|  |
| --- |
| **VISVESVARAYA TECHNOLOGICAL UNIVERSITY**  **“JnanaSangama”, Belgaum -590014, Karnataka.**    **LAB RECORD**  **Bio Inspired Systems (23CS5BSBIS)**  ***Submitted by***  **Pranav Hegde (1BM22CS202)**  ***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 **Pranav Hegde (1BM22CS202),** 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. Basavaraj J**  Assistant Professor  Department of CSE, BMSCE | Dr. Kavitha Sooda  Professor & HOD  Department of CSE, BMSCE |

**Index**

|  |  |  |  |
| --- | --- | --- | --- |
| **Sl.**  **No.** | **Date** | **Experiment Title** | **Page No.** |
| 1 | 16/10/2024 | **Genetic Algorithm for Optimization Problems** | 1-5 |
| 2 | 23/10/2024 | **Particle Swarm Optimization** | 6-10 |
| 3 | 30/10/2024 | **Ant Colony Optimization** | 11-15 |
| 4 | 13/11/2024 | **Cuckoo Search Optimization** | 16-19 |
| 5 | 20/11/2024 | **Grey Wolf Optimizer** | 20-23 |
| 6 | 27/11/2024 | **Parallel Cellular Algorithm** | 24-28 |
| 7 | 18/12/2024 | **Gene Expression** | 29-34 |

Github Link:

https://github.com/pranav0hegde/Bis

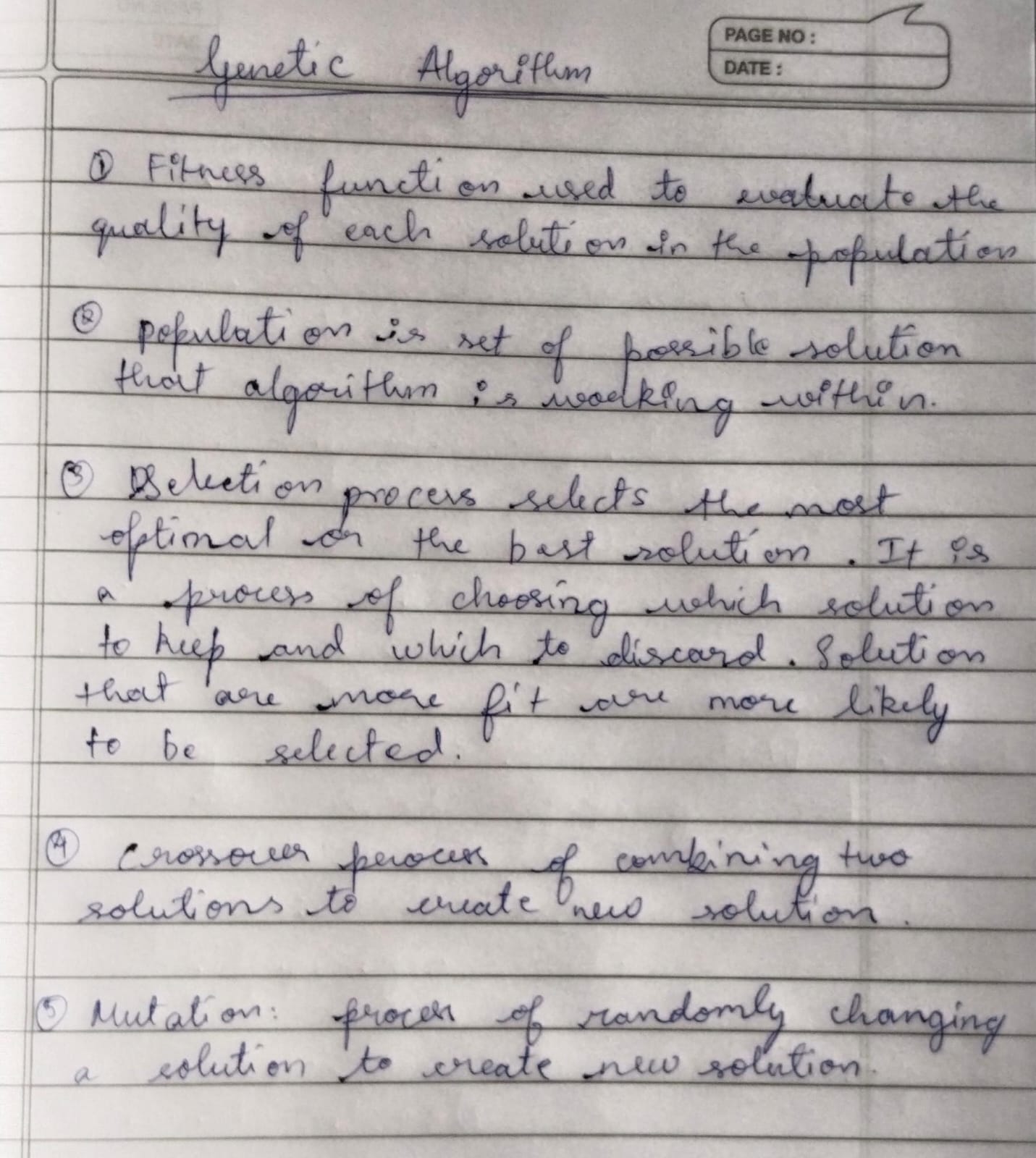
**Program 1**

**Genetic Algorithm for Optimization Problems:**

**Problem Statement:**

Genetic algorithms are a type of optimization algorithm, meaning they are used to find the optimal solution(s) to a given computational problem that maximizes or minimizes a particular function. Genetic algorithms represent one branch of the field of study called evolutionary computation, in that they imitate the biological processes of reproduction and natural selection to solve for the ‘fittest’ solutions.

Algorithm:



Code:

import random

# Define the problem: maximize f(x) = x^2

def fitness\_function(x):

    return x \*\* 2

# Initialize parameters

POPULATION\_SIZE = 100

MUTATION\_RATE = 0.01

CROSSOVER\_RATE = 0.9

NUM\_GENERATIONS = 50

X\_RANGE = (-10, 10)  # The range for the x values

# Create an individual (a solution)

def create\_individual():

    return random.uniform(X\_RANGE[0], X\_RANGE[1])

# Create the initial population

def create\_population():

    return [create\_individual() for \_ in range(POPULATION\_SIZE)]

# Evaluate the fitness of each individual

def evaluate\_population(population):

    return [fitness\_function(ind) for ind in population]

# Selection using roulette wheel method

def roulette\_wheel\_selection(population, fitness\_values):

    total\_fitness = sum(fitness\_values)

    pick = random.uniform(0, total\_fitness)

    current = 0

    for individual, fitness in zip(population, fitness\_values):

        current += fitness

        if current > pick:

            return individual

# Crossover (linear crossover)

def crossover(parent1, parent2):

    if random.random() < CROSSOVER\_RATE:

        alpha = random.random()  # Weighted combination

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

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

        return offspring1, offspring2

    else:

        return parent1, parent2

# Mutation

def mutate(individual):

    if random.random() < MUTATION\_RATE:

        return random.uniform(X\_RANGE[0], X\_RANGE[1])

    return individual

# Genetic Algorithm function

def genetic\_algorithm():

    # Step 1: Initialize the population

    population = create\_population()

    for generation in range(NUM\_GENERATIONS):

        # Step 2: Evaluate the fitness

        fitness\_values = evaluate\_population(population)

        # Track the best solution in this generation

        best\_individual = population[fitness\_values.index(max(fitness\_values))]

        best\_fitness = max(fitness\_values)

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

        # Step 3: Create a new population

        new\_population = []

        while len(new\_population) < POPULATION\_SIZE:

            # Step 4: Selection

            parent1 = roulette\_wheel\_selection(population, fitness\_values)

            parent2 = roulette\_wheel\_selection(population, fitness\_values)

            # Step 5: Crossover

            offspring1, offspring2 = crossover(parent1, parent2)

