# VISVESVARAYA TECHNOLOGICAL UNIVERSITY

**"JnanaSangama", Belgaum -590014, Karnataka.**

**LAB RECORD**

# Bio Inspired Systems (23CS5BSBIS)

***Submitted by***

**TANMAYI S BALIJA (1BM22CS359)**

***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 **Tanmayi S Balija (1BM22CS359),** 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.Syed Akram<br>Assistant Professor<br>Department of CSE, BMSCE | Dr.Kavitha Sooda<br>Professor & HOD<br>Department of CSE, BMSCE |

# Index

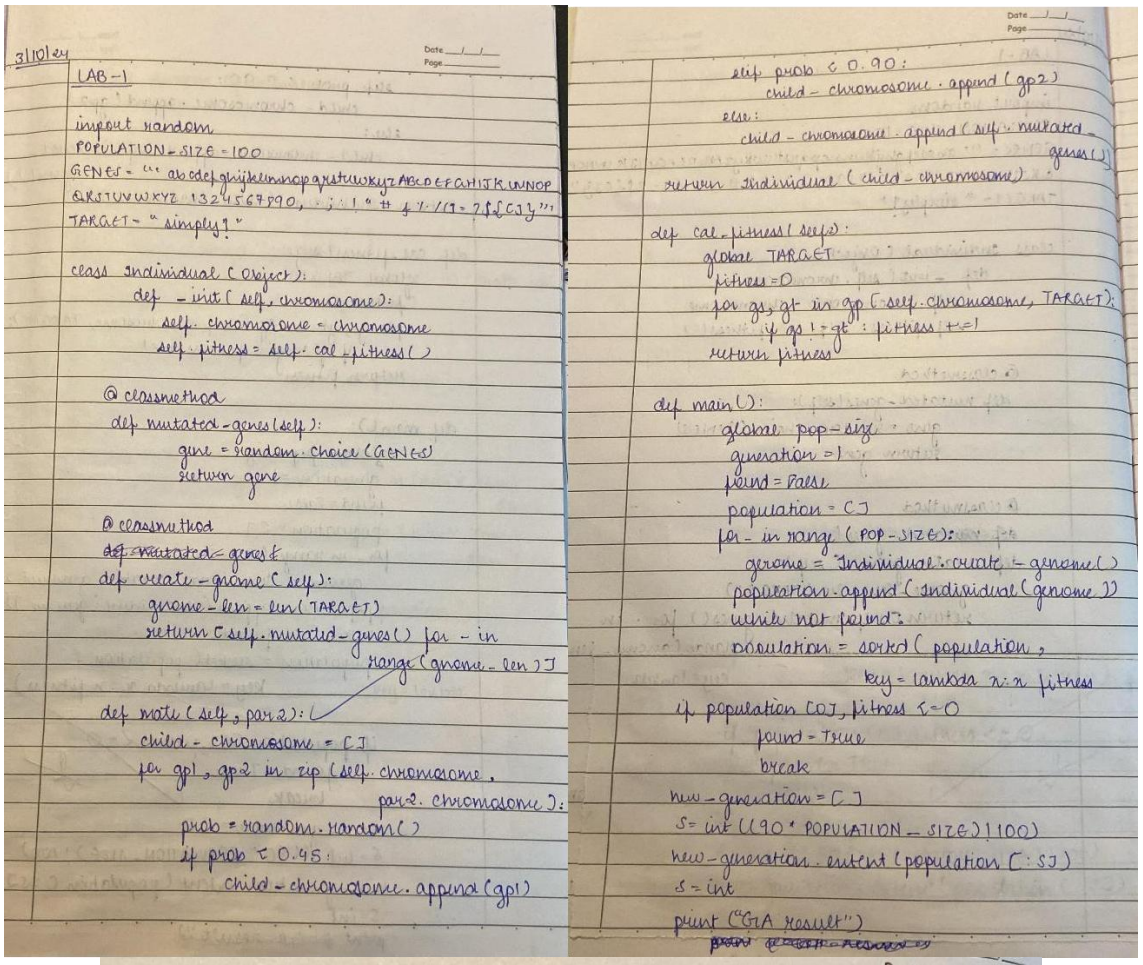


**Github Link**: https://github.com/tan04mayi/BIS

**Program 1**

Genetic Algorithms (GA) are inspired by the process of natural selection and genetics, where the fittest individuals are selected for reproduction to produce the next generation. GAs are widely used for solving optimization and search problems. Implement a Genetic Algorithm using Python to solve a basic optimization problem, such as finding the maximum value of a mathematical function.

