) \* ((1 / distances[ant\_position][city]) \*\* beta)

            probabilities.append(prob)

        else:

            probabilities.append(0)

    probabilities /= np.sum(probabilities)

    return probabilities

best\_path = None

best\_path\_length = float('inf')

for \_ in range(num\_iterations):

    all\_paths = []

    path\_lengths = []

    for ant in range(num\_ants):

        visited = []

        path\_length = 0

        ant\_position = np.random.randint(num\_cities)  # Start at a random city

        visited.append(ant\_position)

        for \_ in range(num\_cities - 1):

            probabilities = calculate\_probabilities(ant\_position, visited)

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

            path\_length += distances[ant\_position][next\_city]

            ant\_position = next\_city

            visited.append(next\_city)

        path\_length += distances[ant\_position][visited[0]]  # Return to start

        all\_paths.append(visited)

        path\_lengths.append(path\_length)

        # Update best path

        if path\_length < best\_path\_length:

            best\_path\_length = path\_length

            best\_path = visited

    pheromone \*= (1 - rho)

    for path, length in zip(all\_paths, path\_lengths):

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

            pheromone[path[i]][path[i+1]] += 1.0 / length

        pheromone[path[-1]][path[0]] += 1.0 / length  # Complete the cycle

print("Best path found:", best\_path)

print("Shortest path length:", best\_path\_length)

Output:

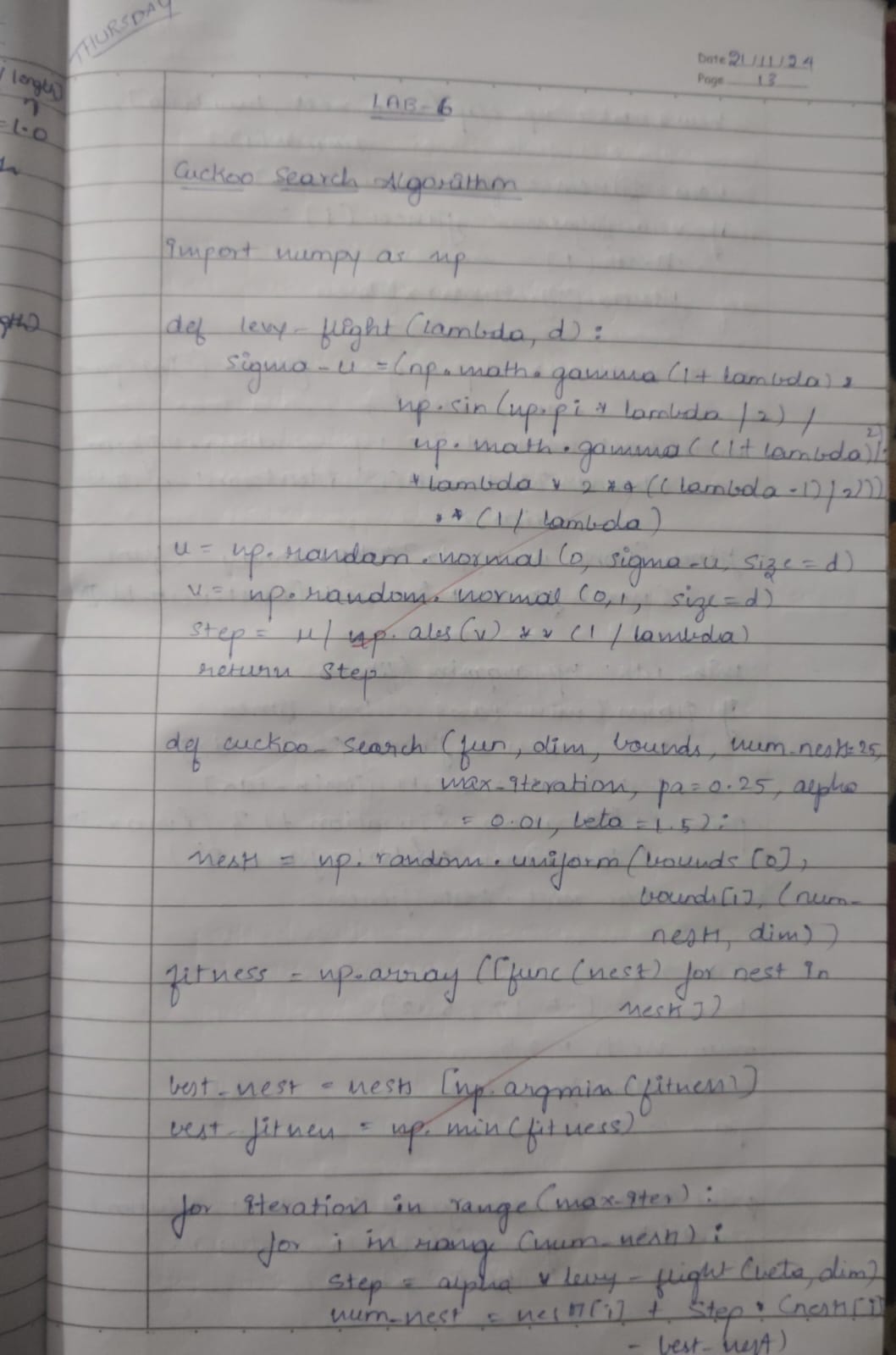
Best path found: [5, 2, 3, 0, 1, 4]

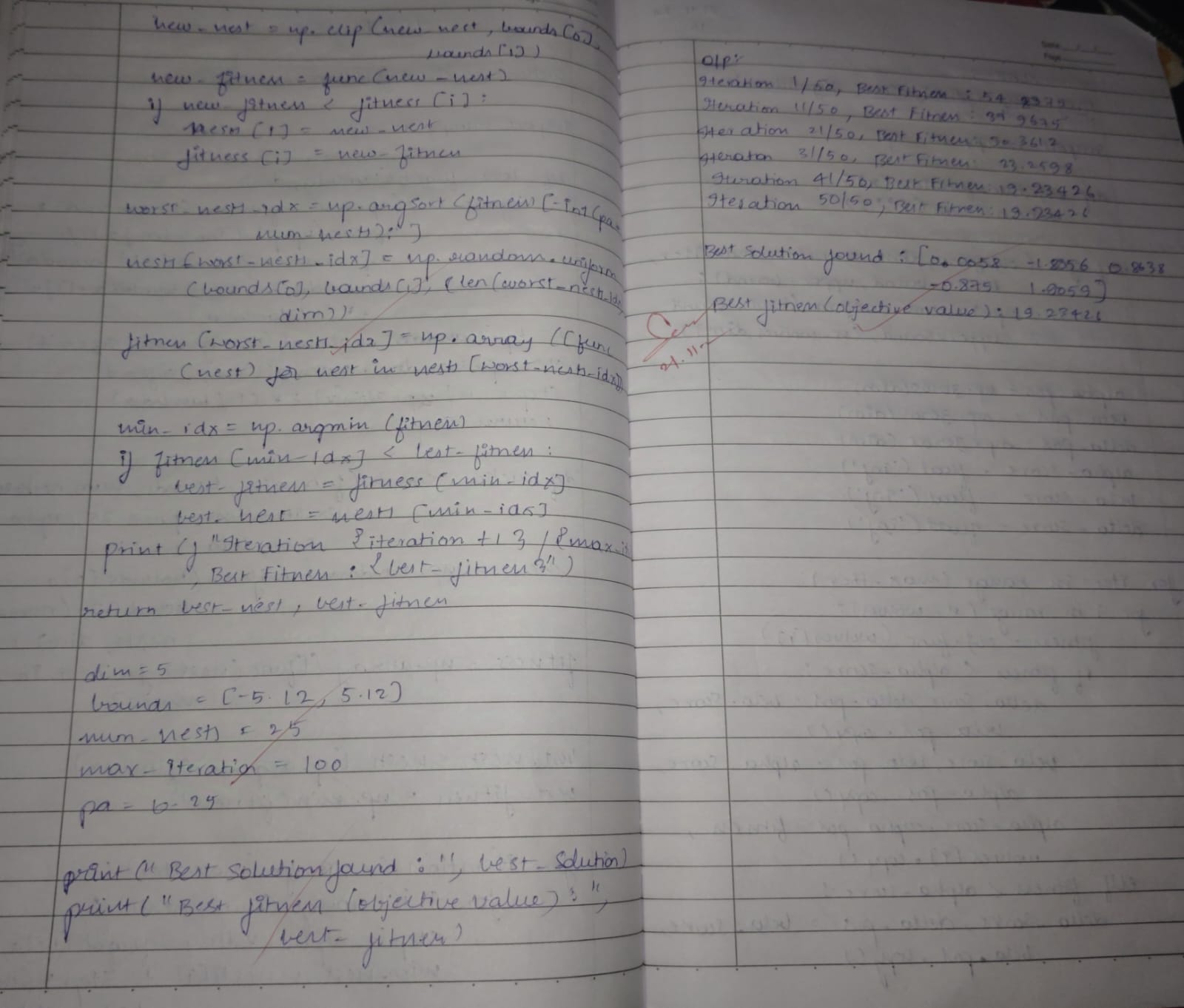
Shortest path length: 24.249159579507822