            # Step 6: Mutation

            offspring1 = mutate(offspring1)

            offspring2 = mutate(offspring2)

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

        # Step 7: Replacement with the new population

        population = new\_population[:POPULATION\_SIZE]  # Ensure population size is maintained

    # After all generations, return the best solution found

    fitness\_values = evaluate\_population(population)

    best\_individual = population[fitness\_values.index(max(fitness\_values))]

    best\_fitness = max(fitness\_values)

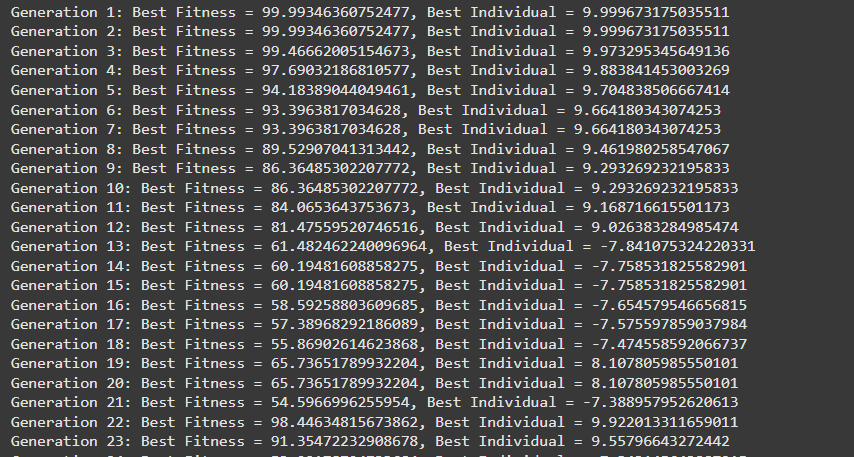
    print(f"\nBest Solution: x = {best\_individual}, f(x) = {best\_fitness}")

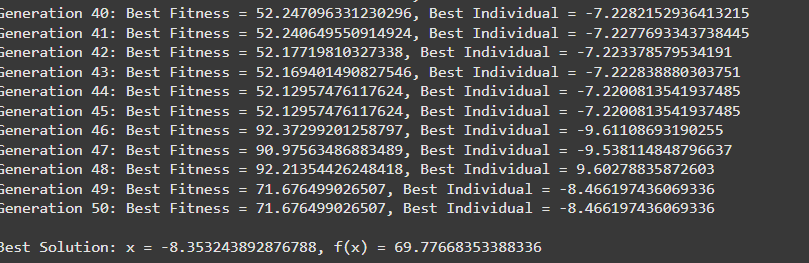
# Run the genetic algorithm

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

    genetic\_algorithm()

Output:





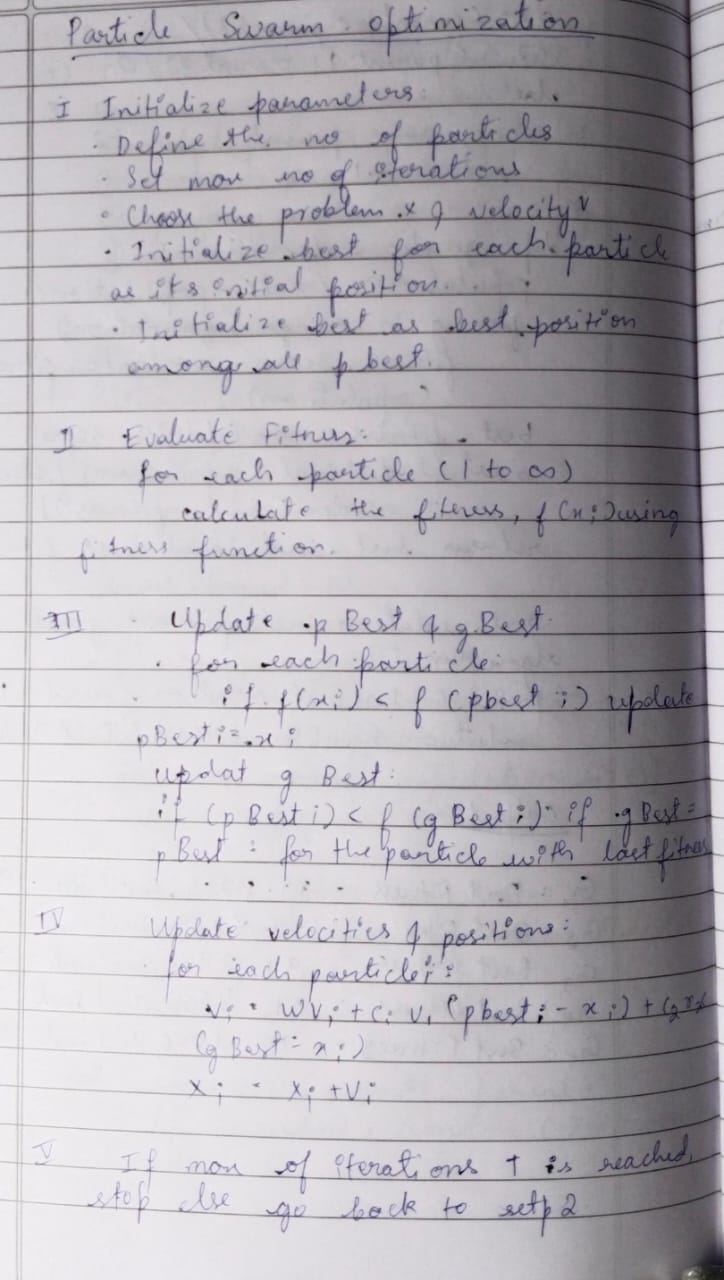
**Program 2**

**Particle Swarm Optimization**

**Problem Statement:**

The objective is to optimize a given function using Particle Swarm Optimization (PSO). PSO is a population-based metaheuristic algorithm inspired by the social behavior of particles in a swarm. The algorithm aims to find the optimal solution by iteratively adjusting particle positions and velocities within a defined search space to minimize or maximize the objective function. The search considers constraints, if any, to ensure feasibility

**Algorithm:**



**Code:**

import random

# Objective function: f(x) = x^2

def fitness\_function(x):

    return x\*\*2

# Particle class to represent each particle in the swarm

class Particle:

    def \_\_init\_\_(self, min\_x, max\_x):

        self.position = random.uniform(min\_x, max\_x)  # Current position

        self.velocity = random.uniform(-1, 1)          # Current velocity

        self.best\_position = self.position               # Best position found by the particle

        self.best\_fitness = fitness\_function(self.position)  # Best fitness value

    def update\_velocity(self, global\_best\_position, inertia\_weight, cognitive\_coefficient, social\_coefficient):

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

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

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

        self.velocity = (inertia\_weight \* self.velocity) + cognitive\_velocity + social\_velocity

    def update\_position(self, min\_x, max\_x):

        self.position += self.velocity

        # Ensure the position is within bounds

        self.position = max(min\_x, min(self.position, max\_x))

        # Update the best position and fitness if needed

        fitness = fitness\_function(self.position)

        if fitness < self.best\_fitness:  # We want to minimize

            self.best\_fitness = fitness

            self.best\_position = self.position

# PSO algorithm

def particle\_swarm\_optimization(pop\_size, min\_x, max\_x, generations, inertia\_weight, cognitive\_coefficient, social\_coefficient):

    # Initialize particles

    swarm = [Particle(min\_x, max\_x) for \_ in range(pop\_size)]

    # Global best position initialized to None

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

    global\_best\_fitness = swarm[0].best\_fitness

    for generation in range(generations):

        for particle in swarm:

            # Update global best position

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

                global\_best\_fitness = particle.best\_fitness

                global\_best\_position = particle.best\_position

            # Update particle velocity and position

            particle.update\_velocity(global\_best\_position, inertia\_weight, cognitive\_coefficient, social\_coefficient)

            particle.update\_position(min\_x, max\_x)