**Algorithm**:



```
LAB-1

import random
POPULATION_SIZE = 100
GENES = ''' abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOP
QRSTUVWXYZ 1234567890, .;:!"#$%&/()=?@${[]}'''
TARGET = "simply!"

class Individual(object):
    def __init__(self, chromosome):
        self.chromosome = chromosome
        self.fitness = self.cal_fitness()

    @classmethod
    def mutated_genes(self):
        gene = random.choice(GENES)
        return gene

    @classmethod
    def create_gnome(self):
        gnome_len = len(TARGET)
        return [self.mutated_genes() for _ in
                range(gnome_len)]

    def mate(self, par2):
        child_chromosome = []
        for gp1, gp2 in zip(self.chromosome,
                            par2.chromosome):
            prob = random.random()
            if prob < 0.45:
                child_chromosome.append(gp1)
            elif prob < 0.90:
                child_chromosome.append(gp2)
            else:
                child_chromosome.append(self.mutated_genes())
        return Individual(child_chromosome)

    def cal_fitness(self):
        global TARGET
        fitness = 0
        for gs, gt in zip(self.chromosome, TARGET):
            if gs != gt: fitness += 1
        return fitness

def main():
    global POP_SIZE
    generation = 1
    found = False
    population = []
    for _ in range(POP_SIZE):
        genome = Individual.create_gnome()
        population.append(Individual(genome))
    while not found:
        population = sorted(population,
                            key = lambda x: x.fitness)
        if population[0].fitness <= 0:
            found = True
            break
        new_generation = []
        s = int((90*POPULATION_SIZE)/100)
        new_generation.extend(population[:s])
        s = int
        print("GA result")
```

**Output**:

generation 1: String : SL ; = $ y Fitness : 4
generation 2: String : SL ; = $ y fitness : 4
generation 3: String : SL ; = $ y fitness : 4
generation 4: String : SL ; $ y fitness : 4
generation 5: String : SL ; sleplm Fitness : 3
generation 6: String : SL ; STmplm Fitness : 2

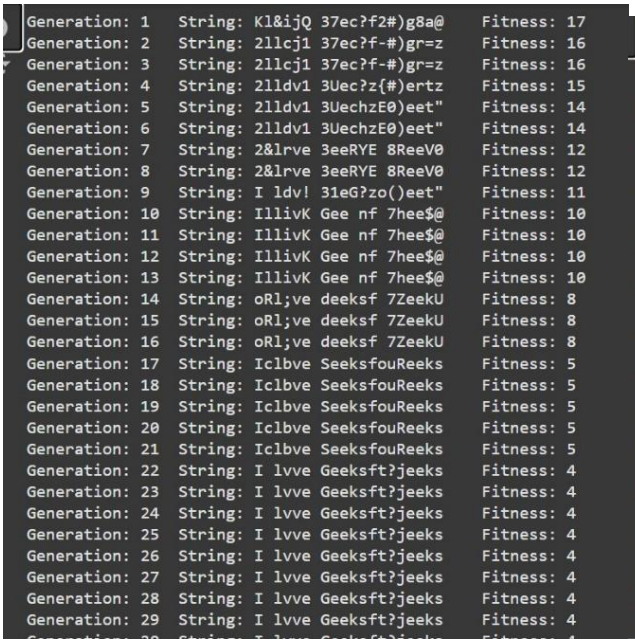
**Code:**

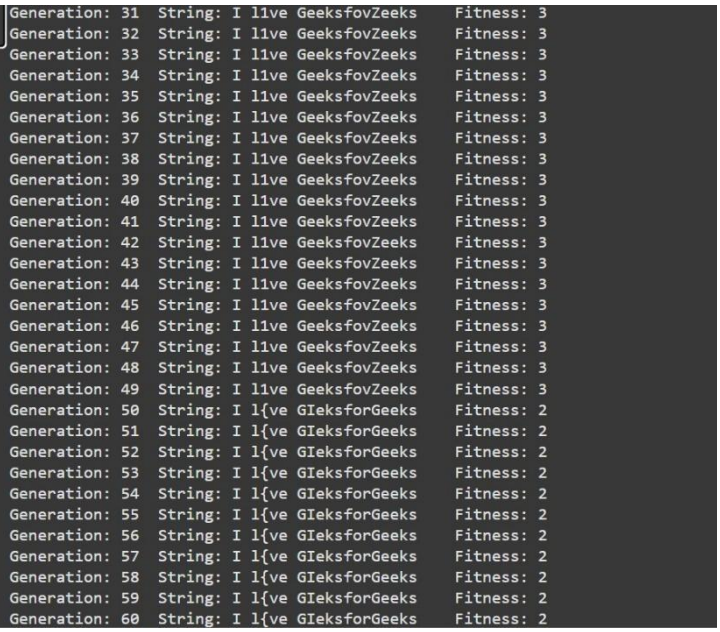
```
import random

POPULATION_SIZE = 100

GENES = '''abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOP
QRSTUVWXYZ 1234567890, .-;:_!"#%&/()=?@${[]}'''

TARGET = "I love GeeksforGeeks"

class Individual:
    '''
    Class representing an individual in the population.
    '''
    def _init_(self, chromosome):
        self.chromosome = chromosome
        self.fitness = self.calculate_fitness()

    @staticmethod
    def mutated_gene():
        '''
        Create a random gene for mutation.
        '''
        return random.choice(GENES)

    @staticmethod
    def create_gnome():
        '''
        Create a chromosome (list of genes) of the same length as the target string.
        '''
        return [Individual.mutated_gene() for _ in range(len(TARGET))]

    def mate(self, partner):
        '''
        Perform mating and produce a new offspring.
        '''
        child_chromosome = []
        for gene_self, gene_partner in zip(self.chromosome, partner.chromosome):
            prob = random.random()
            if prob < 0.45:
                child_chromosome.append(gene_self)
            elif prob < 0.90:
                child_chromosome.append(gene_partner)
            else:
                child_chromosome.append(Individual.mutated_gene())
        return Individual(child_chromosome)
```