**Program 4**

Implementation of Cuckoo Search Algorithm

Algorithm:





Code:

#LAB-6:CUCKOO SEARCH ALGORITHM

import numpy as np

# Define the Rastrigin function (used for optimization problems)

def rastrigin(x):

    A = 10

    return A \* len(x) + sum(x\*\*2 - A \* np.cos(2 \* np.pi \* x))

# Levy Flight function

def levy\_flight(Lambda, d):

    # Generate Lévy flights

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

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

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

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

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

    return step

# Cuckoo Search Algorithm

def cuckoo\_search(func, dim, bounds, num\_nests=25, max\_iter=100, pa=0.25, alpha=0.01, beta=1.5):

    # Initial population (random solutions within bounds)

    nests = np.random.uniform(bounds[0], bounds[1], (num\_nests, dim))

    fitness = np.array([func(nest) for nest in nests])

    # Best nest found so far

    best\_nest = nests[np.argmin(fitness)]

    best\_fitness = np.min(fitness)

    # Main loop

    for iteration in range(max\_iter):

        # Generate new solutions using Lévy flights

        for i in range(num\_nests):

            step = alpha \* levy\_flight(beta, dim)

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

            # Bound checking (to ensure new nests are within bounds)

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

            # Calculate fitness of the new nest

            new\_fitness = func(new\_nest)

            # If new solution is better, replace old nest

            if new\_fitness < fitness[i]:

                nests[i] = new\_nest

                fitness[i] = new\_fitness

        # Abandon the worst nests and replace them with new random ones

        worst\_nests\_idx = np.argsort(fitness)[-int(pa \* num\_nests):]

        nests[worst\_nests\_idx] = np.random.uniform(bounds[0], bounds[1], (len(worst\_nests\_idx), dim))

        fitness[worst\_nests\_idx] = np.array([func(nest) for nest in nests[worst\_nests\_idx]])

        # Update the best nest

        min\_idx = np.argmin(fitness)

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

            best\_fitness = fitness[min\_idx]

            best\_nest = nests[min\_idx]

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

    return best\_nest, best\_fitness

# Define problem bounds and parameters

dim = 5  # Problem dimensionality (e.g., 5-dimensional problem)

bounds = [-5.12, 5.12]  # Bounds of the search space (for Rastrigin function)

num\_nests = 25  # Number of nests (solutions)

max\_iter = 100  # Maximum number of iterations

pa = 0.25  # Probability of discovering a nest (fraction of worst nests to abandon)

# Run the Cuckoo Search algorithm

best\_solution, best\_fitness = cuckoo\_search(rastrigin, dim, bounds, num\_nests, max\_iter, pa)

print("Best solution found: ", best\_solution)

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

Output:

<ipython-input-1-48ab2d5bdd0a>:11: DeprecationWarning: `np.math` is a deprecated alias for the standard library `math` module (Deprecated Numpy 1.25). Replace usages of `np.math` with `math`

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

<ipython-input-1-48ab2d5bdd0a>:12: DeprecationWarning: `np.math` is a deprecated alias for the standard library `math` module (Deprecated Numpy 1.25). Replace usages of `np.math` with `math`