        # Print the best fitness in the current generation

        print(f"Generation {generation + 1}: Best solution = {global\_best\_position}, Fitness = {global\_best\_fitness}")

    return global\_best\_position

# Parameters

population\_size = 30

min\_value = -10

max\_value = 10

num\_generations = 50

inertia\_weight = 0.5

cognitive\_coefficient = 1.5

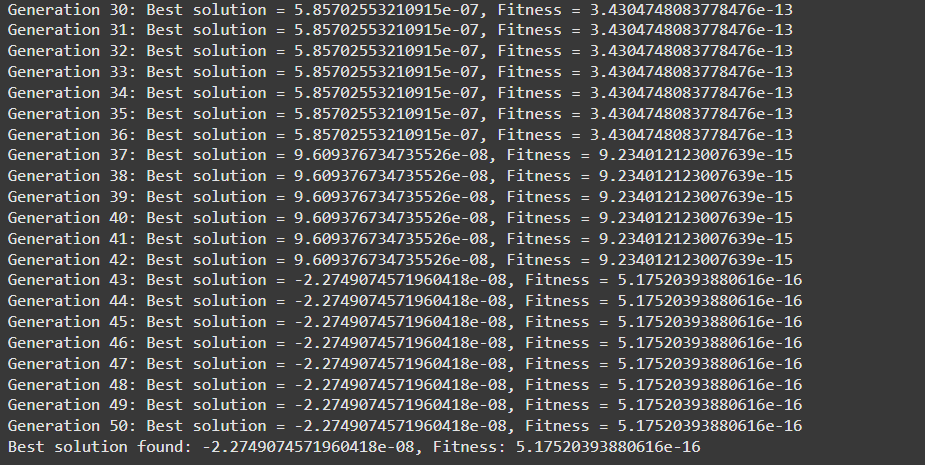
social\_coefficient = 1.5

# Run Particle Swarm Optimization

best\_solution = particle\_swarm\_optimization(population\_size, min\_value, max\_value, num\_generations, inertia\_weight, cognitive\_coefficient, social\_coefficient)

print(f"Best solution found: {best\_solution}, Fitness: {fitness\_function(best\_solution)}")

Output:



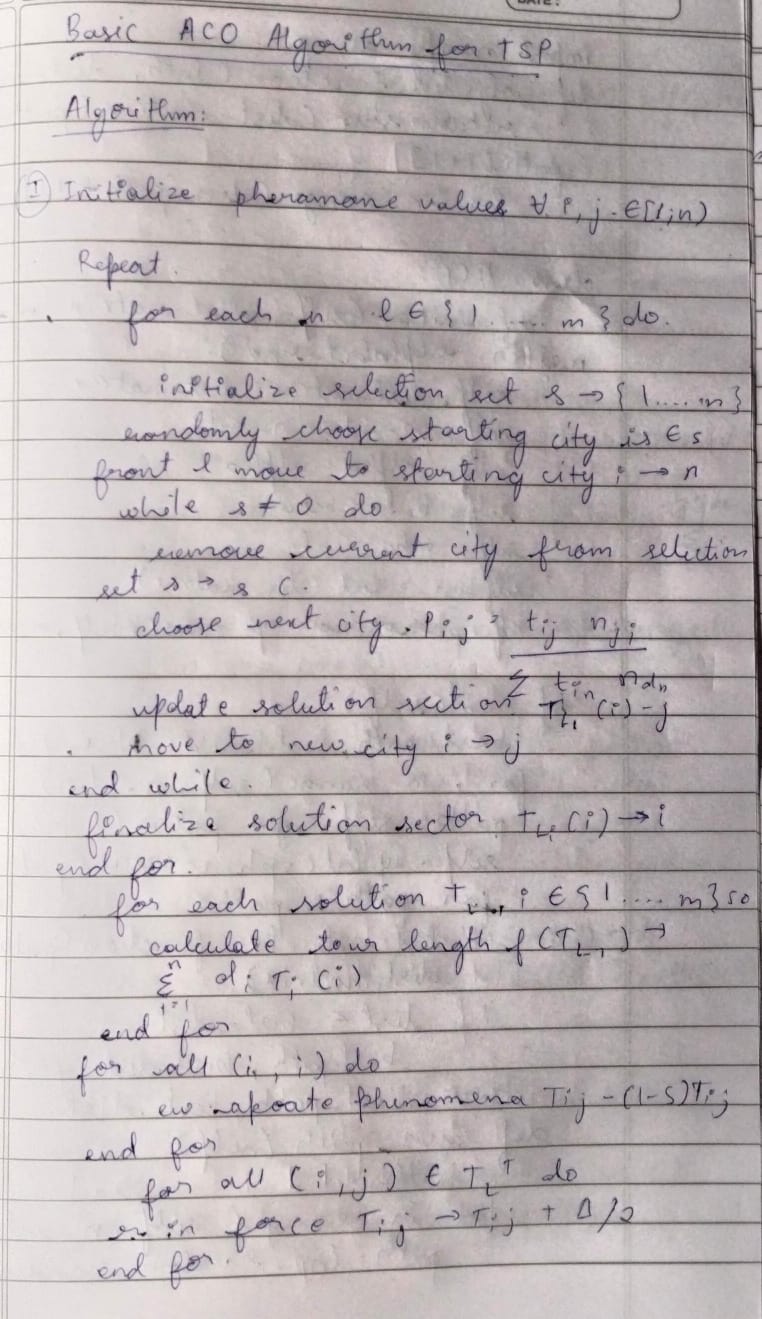
**Program 3**

**Ant Colony Optimization**

**Problem Statement:**

Ant Colony Optimization (ACO) algorithm solves combinatorial optimization problem, such as finding the shortest path, optimizing resource allocation, or scheduling tasks. The algorithm should simulate the behavior of ants by using pheromone trails and heuristic information to iteratively discover and refine optimal or near-optimal solutions while adhering to the constraints of the problem.

**Algorithm:**



1. **Initialization**:

* Define the number of nodes, ants, and initialize parameters: pheromone matrix, distance matrix, and heuristic information.
* Set initial pheromone levels to 1 on all edges.

1. **Ant Construction Phase**:

For each ant:

* 1. Start from a random node.
  2. Repeat until all nodes are visited:
     1. Compute the probability of moving to each unvisited node based on pheromone level and heuristic value (distance).
     2. Select the next node probabilistically and move there.
     3. Update the ant’s path and distance.
  3. Complete the tour by returning to the starting node.

1. **Pheromone Update Phase**:

* Evaporate pheromone on all edges by multiplying them with 1 - evaporation rate.
* For each ant, deposit pheromone along its path proportionally to the inverse of its tour distance.

1. **Iterative Optimization**:

* Repeat the **Ant Construction Phase** and **Pheromone Update Phase** for a fixed number of iterations.
* Track the best solution (path and distance) found across all iterations.

1. **Return the Best Solution**:

* Output the shortest path and its corresponding distance.