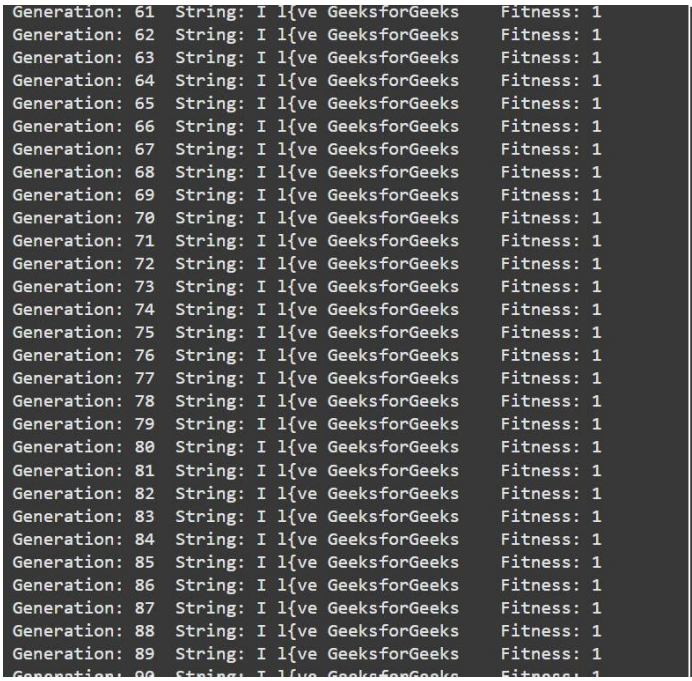
```
    def calculate_fitness(self):
        '''
        Calculate the fitness score as the number of differing characters.
        Lower fitness is better (0 is the optimal score).
        '''
        return sum(gene != target_gene for gene, target_gene in zip(self.chromosome, TARGET))

def main():
    generation = 1
    found = False
    population = [Individual(Individual.create_gnome()) for _ in range(POPULATION_SIZE)]

    while not found:
        # Sort the population by fitness (lower is better)
        population.sort(key=lambda individual: individual.fitness)

        # If the best individual has fitness 0, the target string is found
        if population[0].fitness == 0:
            found = True
            break

        # Create a new generation
        new_generation = []

        # Carry forward the top 10% of individuals
        top_individuals = int(0.1 * POPULATION_SIZE)
        new_generation.extend(population[:top_individuals])

        # Mating to fill the remaining 90% of the population
        remaining_population = POPULATION_SIZE - top_individuals
        for _ in range(remaining_population):
            parent1 = random.choice(population[:50])
            parent2 = random.choice(population[:50])
            child = parent1.mate(parent2)
            new_generation.append(child)

        population = new_generation

        # Print the best individual of the current generation
        best_individual = population[0]
        print(f"Generation: {generation}\tString: {".join(best_individual.chromosome)}\tFitness:
{best_individual.fitness}")

        generation += 1

    # Print the final generation and the best individual
    best_individual = population[0]
```

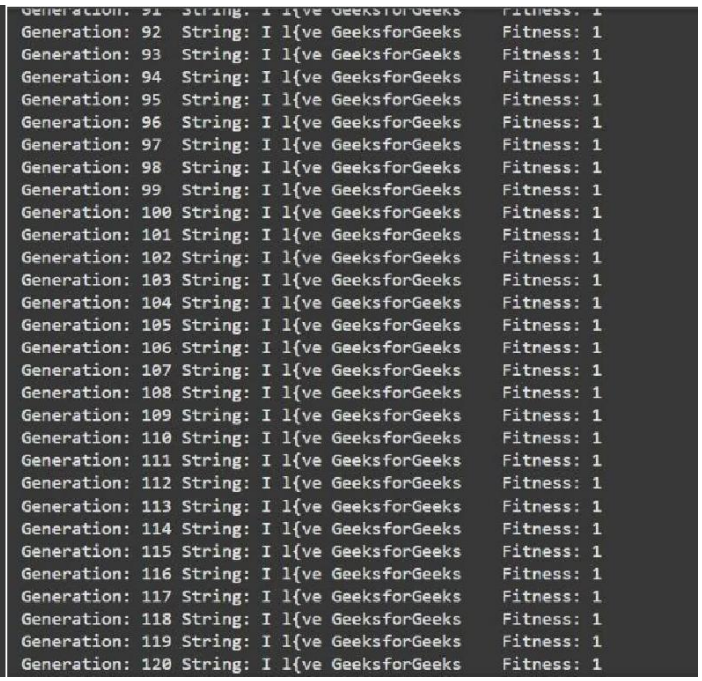
```
    print(f"Generation: {generation}\tString: {''.join(best_individual.chromosome)}\tFitness: {best_individual.fitness}")

if _name_ == "_main_":
    main()
```