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

Iteration 1/100, Best Fitness: 34.10283829263206

Iteration 2/100, Best Fitness: 34.10283829263206

Iteration 3/100, Best Fitness: 34.10283829263206

Iteration 4/100, Best Fitness: 34.10283829263206

Iteration 5/100, Best Fitness: 34.10283829263206

Iteration 6/100, Best Fitness: 34.10283829263206

Iteration 7/100, Best Fitness: 34.10283829263206

Iteration 8/100, Best Fitness: 34.10283829263206

Iteration 9/100, Best Fitness: 34.10283829263206

Iteration 10/100, Best Fitness: 34.10283829263206

Iteration 11/100, Best Fitness: 34.10283829263206

Iteration 12/100, Best Fitness: 34.10283829263206

Iteration 13/100, Best Fitness: 34.10283829263206

Iteration 14/100, Best Fitness: 34.10283829263206

Iteration 15/100, Best Fitness: 34.10283829263206

Iteration 16/100, Best Fitness: 34.10283829263206

Iteration 17/100, Best Fitness: 34.10283829263206

Iteration 18/100, Best Fitness: 34.10283829263206

Iteration 19/100, Best Fitness: 34.10283829263206

Iteration 20/100, Best Fitness: 34.10283829263206

Iteration 21/100, Best Fitness: 34.10283829263206

Iteration 22/100, Best Fitness: 34.10283829263206

Iteration 23/100, Best Fitness: 34.10283829263206

Iteration 24/100, Best Fitness: 34.10283829263206

Iteration 25/100, Best Fitness: 34.10283829263206

Iteration 26/100, Best Fitness: 29.34980262200628

Iteration 27/100, Best Fitness: 29.34980262200628

Iteration 28/100, Best Fitness: 29.34980262200628

Iteration 29/100, Best Fitness: 23.133784372306057

Iteration 30/100, Best Fitness: 23.133784372306057

Iteration 31/100, Best Fitness: 23.133784372306057

Iteration 32/100, Best Fitness: 23.133784372306057

Iteration 33/100, Best Fitness: 23.133784372306057

Iteration 34/100, Best Fitness: 23.133784372306057

Iteration 35/100, Best Fitness: 23.133784372306057

Iteration 36/100, Best Fitness: 23.133784372306057

Iteration 37/100, Best Fitness: 23.133784372306057

Iteration 38/100, Best Fitness: 22.800741517549888

Iteration 39/100, Best Fitness: 22.800741517549888

Iteration 40/100, Best Fitness: 22.800741517549888

Iteration 41/100, Best Fitness: 22.800741517549888

Iteration 42/100, Best Fitness: 22.800741517549888

Iteration 43/100, Best Fitness: 21.978739473969867

Iteration 44/100, Best Fitness: 21.331310300542093

Iteration 45/100, Best Fitness: 21.134160680988344

Iteration 46/100, Best Fitness: 21.134160680988344

Iteration 47/100, Best Fitness: 21.112031250864927

Iteration 48/100, Best Fitness: 20.22783836901366

Iteration 49/100, Best Fitness: 20.22783836901366

Iteration 50/100, Best Fitness: 20.165109724434036

Iteration 51/100, Best Fitness: 20.165109724434036

Iteration 52/100, Best Fitness: 18.992313289468903

Iteration 53/100, Best Fitness: 18.992313289468903

Iteration 54/100, Best Fitness: 18.992313289468903

Iteration 55/100, Best Fitness: 18.992313289468903

Iteration 56/100, Best Fitness: 18.992313289468903

Iteration 57/100, Best Fitness: 18.992313289468903

Iteration 58/100, Best Fitness: 18.992313289468903

Iteration 59/100, Best Fitness: 16.345713901046942

Iteration 60/100, Best Fitness: 16.345713901046942

Iteration 61/100, Best Fitness: 16.345713901046942

Iteration 62/100, Best Fitness: 16.345713901046942

Iteration 63/100, Best Fitness: 16.345713901046942

Iteration 64/100, Best Fitness: 16.345713901046942

Iteration 65/100, Best Fitness: 16.345713901046942

Iteration 66/100, Best Fitness: 16.345713901046942

Iteration 67/100, Best Fitness: 16.345713901046942

Iteration 68/100, Best Fitness: 16.345713901046942

Iteration 69/100, Best Fitness: 16.345713901046942

Iteration 70/100, Best Fitness: 16.345713901046942

Iteration 71/100, Best Fitness: 16.126755706184014

Iteration 72/100, Best Fitness: 15.66247782758252

Iteration 73/100, Best Fitness: 15.66247782758252

Iteration 74/100, Best Fitness: 15.213018385737115

Iteration 75/100, Best Fitness: 15.213018385737115

Iteration 76/100, Best Fitness: 15.213018385737115

Iteration 77/100, Best Fitness: 14.351369913536452

Iteration 78/100, Best Fitness: 14.16685937126772

Iteration 79/100, Best Fitness: 14.16685937126772

Iteration 80/100, Best Fitness: 14.16685937126772

Iteration 81/100, Best Fitness: 14.16685937126772

Iteration 82/100, Best Fitness: 13.214801041894482

Iteration 83/100, Best Fitness: 13.214801041894482

Iteration 84/100, Best Fitness: 13.214801041894482

Iteration 85/100, Best Fitness: 13.214801041894482

Iteration 86/100, Best Fitness: 13.214801041894482

Iteration 87/100, Best Fitness: 12.893913779918272

Iteration 88/100, Best Fitness: 12.893913779918272

Iteration 89/100, Best Fitness: 12.893913779918272

Iteration 90/100, Best Fitness: 12.893913779918272

Iteration 91/100, Best Fitness: 12.893913779918272

Iteration 92/100, Best Fitness: 12.893913779918272

Iteration 93/100, Best Fitness: 12.705031336984696

Iteration 94/100, Best Fitness: 12.705031336984696

Iteration 95/100, Best Fitness: 12.705031336984696

Iteration 96/100, Best Fitness: 12.705031336984696

Iteration 97/100, Best Fitness: 12.705031336984696

Iteration 98/100, Best Fitness: 12.705031336984696

Iteration 99/100, Best Fitness: 12.705031336984696

Iteration 100/100, Best Fitness: 12.705031336984696

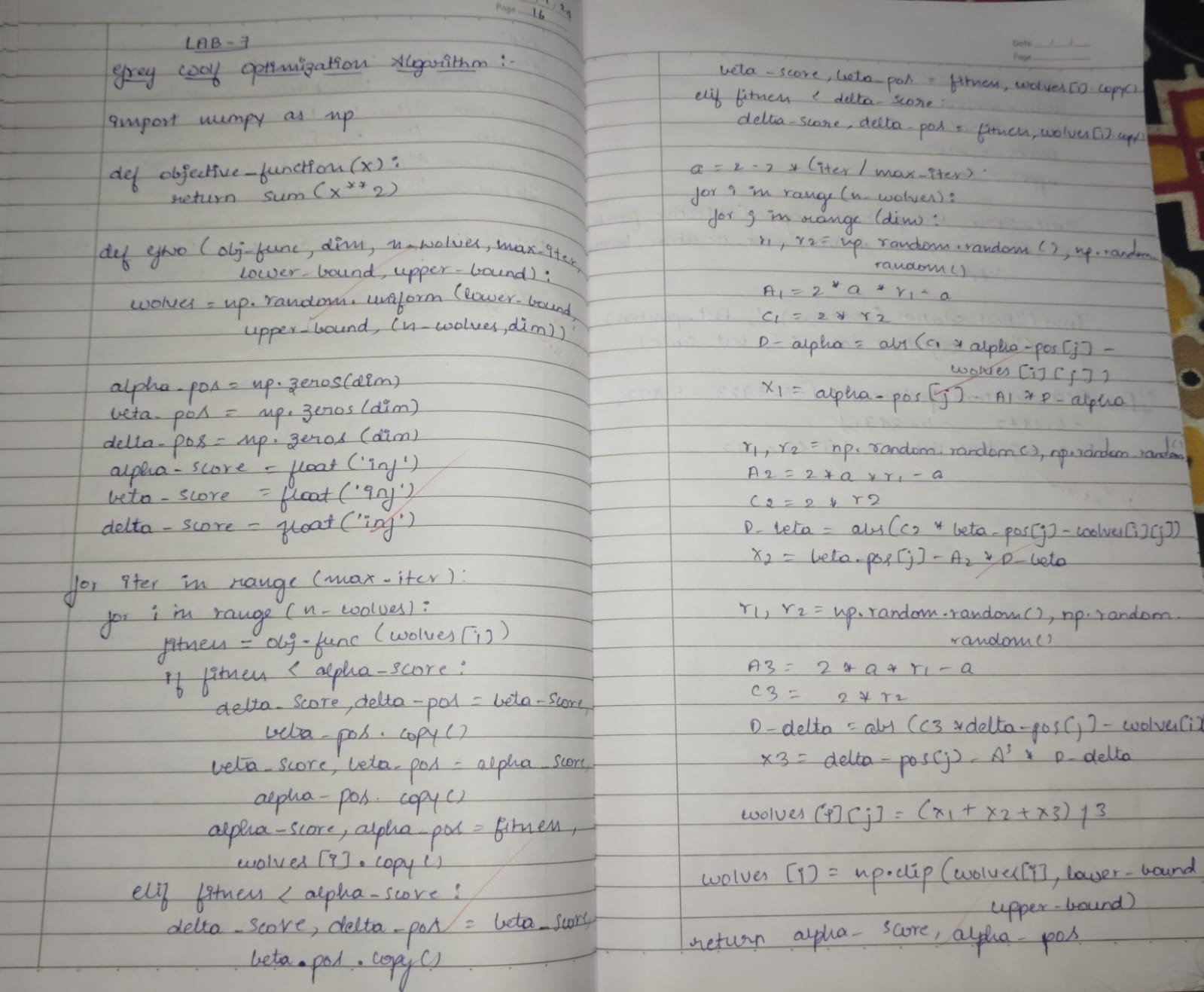
Best solution found: [-1.92879105 1.96608788 -1.02863024 -0.10117856 0.06084245]