**Code:**

import numpy as np

import random

class Ant:

    def \_\_init\_\_(self, num\_nodes):

        self.path = []

        self.distance = 0

        self.num\_nodes = num\_nodes

    def visit\_node(self, node, distance\_matrix):

        if len(self.path) > 0:

            self.distance += distance\_matrix[self.path[-1]][node]

        self.path.append(node)

    def tour\_complete(self, distance\_matrix):

        return\_to\_start = distance\_matrix[self.path[-1]][self.path[0]]

        self.distance += return\_to\_start

        self.path.append(self.path[0])  # return to start node

class AntColonyOptimizer:

    def \_\_init\_\_(self, num\_nodes, distance\_matrix, num\_ants, alpha=1, beta=2, evaporation=0.5, q=10):

        self.num\_nodes = num\_nodes

        self.distance\_matrix = distance\_matrix

        self.num\_ants = num\_ants

        self.alpha = alpha

        self.beta = beta

        self.evaporation = evaporation

        self.q = q

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

    def \_probability(self, i, j, visited):

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

        heuristic = (1 / self.distance\_matrix[i][j]) \*\* self.beta

        return pheromone \* heuristic if j not in visited else 0

    def \_select\_next\_node(self, ant):

        unvisited = [node for node in range(self.num\_nodes) if node not in ant.path]

        probabilities = [self.\_probability(ant.path[-1], node, ant.path) for node in unvisited]

        total = sum(probabilities)

        if total == 0: return random.choice(unvisited)

        probabilities = [p / total for p in probabilities]

        return np.random.choice(unvisited, p=probabilities)

    def \_update\_pheromones(self, ants):

        self.pheromone \*= (1 - self.evaporation)

        for ant in ants:

            contribution = self.q / ant.distance

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

                u, v = ant.path[i], ant.path[i + 1]

                self.pheromone[u][v] += contribution

                self.pheromone[v][u] += contribution

    def run(self, iterations=100):

        best\_distance = float('inf')

        best\_path = []

        for \_ in range(iterations):

            ants = [Ant(self.num\_nodes) for \_ in range(self.num\_ants)]

            for ant in ants:

                ant.visit\_node(random.randint(0, self.num\_nodes - 1), self.distance\_matrix)

                while len(ant.path) < self.num\_nodes:

                    next\_node = self.\_select\_next\_node(ant)

                    ant.visit\_node(next\_node, self.distance\_matrix)

                ant.tour\_complete(self.distance\_matrix)

                if ant.distance < best\_distance:

                    best\_distance = ant.distance

                    best\_path = ant.path

            self.\_update\_pheromones(ants)

        return best\_path, best\_distance

# Example Usage

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

    num\_nodes = 5

    distance\_matrix = np.array([

        [0, 2, 2, 3, 4],

        [2, 0, 4, 5, 3],

        [2, 4, 0, 2, 3],

        [3, 5, 2, 0, 5],

        [4, 3, 3, 5, 0]

    ])

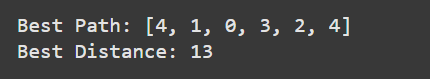
    optimizer = AntColonyOptimizer(num\_nodes, distance\_matrix, num\_ants=10)

    best\_path, best\_distance = optimizer.run(iterations=100)

    print(f"Best Path: {best\_path}")

    print(f"Best Distance: {best\_distance}")

Output:



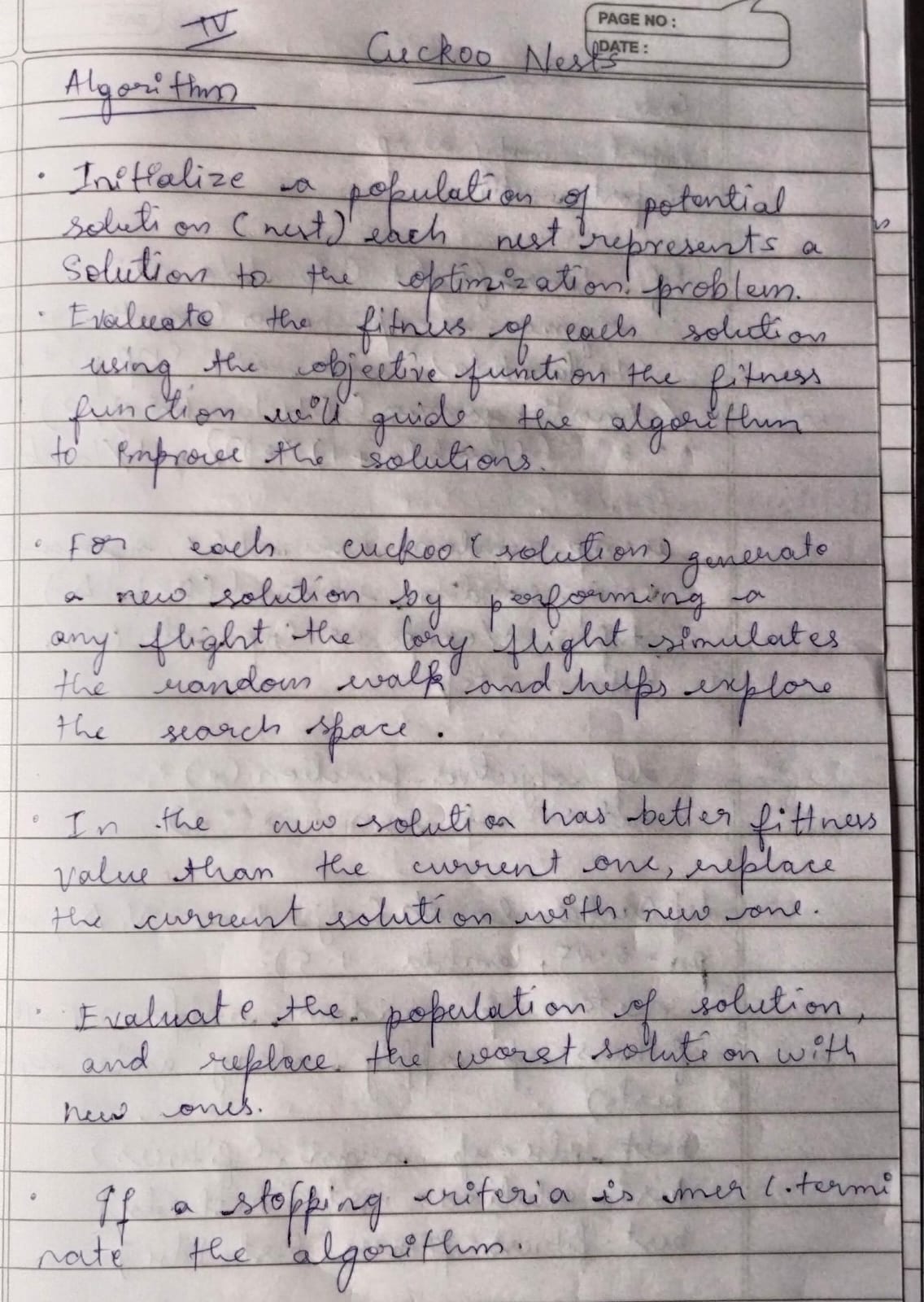
**Program 4**

**Cuckoo Search Optimization**

**Problem Statement:**

Design a Cuckoo Search algorithm to solve an optimization problem by mimicking the behavior of cuckoo birds laying eggs in host nests. The algorithm should use Lévy flights to explore the solution space, evaluate the fitness of solutions, and iteratively replace weaker solutions with stronger ones, aiming to find the optimal result for the given objective.

**Algorithm:**



**Code:**

import numpy as np

# Objective function to minimize

def objective\_function(x):

return x[0]\*2 + x[1]\*2 # Example: simple quadratic function

# Lévy flight step generator

def levy\_flight(Lambda):

sigma = (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, 2)

v = np.random.normal(0, 1, 2)

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

return step

# CS parameters

num\_nests = 25

discovery\_rate = 0.25

iterations = 100

Lambda = 1.5 # Parameter for Lévy flights

# Initialize nests

nests = np.random.uniform(-10, 10, (num\_nests, 2))

best\_nest = nests[0]

best\_fitness = objective\_function(best\_nest)

# Main loop

for \_ in range(iterations):

# Generate new solutions using Lévy flight

for i in range(num\_nests):

step\_size = levy\_flight(Lambda)

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

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

# If new solution is better, replace the current solution

if new\_fitness < objective\_function(nests[i]):

nests[i] = new\_solution

# Update best solution

if new\_fitness < best\_fitness:

best\_fitness = new\_fitness

best\_nest = new\_solution

# Abandon a fraction of worst nests

num\_abandoned = int(discovery\_rate \* num\_nests)