**Output:**



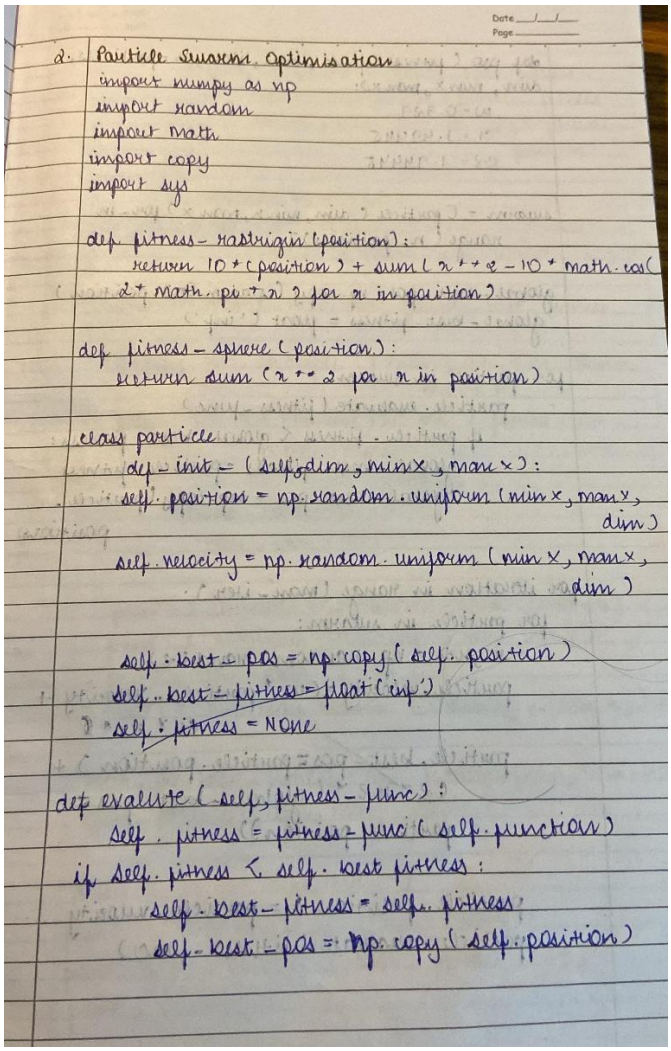



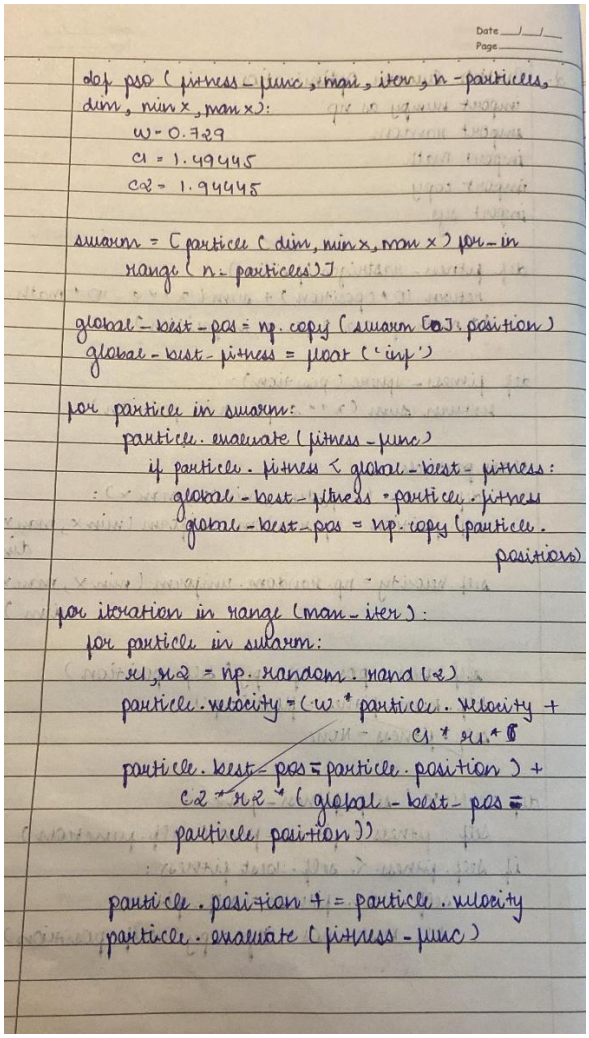
**Program 2:**

Particle Swarm Optimization (PSO) is inspired by the social behavior of birds flocking or fish schooling. PSO is used to find optimal solutions by iteratively improving a candidate solution with regard to a given measure of quality. Implement the PSO algorithm using Python to optimize a mathematical function.