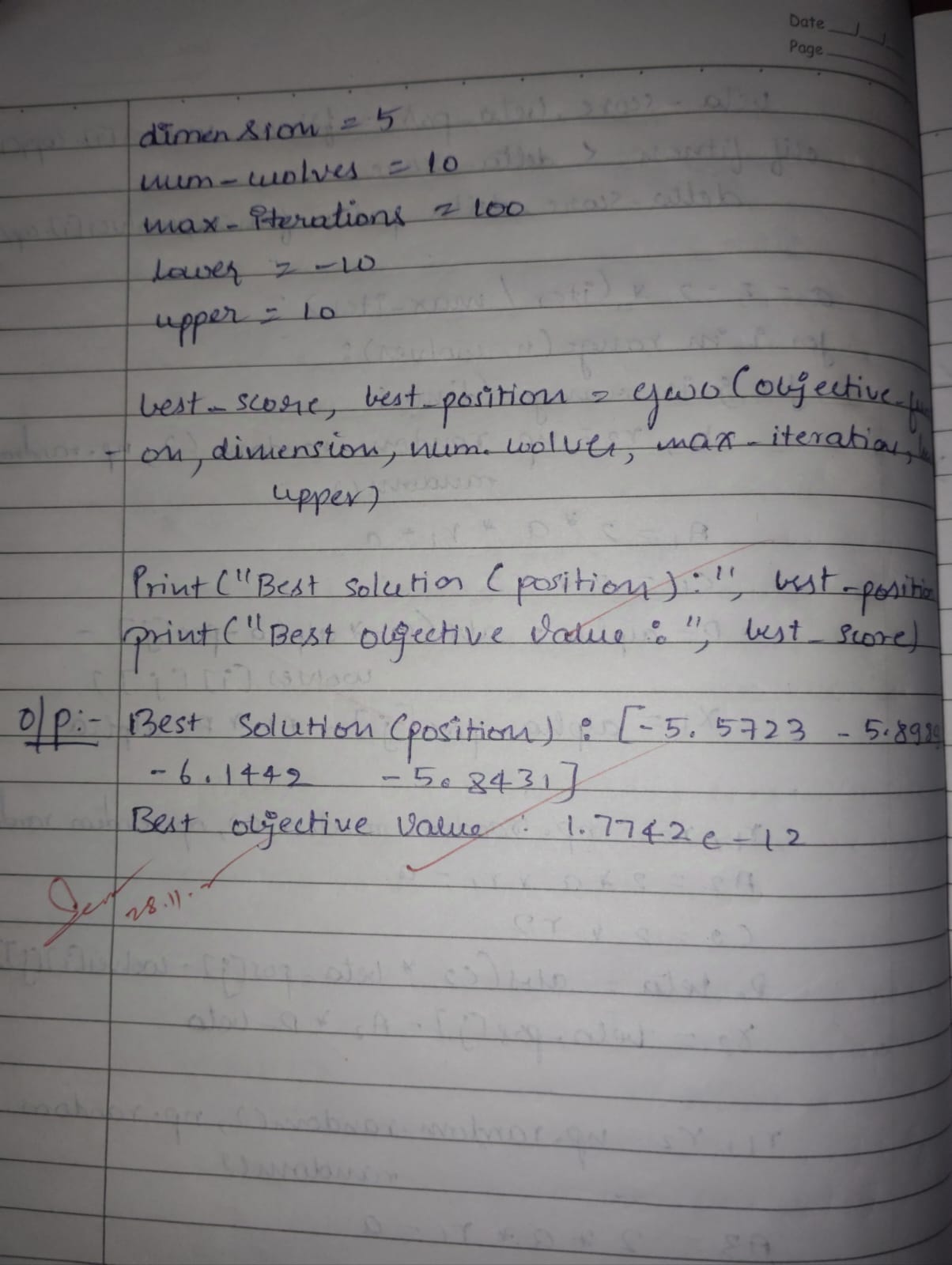
Best fitness (objective value): 12.705031336984696

**Program 5**

Implementation of Grey Wolf Search Optimization Algorithm

Algorithm:





Code:

import numpy as np

# Define the function to optimize (objective function)

def objective\_function(x):

    return sum(x\*\*2)  # Example: minimize the sum of squares of elements

# GWO algorithm implementation

def GWO(obj\_func, dim, n\_wolves, max\_iter, lower\_bound, upper\_bound):

    # Initialize the positions of wolves

    wolves = np.random.uniform(lower\_bound, upper\_bound, (n\_wolves, dim))

    # Initialize alpha, beta, delta wolves (best, second-best, third-best solutions)

    alpha\_pos = np.zeros(dim)

    beta\_pos = np.zeros(dim)

    delta\_pos = np.zeros(dim)

    alpha\_score = float('inf')  # Lowest value of the objective function

    beta\_score = float('inf')

    delta\_score = float('inf')

    # Main iteration loop

    for iter in range(max\_iter):

        # Evaluate the fitness of each wolf

        for i in range(n\_wolves):

            fitness = obj\_func(wolves[i])

            # Update alpha, beta, delta based on fitness

            if fitness < alpha\_score:

                delta\_score, delta\_pos = beta\_score, beta\_pos.copy()

                beta\_score, beta\_pos = alpha\_score, alpha\_pos.copy()

                alpha\_score, alpha\_pos = fitness, wolves[i].copy()

            elif fitness < beta\_score:

                delta\_score, delta\_pos = beta\_score, beta\_pos.copy()

                beta\_score, beta\_pos = fitness, wolves[i].copy()

            elif fitness < delta\_score:

                delta\_score, delta\_pos = fitness, wolves[i].copy()

        # Update the position of each wolf

        a = 2 - 2 \* (iter / max\_iter)  # Linearly decreasing parameter

        for i in range(n\_wolves):

            for j in range(dim):

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

                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

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

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

            # Ensure wolves stay within bounds

            wolves[i] = np.clip(wolves[i], lower\_bound, upper\_bound)

    return alpha\_score, alpha\_pos  # Return the best solution

# Problem definition

dimension = 5

num\_wolves = 10

max\_iterations = 100

lower = -10

upper = 10

# Run the GWO algorithm

best\_score, best\_position = GWO(objective\_function, dimension, num\_wolves, max\_iterations, lower, upper)

print("Best Solution (Position):", best\_position)

print("Best Objective Value:", best\_score)

Output:

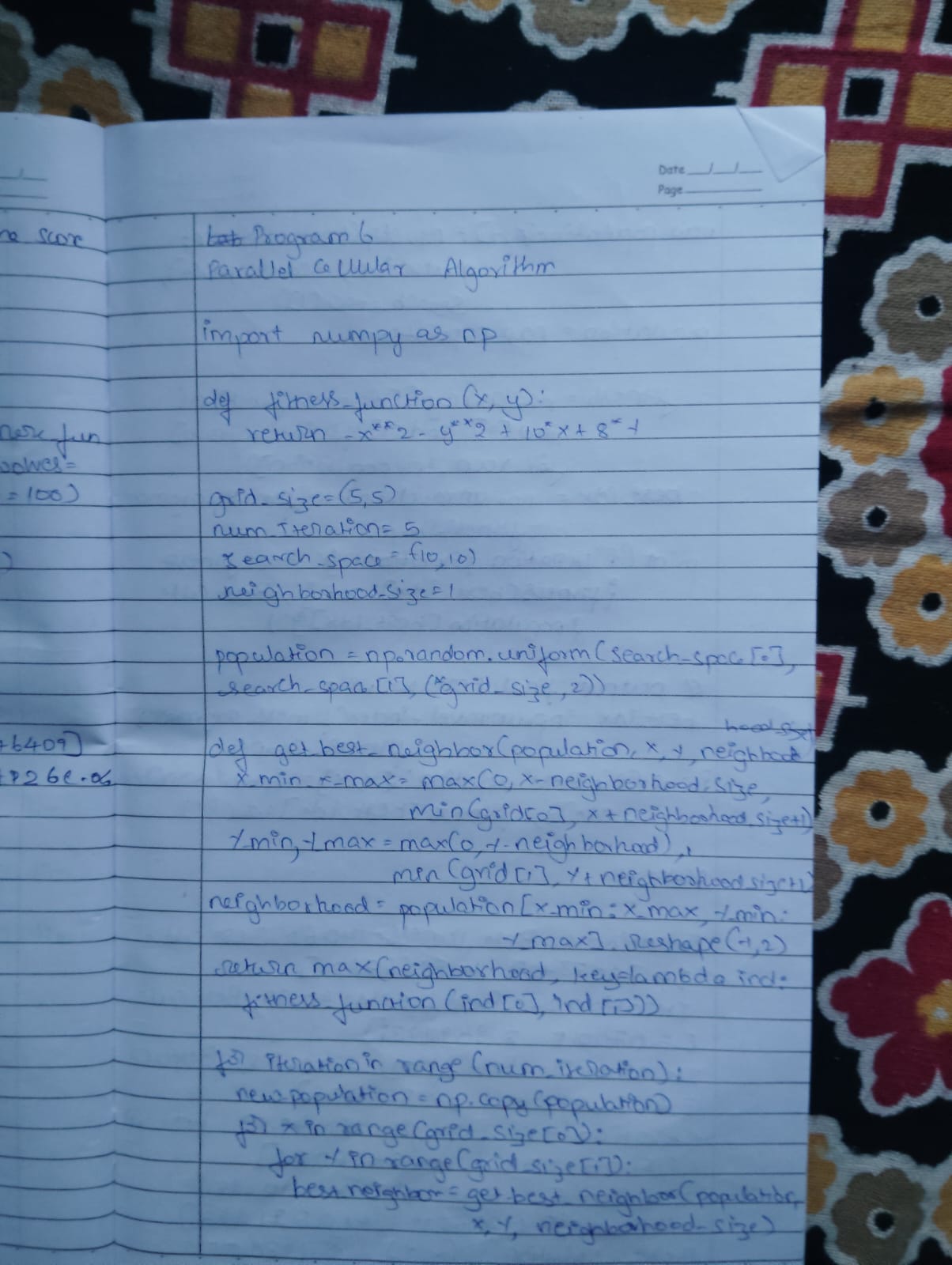
Best Solution (Position): [-5.57231784e-07 -5.89896795e-07 -6.14429041e-07 -5.84311808e-07

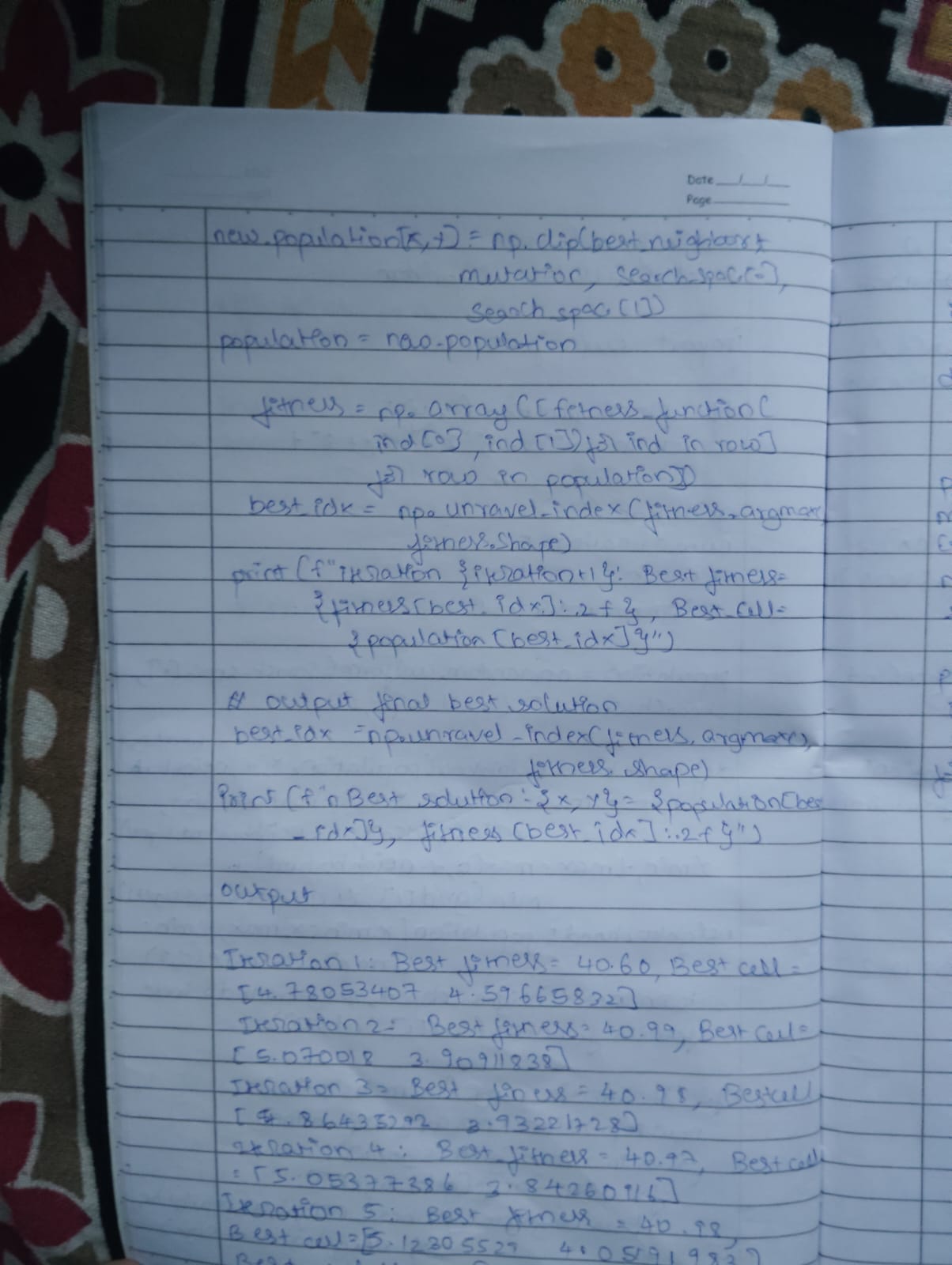
-6.29901822e-07]Best Objective Value: 1.7742051303428747e-12

**Program 6**

Implementation of Parallel Cellular Algorithms and Programs

Algorithm:



Code:

import numpy as np

# Define the fitness function

def fitness\_function(x):

    return -x\*\*2 + 5\*x + 6  # Example function to maximize

# Parameters

grid\_size = 5  # Define a 5x5 grid

num\_particles = grid\_size \*\* 2

num\_iterations = 50

search\_space = (-10, 10)

inertia\_weight = 0.5

personal\_influence = 1.5

neighbor\_influence = 1.5

# Initialize particle positions and velocities

positions = np.random.uniform(search\_space[0], search\_space[1], num\_particles)

velocities = np.random.uniform(-1, 1, num\_particles)

# Store personal best positions and their fitness values

personal\_best\_positions = np.copy(positions)

personal\_best\_fitness = np.array([fitness\_function(pos) for pos in positions])

# Define a 2D grid to simulate cellular automata

grid = positions.reshape((grid\_size, grid\_size))

# Main PCA loop

for iteration in range(num\_iterations):

    # Update particles

    for i in range(grid\_size):

        for j in range(grid\_size):

            # Get the current particle index

            particle\_idx = i \* grid\_size + j

            # Get neighbors in the grid (using periodic boundary conditions)

            neighbors = [

                ((i-1) % grid\_size, j),  # Up

                ((i+1) % grid\_size, j),  # Down

                (i, (j-1) % grid\_size),  # Left

                (i, (j+1) % grid\_size),  # Right

            ]

            # Find the best neighbor

            best\_neighbor\_fitness = -np.inf

            best\_neighbor\_position = positions[particle\_idx]

            for ni, nj in neighbors:

                neighbor\_idx = ni \* grid\_size + nj

                neighbor\_fitness = fitness\_function(positions[neighbor\_idx])

                if neighbor\_fitness > best\_neighbor\_fitness:

                    best\_neighbor\_fitness = neighbor\_fitness

                    best\_neighbor\_position = positions[neighbor\_idx]

            # Update velocity using the PSO formula

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

            velocities[particle\_idx] = (

                inertia\_weight \* velocities[particle\_idx] +

                personal\_influence \* r1 \* (personal\_best\_positions[particle\_idx] - positions[particle\_idx]) +

                neighbor\_influence \* r2 \* (best\_neighbor\_position - positions[particle\_idx])

            )

            # Update position

            positions[particle\_idx] += velocities[particle\_idx]

            # Clip position to search space

            positions[particle\_idx] = np.clip(positions[particle\_idx], search\_space[0], search\_space[1])

    # Update personal bests

    for i in range(num\_particles):

        fitness = fitness\_function(positions[i])

        if fitness > personal\_best\_fitness[i]:

            personal\_best\_fitness[i] = fitness

            personal\_best\_positions[i] = positions[i]

    # Track and print the best fitness in this iteration

    best\_fitness = personal\_best\_fitness.max()

    best\_position = personal\_best\_positions[personal\_best\_fitness.argmax()]

    print(f"Iteration {iteration+1}: Best Fitness = {best\_fitness:.2f}, Best Position = {best\_position:.2f}")

# Output the best solution found

print(f"\nBest solution: x = {best\_position:.2f}, Fitness = {best\_fitness:.2f}")

Output:

Iteration 1: Best Fitness = 12.25, Best Position = 2.50

Iteration 2: Best Fitness = 12.25, Best Position = 2.50

Iteration 3: Best Fitness = 12.25, Best Position = 2.50

Iteration 4: Best Fitness = 12.25, Best Position = 2.50

Iteration 5: Best Fitness = 12.25, Best Position = 2.50

Iteration 6: Best Fitness = 12.25, Best Position = 2.50

Iteration 7: Best Fitness = 12.25, Best Position = 2.50

Iteration 8: Best Fitness = 12.25, Best Position = 2.50

Iteration 9: Best Fitness = 12.25, Best Position = 2.50

Iteration 10: Best Fitness = 12.25, Best Position = 2.50

Iteration 11: Best Fitness = 12.25, Best Position = 2.50

Iteration 12: Best Fitness = 12.25, Best Position = 2.50

Iteration 13: Best Fitness = 12.25, Best Position = 2.50

Iteration 14: Best Fitness = 12.25, Best Position = 2.50

Iteration 15: Best Fitness = 12.25, Best Position = 2.50

Iteration 16: Best Fitness = 12.25, Best Position = 2.50

Iteration 17: Best Fitness = 12.25, Best Position = 2.50

Iteration 18: Best Fitness = 12.25, Best Position = 2.50

Iteration 19: Best Fitness = 12.25, Best Position = 2.50

Iteration 20: Best Fitness = 12.25, Best Position = 2.50

Iteration 21: Best Fitness = 12.25, Best Position = 2.50

Iteration 22: Best Fitness = 12.25, Best Position = 2.50

Iteration 23: Best Fitness = 12.25, Best Position = 2.50

Iteration 24: Best Fitness = 12.25, Best Position = 2.50

Iteration 25: Best Fitness = 12.25, Best Position = 2.50

Iteration 26: Best Fitness = 12.25, Best Position = 2.50

Iteration 27: Best Fitness = 12.25, Best Position = 2.50

Iteration 28: Best Fitness = 12.25, Best Position = 2.50

Iteration 29: Best Fitness = 12.25, Best Position = 2.50

Iteration 30: Best Fitness = 12.25, Best Position = 2.50

Iteration 31: Best Fitness = 12.25, Best Position = 2.50

Iteration 32: Best Fitness = 12.25, Best Position = 2.50

Iteration 33: Best Fitness = 12.25, Best Position = 2.50

Iteration 34: Best Fitness = 12.25, Best Position = 2.50

Iteration 35: Best Fitness = 12.25, Best Position = 2.50

Iteration 36: Best Fitness = 12.25, Best Position = 2.50

Iteration 37: Best Fitness = 12.25, Best Position = 2.50

Iteration 38: Best Fitness = 12.25, Best Position = 2.50

Iteration 39: Best Fitness = 12.25, Best Position = 2.50

Iteration 40: Best Fitness = 12.25, Best Position = 2.50

Iteration 41: Best Fitness = 12.25, Best Position = 2.50

Iteration 42: Best Fitness = 12.25, Best Position = 2.50

Iteration 43: Best Fitness = 12.25, Best Position = 2.50

Iteration 44: Best Fitness = 12.25, Best Position = 2.50

Iteration 45: Best Fitness = 12.25, Best Position = 2.50

Iteration 46: Best Fitness = 12.25, Best Position = 2.50

Iteration 47: Best Fitness = 12.25, Best Position = 2.50

Iteration 48: Best Fitness = 12.25, Best Position = 2.50

Iteration 49: Best Fitness = 12.25, Best Position = 2.50

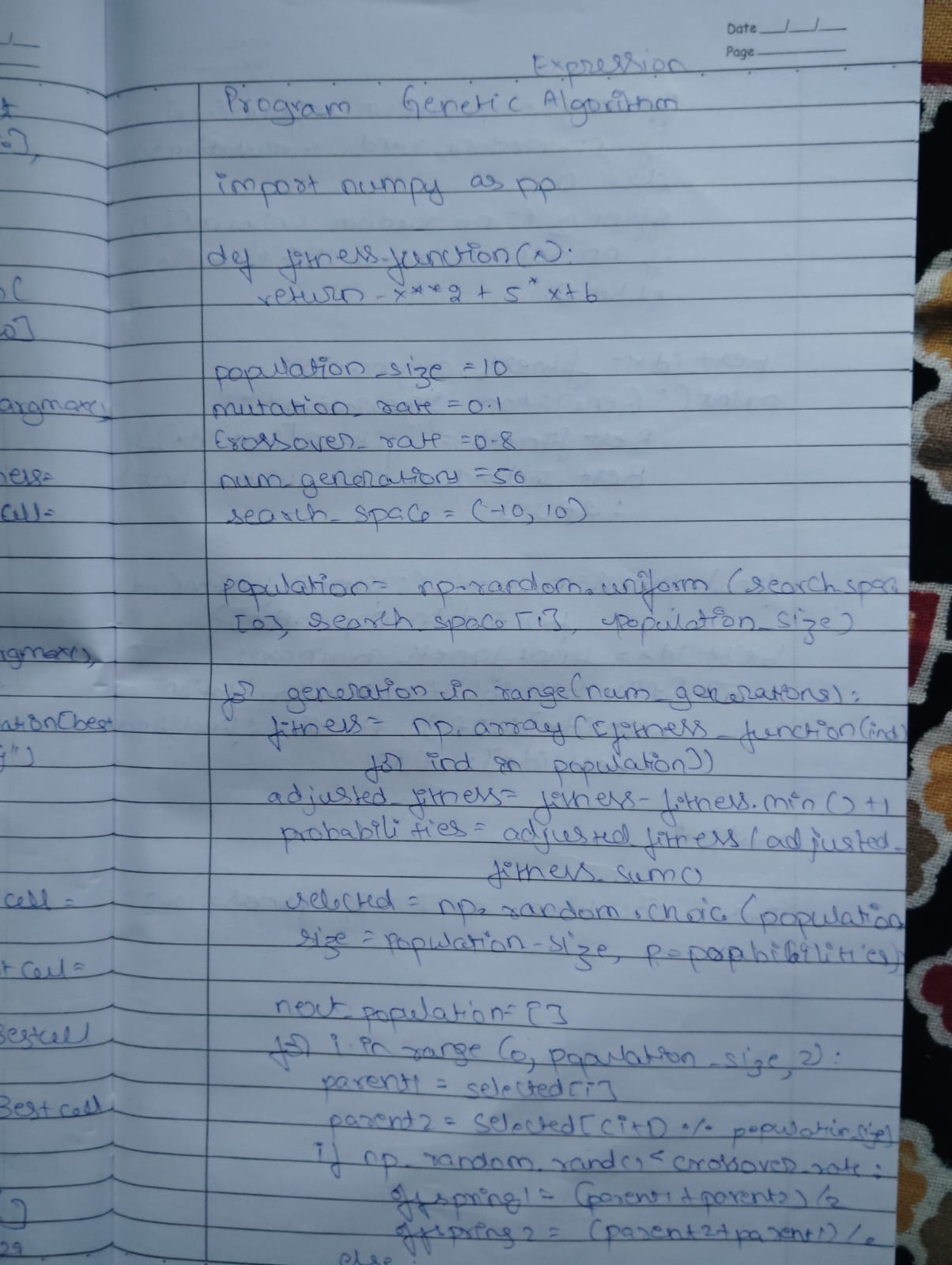
Iteration 50: Best Fitness = 12.25, Best Position = 2.50

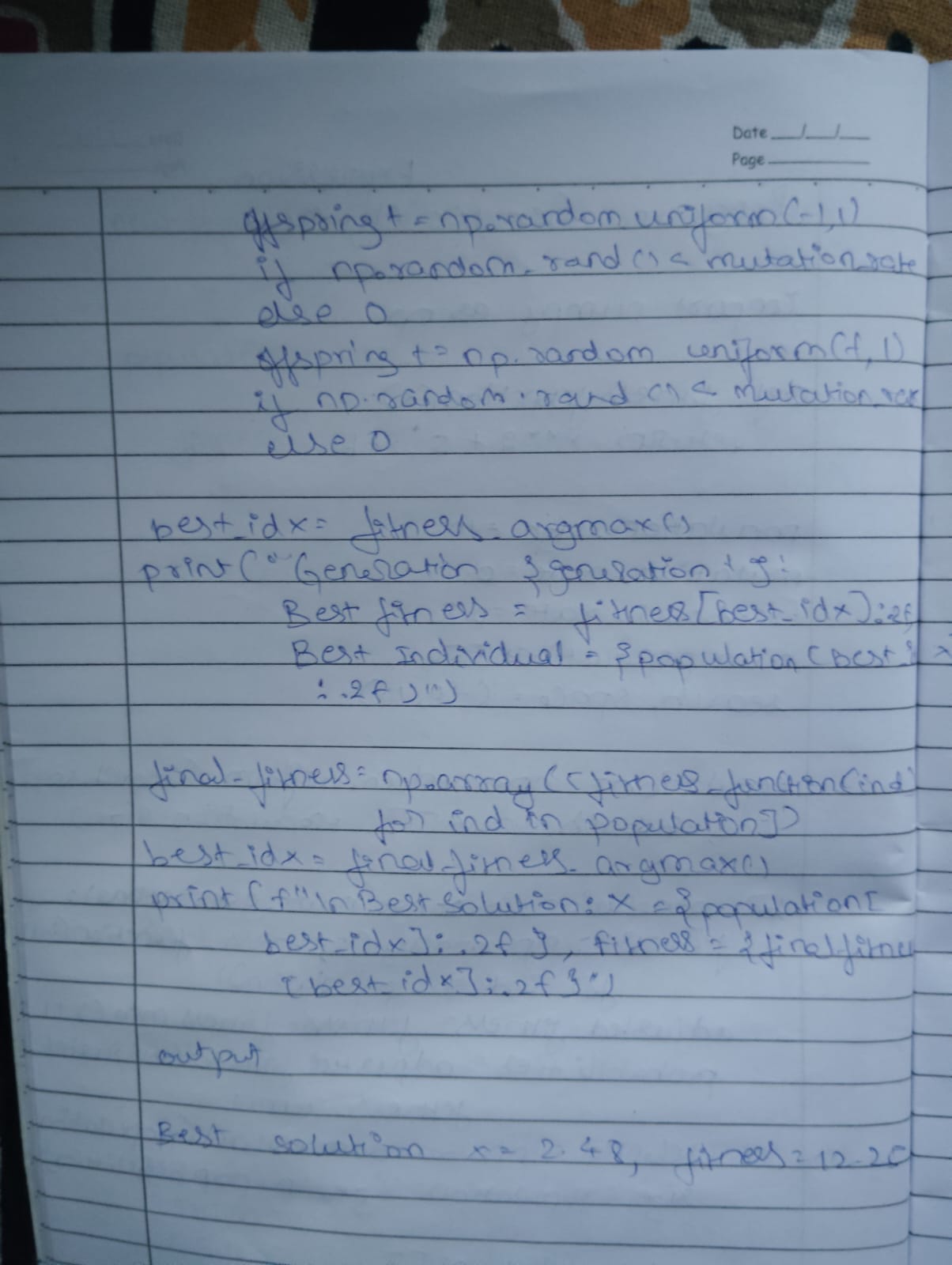
Best solution: x = 2.50, Fitness = 12.25

**Program 7**

Implementation of **Gene Expression Algorithms (GEA)**

Algorithm:





Code:

import numpy as np

# Define the function to optimize

def fitness\_function(x, y):

    return -x\*\*2 - y\*\*2 + 10\*x + 8\*y  # Example function to maximize

# Parameters

population\_size = 20

num\_genes = 2  # Each sequence represents (x, y)

num\_generations = 50

mutation\_rate = 0.1

crossover\_rate = 0.8

search\_space = (-10, 10)