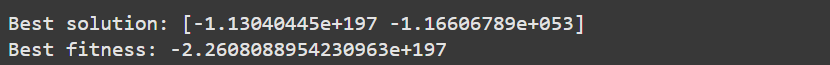
worst\_indices = np.argsort([objective\_function(nest) for nest in nests])[-num\_abandoned:]

nests[worst\_indices] = np.random.uniform(-10, 10, (num\_abandoned, 2))

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

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

**Output:**

****

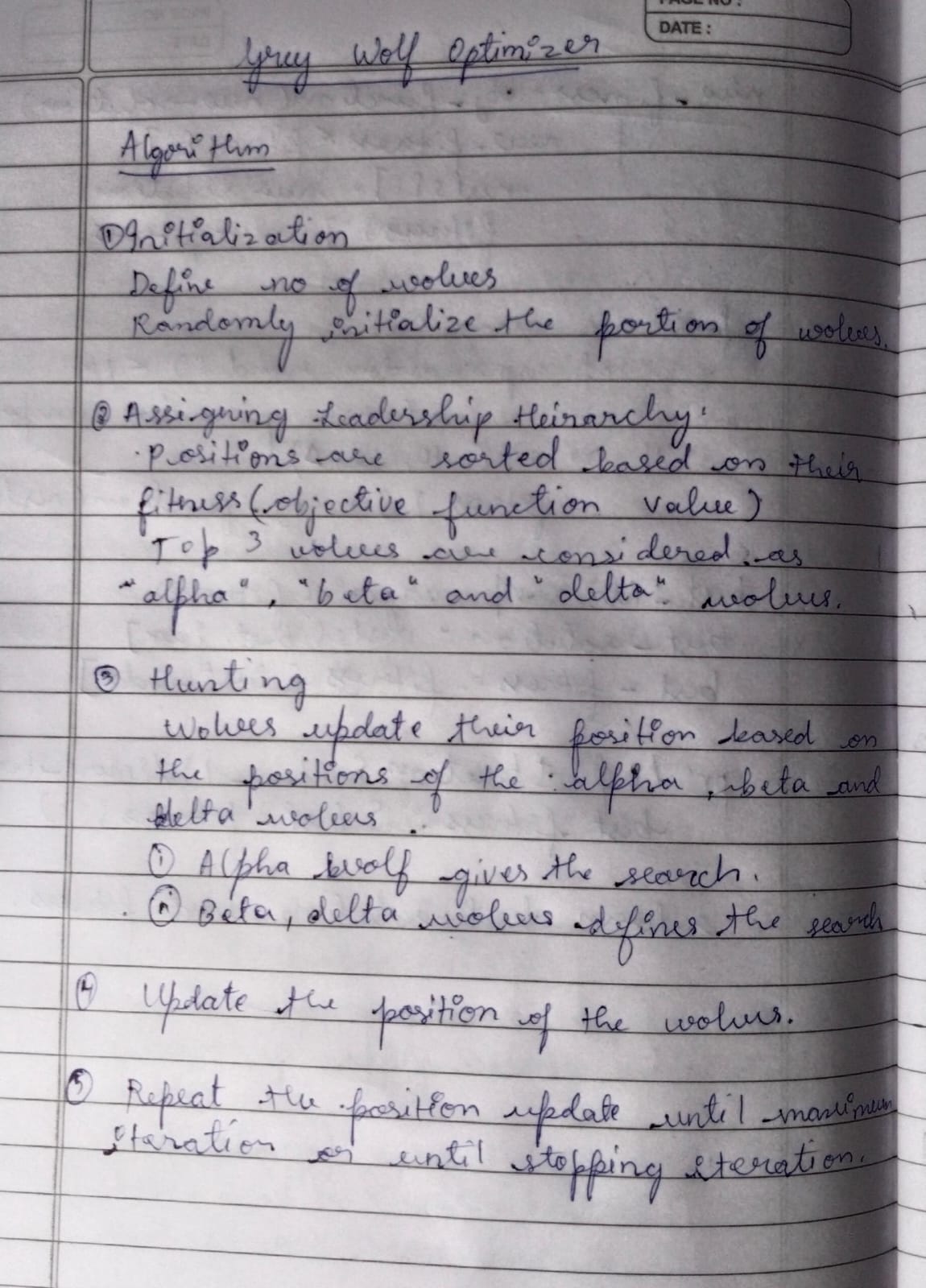
**Program 5**

**Grey Wolf Optimizer:**

**Problem Statement:**

Develop a Grey Wolf Optimizer (GWO) algorithm to solve an optimization problem by mimicking the leadership hierarchy and hunting behavior of grey wolves. The algorithm should simulate the collaborative approach of alpha, beta, delta, and omega wolves to explore and exploit the search space, aiming to find the optimal solution for the given objective.

**Algorithm:**



**Code:**

# Grey Wolf

import numpy as np

def obj\_fn(x):

    """Objective function to minimize."""

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

def gwo(obj\_fn, dim, wolves, iters, lb, ub):

    """Grey Wolf Optimm,kkl,k,,lkpppppppppizer (GWO) implementation."""

    # Initialize wolf positions

    pos = np.random.uniform(low=lb, high=ub, size=(wolves, dim))

    a\_pos, b\_pos, d\_pos = np.zeros(dim), np.zeros(dim), np.zeros(dim)

    a\_score, b\_score, d\_score = float("inf"), float("inf"), float("inf")

    for t in range(iters):

        for i in range(wolves):

            fit = obj\_fn(pos[i])

            # Update Alpha, Beta, Delta

            if fit < a\_score:

                d\_score, d\_pos = b\_score, b\_pos.copy()

                b\_score, b\_pos = a\_score, a\_pos.copy()

                a\_score, a\_pos = fit, pos[i].copy()

            elif fit < b\_score:

                d\_score, d\_pos = b\_score, b\_pos.copy()

                b\_score, b\_pos = fit, pos[i].copy()

            elif fit < d\_score:

                d\_score, d\_pos = fit, pos[i].copy()

        # Update wolf positions

        a = 2 - t \* (2 / iters)  # Linearly decreasing factor

        for i in range(wolves):

            for j in range(dim):

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

                A1, C1 = 2 \* a \* r1 - a, 2 \* r2

                D\_a = abs(C1 \* a\_pos[j] - pos[i, j])

                X1 = a\_pos[j] - A1 \* D\_a

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

                A2, C2 = 2 \* a \* r1 - a, 2 \* r2

                D\_b = abs(C2 \* b\_pos[j] - pos[i, j])

                X2 = b\_pos[j] - A2 \* D\_b

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

                A3, C3 = 2 \* a \* r1 - a, 2 \* r2

                D\_d = abs(C3 \* d\_pos[j] - pos[i, j])

                X3 = d\_pos[j] - A3 \* D\_d

                # Update position

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

            # Keep wolves within bounds

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

        # Print progress

        print(f"Iter {t+1}/{iters}, Best Score: {a\_score}, Best Pos: {a\_pos}")