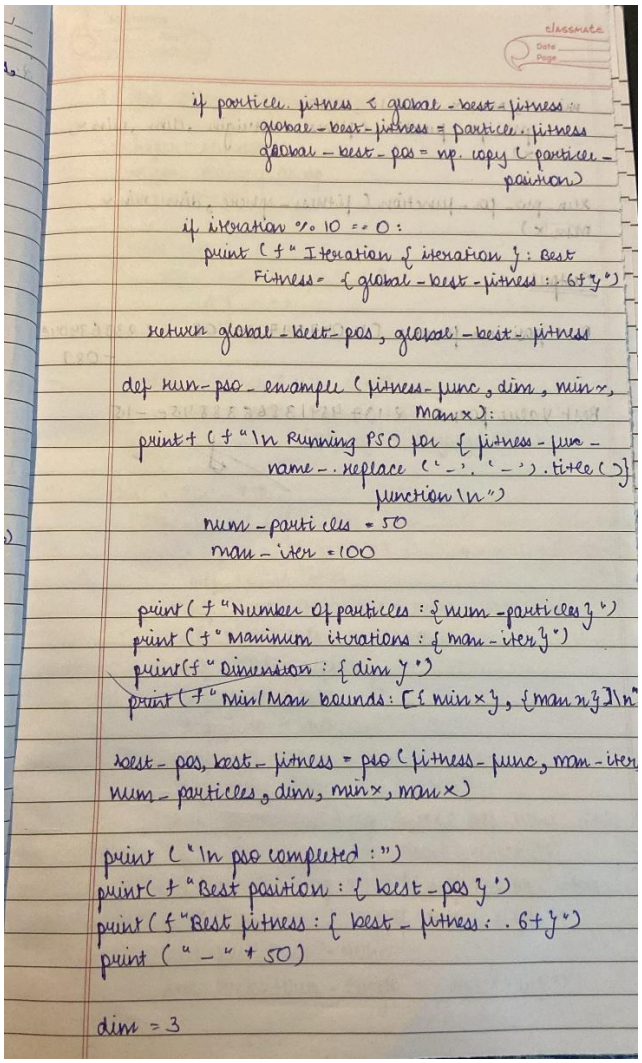
**Algorithm:**



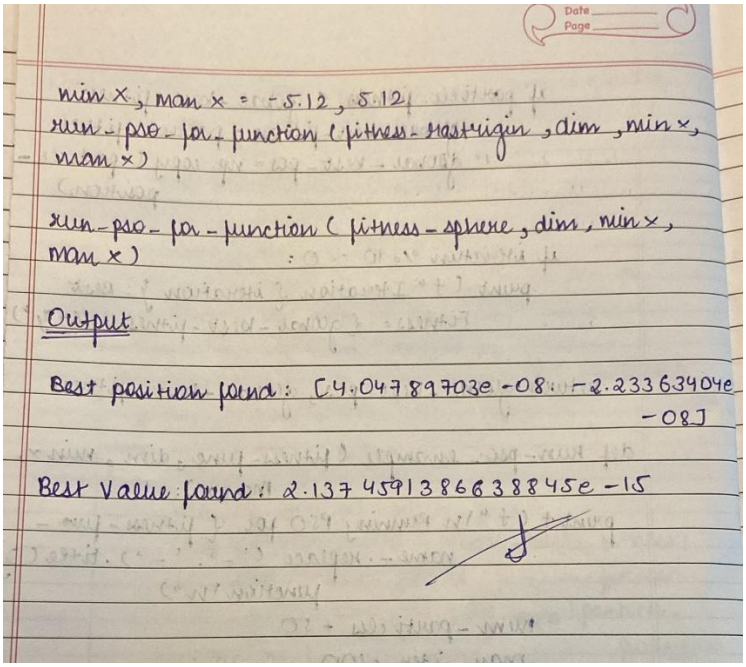
```
2. Particle Swarm Optimisation
import numpy as np
import random
import math
import copy
import sys

def fitness_rastrigin(position):
    return 10*(position) + sum( x**2 - 10*math.cos(
    2*math.pi*x ) for x in position)

def fitness_sphere(position):
    return sum(x**2 for x in position)

class particle
    def __init__(self,dim,minx,maxx):
    self.position = np.random.uniform(minx,maxx,
                                      dim)

    self.velocity = np.random.uniform(minx,maxx,
                                      dim)

    self.best_pos = np.copy(self.position)
    self.best_fitness = float('inf')
    self.fitness = None

def evaluate(self,fitness_func):
    self.fitness = fitness_func(self.function)
if self.fitness < self.best_fitness:
    self.best_fitness = self.fitness
    self.best_pos = np.copy(self.position)
```



```
def pso(fitness_func,iter,n_particles,
dim,minx,maxx):
    w = 0.729
    c1 = 1.49445
    c2 = 1.94445

swarm = [particle(dim,minx,maxx) for _in
    range(n_particles)]

global_best_pos = np.copy(swarm[0].position)
global_best_fitness = float('inf')

for particle in swarm:
    particle.evaluate(fitness_func)
    if particle.fitness < global_best_fitness:
        global_best_fitness = particle.fitness
        global_best_pos = np.copy(particle.
                                  position)

for iteration in range(max_iter):
    for particle in swarm:
        r1,r2 = np.random.rand(2)
        particle.velocity = (w*particle.velocity +
                             c1*r1*(
        particle.best_pos-particle.position) +
            c2*r2*(global_best_pos-
            particle.position))

    particle.position += particle.velocity
    particle.evaluate(fitness_func)
```



```
if particle.fitness < global_best_fitness:
    global_best_fitness = particle.fitness
    global_best_pos = np.copy(particle.position)

if iteration % 10 == 0:
    print(f"Iteration {iteration}: Best Fitness = {global_best_fitness:.6f}")

return global_best_pos, global_best_fitness

def run_pso_example(fitness_func, dim, minx, maxx):
    print(f"\nRunning PSO for {fitness_func.__name__.replace('_', ' ').title()} function\n")
    num_particles = 50
    max_iter = 100

    print(f"Number of particles: {num_particles}")
    print(f"Maximum iterations: {max_iter}")
    print(f"Dimension: {dim}")
    print(f"Min/Max bounds: [{minx}, {maxx}]\n")

    best_pos, best_fitness = pso(fitness_func, max_iter,
    num_particles, dim, minx, maxx)

    print("\nPSO completed:")
    print(f"Best position: {best_pos}")
    print(f"Best fitness: {best_fitness:.6f}")
    print("-" * 50)

dim = 3
```



```
minx, maxx = -5.12, 5.12
run_pso_for_function(fitness_rastrigin, dim, minx, maxx)

run_pso_for_function(fitness_sphere, dim, minx, maxx)
```