# Gene Expression Function (maps genes to variables)

def express\_genes(genes):

    return genes  # In this case, genes directly represent (x, y)

# Initialize population with random genetic sequences

def initialize\_population(size, num\_genes, search\_space):

    return np.random.uniform(search\_space[0], search\_space[1], (size, num\_genes))

# Fitness Evaluation

def evaluate\_fitness(population):

    return np.array([fitness\_function(ind[0], ind[1]) for ind in population])

# Selection (Roulette Wheel Selection)

def select\_parents(population, fitness):

    probabilities = fitness - fitness.min() + 1  # Make fitness non-negative

    probabilities /= probabilities.sum()

    parent\_indices = np.random.choice(len(population), size=len(population), p=probabilities)

    return population[parent\_indices]

# Crossover

def perform\_crossover(parent1, parent2, crossover\_rate):

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

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

        offspring1 = np.concatenate((parent1[:crossover\_point], parent2[crossover\_point:]))

        offspring2 = np.concatenate((parent2[:crossover\_point], parent1[crossover\_point:]))

        return offspring1, offspring2

    return parent1, parent2

# Mutation

def mutate(offspring, mutation\_rate, search\_space):

    for gene\_index in range(len(offspring)):

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

            offspring[gene\_index] += np.random.uniform(-1, 1)

            offspring[gene\_index] = np.clip(offspring[gene\_index], search\_space[0], search\_space[1])

    return offspring

# Main Gene Expression Algorithm

def gene\_expression\_algorithm():

    population = initialize\_population(population\_size, num\_genes, search\_space)

    for generation in range(num\_generations):

        # Evaluate fitness

        fitness = evaluate\_fitness(population)

        # Selection

        parents = select\_parents(population, fitness)

        # Generate offspring

        offspring = []

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

            parent1 = parents[i]

            parent2 = parents[(i + 1) % len(parents)]

            child1, child2 = perform\_crossover(parent1, parent2, crossover\_rate)

            offspring.append(mutate(child1, mutation\_rate, search\_space))

            offspring.append(mutate(child2, mutation\_rate, search\_space))

        # Update population

        population = np.array(offspring)

        # Track best solution

        best\_fitness = fitness.max()

        best\_individual = population[np.argmax(fitness)]

        print(f"Generation {generation+1}: Best Fitness = {best\_fitness:.2f}, Best Individual = {best\_individual}")

    # Final best solution

    fitness = evaluate\_fitness(population)

    best\_index = np.argmax(fitness)

    best\_solution = population[best\_index]

    best\_fitness = fitness[best\_index]

    print(f"\nBest Solution: (x, y) = {best\_solution}, Fitness = {best\_fitness:.2f}")

# Run the algorithm

gene\_expression\_algorithm()

Output:

Generation 1: Best Fitness = 39.32, Best Individual = [6.4443493 6.05169415]

Generation 2: Best Fitness = 38.87, Best Individual = [5.11149113 6.05169415]

Generation 3: Best Fitness = 36.78, Best Individual = [8.82145888 5.21351385]

Generation 4: Best Fitness = 40.27, Best Individual = [3.90027534 0.9369009 ]

Generation 5: Best Fitness = 40.27, Best Individual = [7.15955866 6.05169415]

Generation 6: Best Fitness = 39.51, Best Individual = [5.11149113 6.05169415]

Generation 7: Best Fitness = 39.42, Best Individual = [5.85288378 5.85293052]

Generation 8: Best Fitness = 39.42, Best Individual = [7.15955866 5.21351385]

Generation 9: Best Fitness = 39.42, Best Individual = [5.20227392 5.2437933 ]

Generation 10: Best Fitness = 39.41, Best Individual = [5.65804167 5.21351385]Generation 11: Best Fitness = 39.49, Best Individual = [4.90859442 5.21351385]Generation 12: Best Fitness = 39.52, Best Individual = [4.90859442 5.21351385]Generation 13: Best Fitness = 39.52, Best Individual = [5.85288378 4.30115549]Generation 14: Best Fitness = 40.18, Best Individual = [5.85288378 5.21351385Generation 15: Best Fitness = 39.89, Best Individual = [5.65804167 5.04958577]Generation 16: Best Fitness = 39.89, Best Individual = [5.85288378 5.21351385]Generation 17: Best Fitness = 39.89, Best Individual = [4.90859442 5.04958577]Generation 18: Best Fitness = 39.89, Best Individual = [5.85288378 5.04958577]Generation 19: Best Fitness = 39.95, Best Individual = [5.54882566 5.04958577]Generation 20: Best Fitness = 40.48, Best Individual = [5.54882566 5.04958577]Generation 21: Best Fitness = 40.54, Best Individual = [5.65804167 5.04958577]Generation 22: Best Fitness = 40.41, Best Individual = [5.54882566 5.43490292]

Generation 23: Best Fitness = 39.91, Best Individual = [5.65804167 5.21351385]Generation 24: Best Fitness = 40.19, Best Individual = [5.79364332 4.8861793 ]Generation 25: Best Fitness = 40.70, Best Individual = [5.54882566 5.43490292]Generation 26: Best Fitness = 40.80, Best Individual = [4.63864695 5.04958577]Generation 27: Best Fitness = 40.70, Best Individual = [5.44084835 4.0483479 ]Generation 28: Best Fitness = 40.80, Best Individual = [5.16742759 6.11768563]Generation 29: Best Fitness = 40.74, Best Individual = [5.93615991 4.0483479 ]Generation 30: Best Fitness = 40.82, Best Individual = [5.44084835 4.8861793 ]Generation 31: Best Fitness = 40.99, Best Individual = [5.91045538 3.88969771]Generation 32: Best Fitness = 40.98, Best Individual = [5.34471628 3.88969771]Generation 33: Best Fitness = 40.88, Best Individual = [5.44084835 4.8861793 ]Generation 34: Best Fitness = 40.96, Best Individual = [4.89894219 3.82584918]Generation 35: Best Fitness = 40.96, Best Individual = [5.83107111 3.82584918]

Generation 36: Best Fitness = 40.96, Best Individual = [4.53974585 3.8960765 ]

Generation 37: Best Fitness = 40.96, Best Individual = [5.41385951 3.26677091]

Generation 38: Best Fitness = 40.98, Best Individual = [5.92663004 3.88969771]

Generation 39: Best Fitness = 40.98, Best Individual = [4.89894219 4.45997236]

Generation 40: Best Fitness = 40.99, Best Individual = [4.89894219 4.25553679]

Generation 41: Best Fitness = 40.98, Best Individual = [5.41385951 3.67520291]

Generation 42: Best Fitness = 40.96, Best Individual = [4.35953253 3.20848384]

Generation 43: Best Fitness = 40.99, Best Individual = [5.41385951 3.82584918]

Generation 44: Best Fitness = 40.99, Best Individual = [5.41385951 3.26677091]

Generation 45: Best Fitness = 40.91, Best Individual = [5.41385951 4.18141689]

Generation 46: Best Fitness = 40.99, Best Individual = [4.53277483 4.14149616]

Generation 47: Best Fitness = 40.97, Best Individual = [5.55743837 3.32829027]

Generation 48: Best Fitness = 40.91, Best Individual = [4.58377247 3.26677091]

Generation 49: Best Fitness = 40.87, Best Individual = [4.58377247 3.26677091]

Generation 50: Best Fitness = 40.97, Best Individual = [5.00052736 3.82584918]

Best Solution: (x, y) = [5.00052736 3.82584918], Fitness = 40.97