    return a\_score, a\_pos

# Parameters

dim = 5       # Problem dimension

wolves = 20   # Number of wolves

iters = 50   # Number of iterations

lb = -10      # Lower bound

ub = 10       # Upper bound

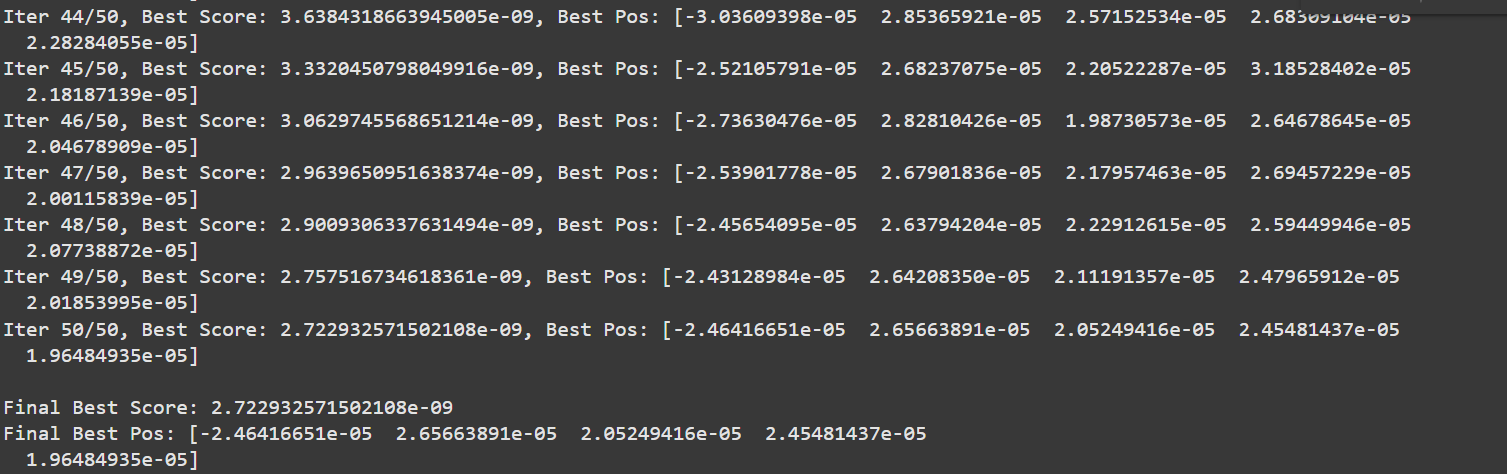
# Run GWO

best\_score, best\_pos = gwo(obj\_fn, dim, wolves, iters, lb, ub)

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

print("Final Best Pos:", best\_pos)

**Output:**

****

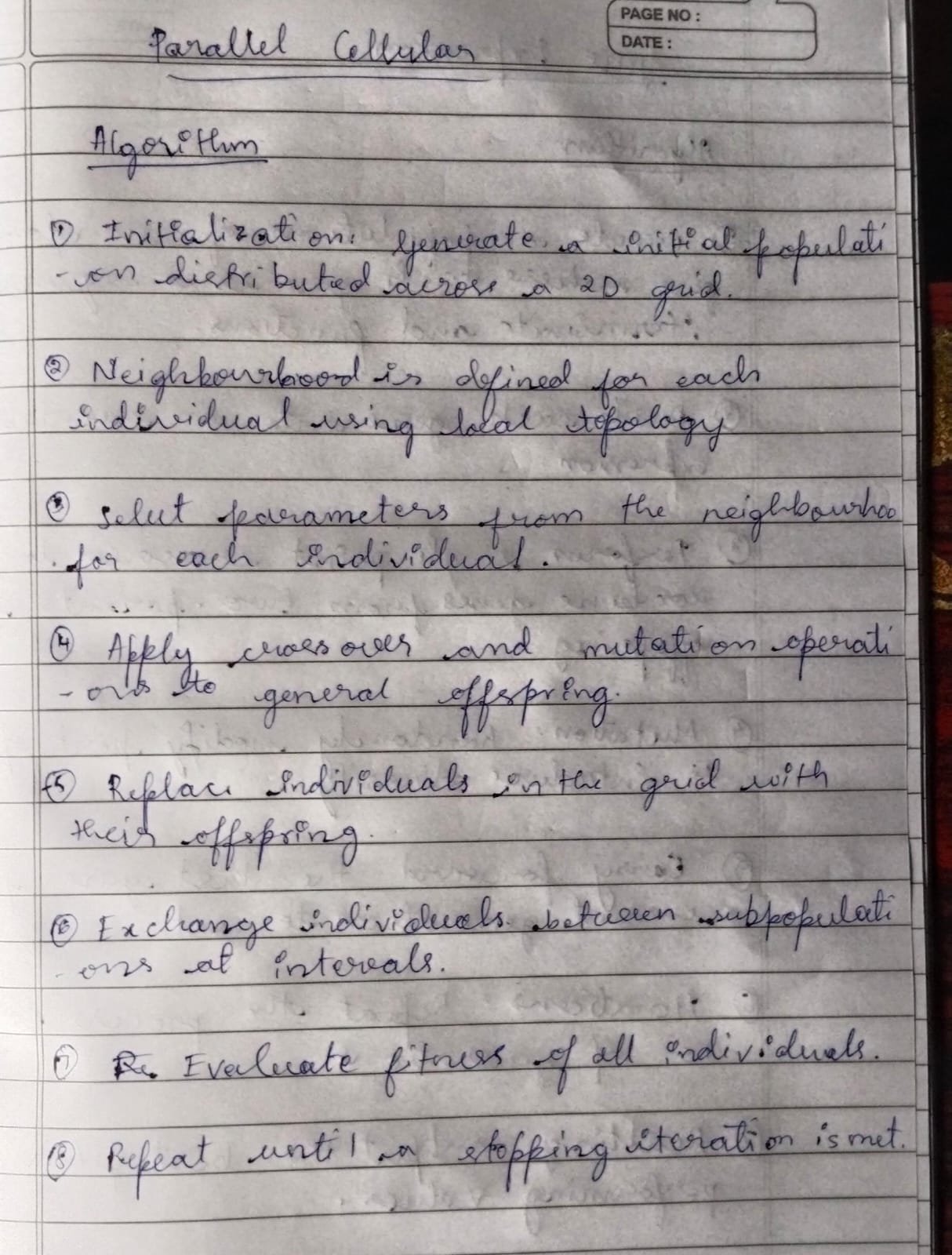
**Program 6**

**Parllel Cellular Algorithm**

**Problem Statement:**

Design a Parallel Cellular Algorithm to solve [specific optimization problem]. The algorithm should utilize a grid-based approach where each cell represents an independent entity capable of local computation. These cells communicate with their neighbors to iteratively improve the solution, leveraging parallel processing to accelerate convergence. The goal is to find the optimal or near-optimal solution for the given problem, ensuring efficiency and scalability across multiple processors.

**Algorithm:**



**Code:**

import numpy as np

import random

# Objective function (Sphere function)

def objective\_function(x):

    return np.sum(x \*\* 2)

# Initialize the grid (population)

def initialize\_grid(grid\_size, dim, bounds):

    return np.random.uniform(bounds[0], bounds[1], (grid\_size, grid\_size, dim))

# Evaluate fitness of the grid

def evaluate\_grid(grid, objective\_function):

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

    for i in range(grid.shape[0]):

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

            fitness[i, j] = objective\_function(grid[i, j])

    return fitness

# Selection using the best individual in the neighborhood

def select\_best\_neighbor(grid, fitness, x, y):

    neighbors = [

        ((x - 1) % grid.shape[0], y),   # Up

        ((x + 1) % grid.shape[0], y),   # Down

        (x, (y - 1) % grid.shape[1]),   # Left

        (x, (y + 1) % grid.shape[1]),   # Right

    ]

    best\_pos = min(neighbors, key=lambda pos: fitness[pos[0], pos[1]])

    return grid[best\_pos[0], best\_pos[1]]

# Crossover operation

def crossover(parent1, parent2):

    alpha = np.random.rand()

    return alpha \* parent1 + (1 - alpha) \* parent2

# Mutation operation

def mutate(individual, bounds, mutation\_rate=0.1):

    for i in range(len(individual)):

        if random.random() < mutation\_rate:

            individual[i] += np.random.uniform(-1, 1)

            individual[i] = np.clip(individual[i], bounds[0], bounds[1])

    return individual

# Main Parallel Cellular Genetic Algorithm

def parallel\_cellular\_ga(objective\_function, grid\_size=5, dim=2, bounds=(-5, 5), max\_iter=100, mutation\_rate=0.1):

    # Initialize the grid and fitness

    grid = initialize\_grid(grid\_size, dim, bounds)

    fitness = evaluate\_grid(grid, objective\_function)

    for iteration in range(max\_iter):

        new\_grid = np.copy(grid)

        for i in range(grid\_size):

            for j in range(grid\_size):