Output

```
Best position found: [4.047897e-08, -2.233634e-08]

Best value found: 2.137459138663885e-15
```

**Code:**

```
import random
import math
import copy
import sys

# Fitness functions
def fitness_rastrigin(position):
    return sum((xi * xi) - (10 * math.cos(2 * math.pi * xi)) + 10 for xi in position)

def fitness_sphere(position):
    return sum(xi * xi for xi in position)

# Particle class
class Particle:
    def __init__(self, fitness, dim, minx, maxx, seed):
        self.rnd = random.Random(seed)
        self.position = [(maxx - minx) * self.rnd.random() + minx for _ in range(dim)]
```

```
        self.velocity = [(maxx - minx) * self.rnd.random() + minx for _ in range(dim)]
        self.best_part_pos = self.position[:]
        self.fitness = fitness(self.position)
        self.best_part_fitnessVal = self.fitness

# PSO function
def pso(fitness, max_iter, n, dim, minx, maxx):
    w, c1, c2 = 0.729, 1.49445, 1.49445
    rnd = random.Random(0)
    swarm = [Particle(fitness, dim, minx, maxx, i) for i in range(n)]


    best_swarm_pos, best_swarm_fitnessVal = [0.0] * dim, sys.float_info.max
    for p in swarm:
        if p.fitness < best_swarm_fitnessVal:
            best_swarm_fitnessVal = p.fitness
            best_swarm_pos = p.position[:]

    for Iter in range(max_iter):
        if Iter % 10 == 0 and Iter > 1:
            print(f"Iter = {Iter} best fitness = {best_swarm_fitnessVal:.3f}")

        for p in swarm:
            for k in range(dim):
                r1, r2 = rnd.random(), rnd.random()
                p.velocity[k] = w * p.velocity[k] + c1 * r1 * (p.best_part_pos[k] - p.position[k]) + c2 * r2 *
(best_swarm_pos[k] - p.position[k])
                p.velocity[k] = max(min(p.velocity[k], maxx), minx)

            p.position = [p.position[k] + p.velocity[k] for k in range(dim)]
            p.fitness = fitness(p.position)

            if p.fitness < p.best_part_fitnessVal:
                p.best_part_fitnessVal = p.fitness
                p.best_part_pos = p.position[:]

            if p.fitness < best_swarm_fitnessVal:
                best_swarm_fitnessVal = p.fitness
                best_swarm_pos = p.position[:]

    return best_swarm_pos

# Driver for Rastrigin function
def run_pso(fitness, dim, minx, maxx):
    print(f"Goal is to minimize the function in {dim} variables")
    print(f"Function has known min = 0.0 at ({', '.join(['0'] * (dim - 1))}, 0)")

    num_particles, max_iter = 50, 100
```

```
    best_position = pso(fitness, max_iter, num_particles, dim, minx, maxx)

    print(f"Best solution found: {', '.join([f'{x:.6f}' for x in best_position])}")
    print(f"Fitness of best solution = {fitness(best_position):.6f}\n")

# Run PSO for Rastrigin and Sphere functions
print("\nBegin PSO for Rastrigin function\n")
run_pso(fitness_rastrigin, 3, -10.0, 10.0)

print("\nBegin PSO for Sphere function\n")
run_pso(fitness_sphere, 3, -10.0, 10.0)
```

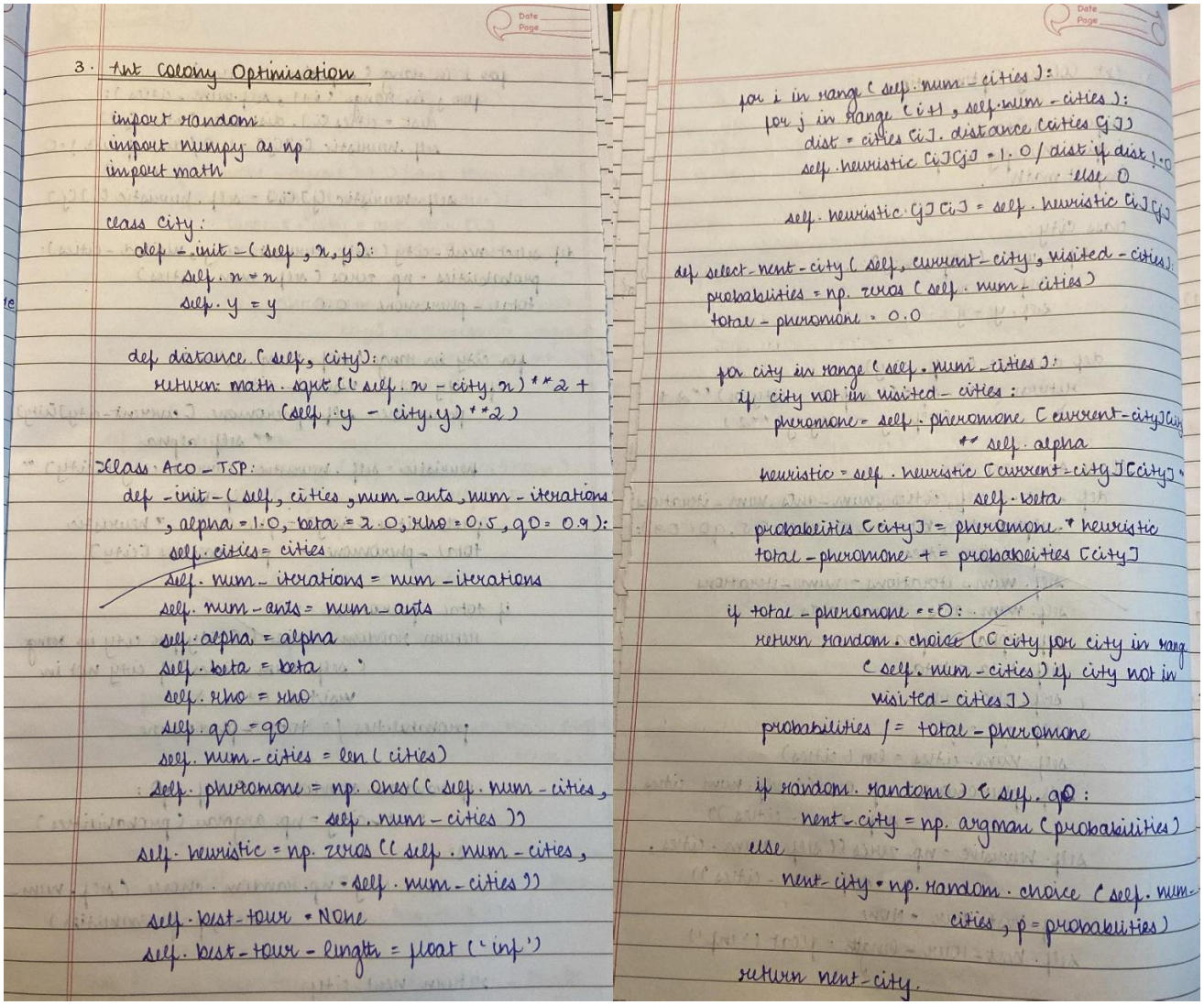
**Output:**

```
Begin PSO for Rastrigin function

Goal is to minimize the function in 3 variables
Function has known min = 0.0 at (0, 0, 0)
Iter = 10 best fitness = 10.262
Iter = 20 best fitness = 5.113
Iter = 30 best fitness = 2.217
Iter = 40 best fitness = 0.031
Iter = 50 best fitness = 0.001
Iter = 60 best fitness = 0.000
Iter = 70 best fitness = 0.000
Iter = 80 best fitness = 0.000
Iter = 90 best fitness = 0.000
Best solution found: 0.000012, 0.000006, 0.000111
Fitness of best solution = 0.000002


Begin PSO for Sphere function

Goal is to minimize the function in 3 variables
Function has known min = 0.0 at (0, 0, 0)
Iter = 10 best fitness = 0.182
Iter = 20 best fitness = 0.002
Iter = 30 best fitness = 0.000
Iter = 40 best fitness = 0.000
Iter = 50 best fitness = 0.000
Iter = 60 best fitness = 0.000
Iter = 70 best fitness = 0.000
Iter = 80 best fitness = 0.000
Iter = 90 best fitness = 0.000
Best solution found: 0.000005, 0.000000, 0.000005
Fitness of best solution = 0.000000
```

**Program 3:**

The foraging behavior of ants has inspired the development of optimization algorithms that can solve complex problems such as the Traveling Salesman Problem (TSP). Ant Colony Optimization (ACO) simulates the way ants find the shortest path between food sources and their nest. Implement the ACO algorithm using Python to solve the TSP, where the objective is to find the shortest possible route that visits a list of cities and returns to the origin city.