                # Select parents from the neighborhood

                parent1 = grid[i, j]

                parent2 = select\_best\_neighbor(grid, fitness, i, j)

                # Apply crossover and mutation

                offspring = crossover(parent1, parent2)

                offspring = mutate(offspring, bounds, mutation\_rate)

                # Replace if offspring is better

                offspring\_fitness = objective\_function(offspring)

                if offspring\_fitness < fitness[i, j]:

                    new\_grid[i, j] = offspring

                    fitness[i, j] = offspring\_fitness

        grid = new\_grid

        # Output the best solution in the grid

        best\_position = np.unravel\_index(np.argmin(fitness), fitness.shape)

        best\_fitness = fitness[best\_position]

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

    # Return the best solution

    best\_position = np.unravel\_index(np.argmin(fitness), fitness.shape)

    return grid[best\_position[0], best\_position[1]], fitness[best\_position]

# Parameters

grid\_size = 5         # Size of the grid

dim = 2               # Dimensionality of the problem

bounds = (-5, 5)      # Search space boundaries

max\_iter = 50         # Number of iterations

mutation\_rate = 0.1   # Mutation rate

# Run PCGA

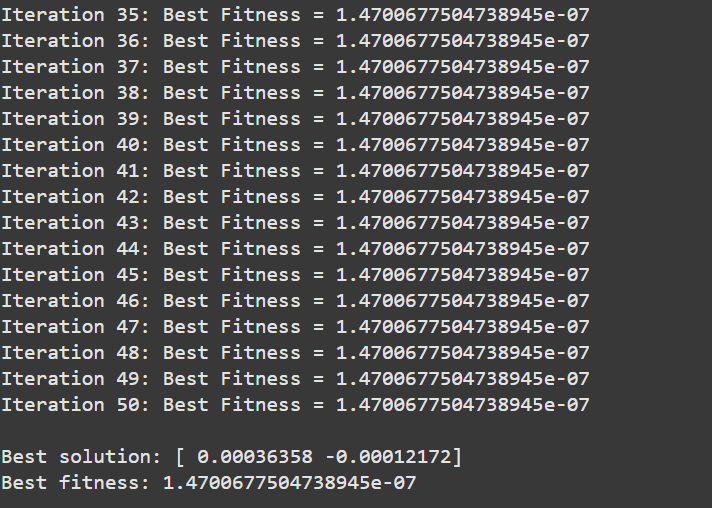
best\_solution, best\_fitness = parallel\_cellular\_ga(objective\_function, grid\_size, dim, bounds, max\_iter, mutation\_rate)

# Output the best solution

print(f"\nBest solution: {best\_solution}")

print(f"Best fitness: {best\_fitness}")

**Output:**

****

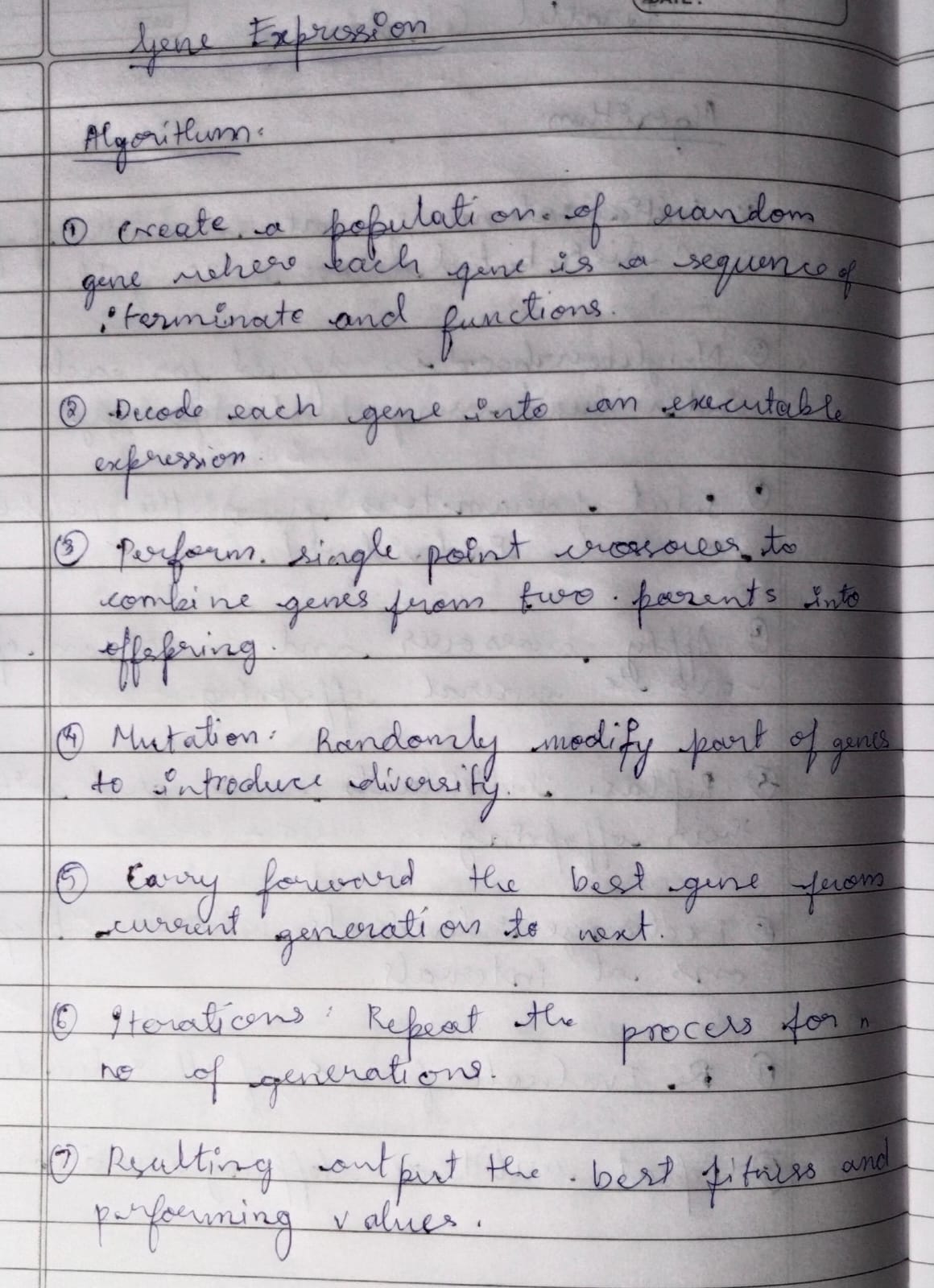
**Program 7**

**Optimization via Gene Expression**

**Problem Statement:**

Design an optimization system using the Gene Expression Algorithm to evolve mathematical expressions that minimize a given cost function. The problem requires creating a population of encoded mathematical expressions (genes) that are iteratively refined through genetic operations like selection, crossover, and mutation. The goal is to decode these expressions and evaluate their fitness based on how closely they approximate the desired output of the cost function, ensuring the algorithm converges to the most optimal solution over successive generations.

**Algorithm:**



**Code:**

import random

import math

# --- PARAMETERS ---

POPULATION\_SIZE = 50

GENE\_LENGTH = 30

GENERATIONS = 100

MUTATION\_RATE = 0.05

CROSSOVER\_RATE = 0.7

# Terminals (constants, variable 'x') and Functions

TERMINALS = ['x', '1', '2', '3', '4', '5']

FUNCTIONS = ['+', '-', '\*', '/', 'sin', 'cos']

# Target Cost Function (to minimize)

def cost\_function(x):

    """ Example cost function to minimize. Replace with your target function. """

    return x\*\*2 - 10 \* math.sin(2 \* x)

# --- GENE EXPRESSION CLASS ---

class GeneExpression:

    def \_\_init\_\_(self):

        self.gene = self.\_random\_gene()

        self.cached\_fitness = None  # To store fitness value

    def \_random\_gene(self):

        """ Initialize a random gene sequence. """

        return [random.choice(TERMINALS + FUNCTIONS) for \_ in range(GENE\_LENGTH)]

    def decode\_gene(self, x):

        """ Decode the gene into a mathematical expression and evaluate it. """

        stack = []

        for token in self.gene:

            if token in TERMINALS:

                stack.append(float(x) if token == 'x' else float(token))

            elif token in FUNCTIONS:

                if len(stack) >= 1 and token in ['sin', 'cos']:

                    arg = stack.pop()

                    stack.append(math.sin(arg) if token == 'sin' else math.cos(arg))

                elif len(stack) >= 2:

                    b, a = stack.pop(), stack.pop()

                    if token == '+': stack.append(a + b)

                    elif token == '-': stack.append(a - b)

                    elif token == '\*': stack.append(a \* b)

                    elif token == '/' and b != 0: stack.append(a / b)

                else:

                    return float('inf')  # Malformed gene

        return stack[0] if len(stack) == 1 else float('inf')

    def fitness(self, x):

        """ Evaluate fitness: minimize cost\_function(output). """

        if self.cached\_fitness is None:

            try:

                result = self.decode\_gene(x)

                self.cached\_fitness = abs(cost\_function(result))

            except:

                self.cached\_fitness = float('inf')

        return self.cached\_fitness

# --- GENETIC OPERATIONS ---

def selection(population, fitnesses):

    """ Tournament selection: Select the best from random candidates. """

    tournament\_size = 3

    candidates = random.sample(list(zip(population, fitnesses)), tournament\_size)

    return min(candidates, key=lambda c: c[1])[0]

def crossover(parent1, parent2):

    """ Perform single-point crossover between two parents. """

    if random.random() < CROSSOVER\_RATE:

        point = random.randint(1, GENE\_LENGTH - 1)

        child1 = GeneExpression()

        child2 = GeneExpression()

        child1.gene = parent1.gene[:point] + parent2.gene[point:]

        child2.gene = parent2.gene[:point] + parent1.gene[point:]

        return child1, child2

    return parent1, parent2

def mutate(individual):

    """ Apply mutation by altering random parts of the gene. """

    for i in range(GENE\_LENGTH):

        if random.random() < MUTATION\_RATE:

            individual.gene[i] = random.choice(TERMINALS + FUNCTIONS)

# --- MAIN EVOLUTION FUNCTION ---

def geneExpression():

    # Initialization

    population = [GeneExpression() for \_ in range(POPULATION\_SIZE)]

    x\_value = random.uniform(-10, 10)  # Random input to test optimization

    # Evolutionary loop

    for generation in range(GENERATIONS):

        fitnesses = [ind.fitness(x\_value) for ind in population]

        best\_idx = fitnesses.index(min(fitnesses))

        print(f"Generation {generation}: Best Fitness = {fitnesses[best\_idx]:.5f}")

        # Elitism: Preserve the best individual

        new\_population = [population[best\_idx]]

        # Create next generation

        while len(new\_population) < POPULATION\_SIZE:

            parent1 = selection(population, fitnesses)

            parent2 = selection(population, fitnesses)

            child1, child2 = crossover(parent1, parent2)

            mutate(child1)

            mutate(child2)

            new\_population.extend([child1, child2])

        population = new\_population

    # Final Solution

    final\_fitnesses = [ind.fitness(x\_value) for ind in population]

    best\_idx = final\_fitnesses.index(min(final\_fitnesses))

    print("\nOptimized Solution:")

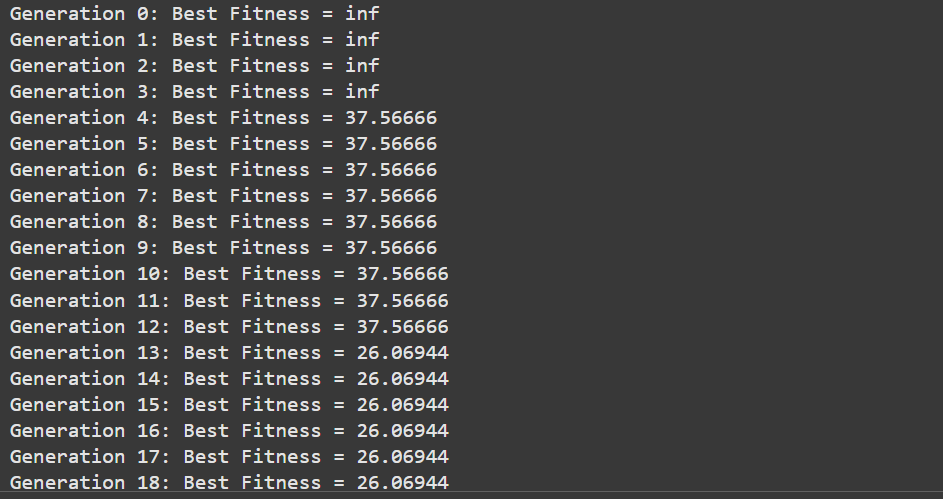
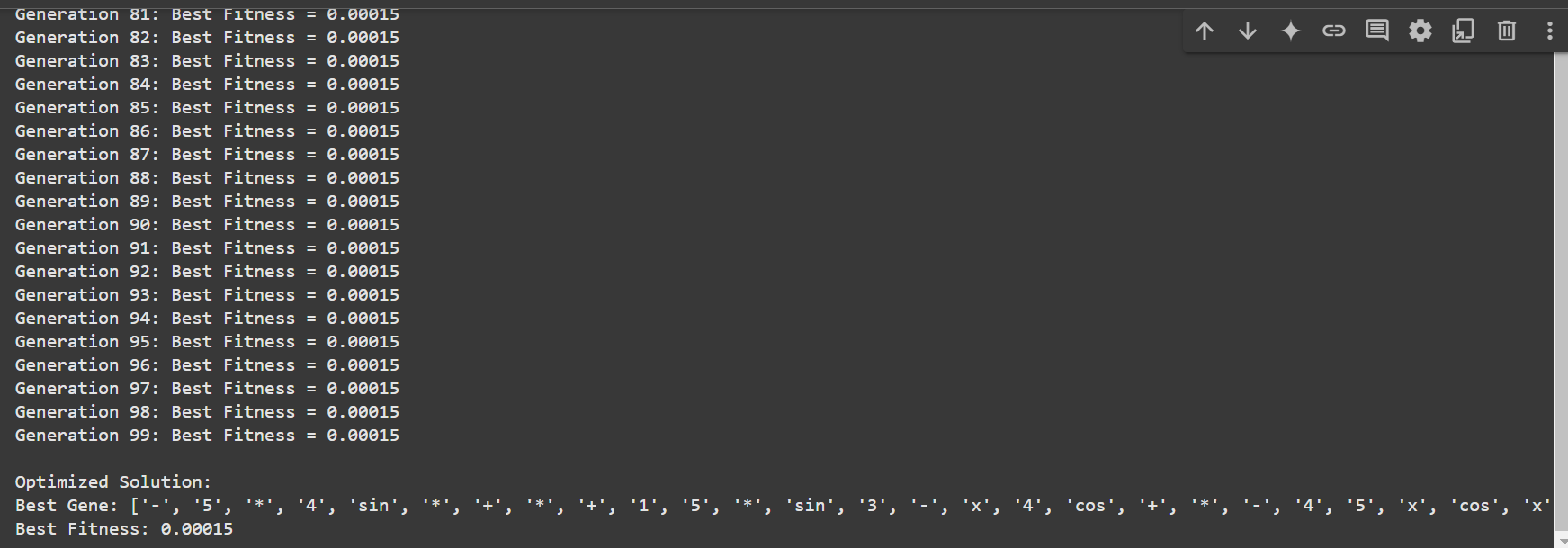
    print(f"Best Gene: {population[best\_idx].gene}")

    print(f"Best Fitness: {final\_fitnesses[best\_idx]:.5f}")

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

    geneExpression()

**Output:**

**** ****