**Algorithm:**



```
3. Ant Colony Optimisation

import random
import numpy as np
import math

class City:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def distance(self, city):
        return: math.sqrt((self.x - city.x)**2 +
                (self.y - city.y)**2)

class ACO_TSP:
    def __init__(self, cities, num_ants, num_iterations
        , alpha=1.0, beta=2.0, rho=0.5, q0=0.9):
        self.cities = cities
        self.num_iterations = num_iterations
        self.num_ants = num_ants
        self.alpha = alpha
        self.beta = beta
        self.rho = rho
        self.q0 = q0
        self.num_cities = len(cities)
        self.pheromone = np.ones((self.num_cities,
                         self.num_cities))
        self.heuristic = np.zeros((self.num_cities,
                         self.num_cities))
        self.best_tour = None
        self.best_tour_length = float('inf')

    for i in range(self.num_cities):
        for j in range(i+1, self.num_cities):
            dist = cities[i].distance(cities[j])
            self.heuristic[i][j] = 1.0/dist if dist!=0
                                   else 0
            self.heuristic[j][i] = self.heuristic[i][j]

    def select_next_city(self, current_city, visited_cities):
        probabilities = np.zeros(self.num_cities)
        total_pheromone = 0.0

        for city in range(self.num_cities):
            if city not in visited_cities:
                pheromone = self.pheromone[current_city][city]
                            ** self.alpha
                heuristic = self.heuristic[current_city][city]**
                            self.beta
                probabilities[city] = pheromone * heuristic
                total_pheromone += probabilities[city]

        if total_pheromone == 0:
            return random.choice([city for city in range
                (self.num_cities) if city not in
                visited_cities])
        probabilities /= total_pheromone

        if random.random() < self.q0:
            next_city = np.argmax(probabilities)
        else
            next_city = np.random.choice(self.num_
                        cities, p=probabilities)

        return next_city
```

```
def update_pheromone(self, ants):
    self.pheromone *= (1 - self.rho)

    for ant in ants:
        pheromone_deposit = 1.0 / ant.tour_length
        for i in range(self.num_cities):
            current_city = ant.tour[i]
            next_city = ant.tour[(i+1) % self.
                                 num_cities]
            self.pheromone[current_city][next_
                                 city] += pheromone_deposit
            self.pheromone[next_city][current_city]
                                 += pheromone_deposit

def run(self):
    for iteration in range(self.num_iterations):
        ants = [Ant(self.num_cities, self) for _ in
                range(self.num_ants)]
        for ant in ants:
            ant.construct_solution()

        self.update_pheromone(ants)

        for ant in ants:
            if ant.tour_length < self.best_tour_length:
                self.best_tour_length = ant.tour_length
                self.best_tour = ant.tour

        print(f"Iteration {iteration+1}/{self.num_
                iterations}: Best Tour length = {self.best_
                tour_length}")

    return self.best_tour, self.best_tour_length
```

```
class Ant:
    def __init__(self, num_cities, aco_tsp):
        self.num_cities = num_cities
        self.aco_tsp = aco_tsp
        self.tour = []
        self.tour_length = 0.0

    def construct_solution(self):
        start_city = random.randint(0, self.num_cities-1)
        self.tour = [start_city]
        self.tour_length = 0.0
        visited = set(self.tour)

        current_city = start_city
        while len(self.tour) < self.num_cities:
            next_city = self.aco_tsp.select_next_city(
                current_city, visited)
            self.tour.append(next_city)
            visited.add(next_city)
            self.tour_length += self.aco_tsp.cities[current_
                city].distance(self.aco_tsp.cities[next_city])
            current_city = next_city

        self.tour_length += self.aco_tsp.cities[self.tour[
            -1]].distance(self.aco_tsp.cities[self.tour[0]])

if __name__ == "__main__":
    cities = [City(0,0), City(1,3), City(4,3), City(6,1),
              City(3,0)]

    aco = ACO_TSP(cities=cities, num_ants=10,
                  num_iterations=100, alpha=1.0, beta=2.0,
                  rho=0.5, q0=0.9)
```

```
    best_tour, best_tour_length = aco.run()

    print("\nBest tour found:", best_tour)
    print("Best tour length:", best_tour_length)
```

Output

```
Best tour found: [3, np.int64(2), np.int64(1),
    np.int64(0), np.int64(4)]

Best tour length: 15.152928.
```

**Code:**

```
import random
import numpy as np
import math

# Define the Problem: Cities with coordinates (can be modified with real data)
class City:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def distance(self, city):
        return math.sqrt((self.x - city.x)**2 + (self.y - city.y)**2)

# Ant Colony Optimization for TSP
class ACO_TSP:
    def __init__(self, cities, num_ants, num_iterations, alpha=1.0, beta=2.0, rho=0.5, q0=0.9):
        self.cities = cities
        self.num_ants = num_ants
        self.num_iterations = num_iterations
        self.alpha = alpha  # Importance of pheromone
        self.beta = beta    # Importance of heuristic (distance)
        self.rho = rho      # Pheromone evaporation rate
        self.q0 = q0        # Probability of choosing the best path
        self.num_cities = len(cities)
        self.pheromone = np.ones((self.num_cities, self.num_cities))  # Initial pheromone values
        self.heuristic = np.zeros((self.num_cities, self.num_cities))  # Heuristic info (inverse of distance)
        self.best_tour = None
        self.best_tour_length = float('inf')

        # Compute heuristic (inverse of distance)
        for i in range(self.num_cities):
            for j in range(i + 1, self.num_cities):
                dist = cities[i].distance(cities[j])
                self.heuristic[i][j] = 1.0 / dist if dist != 0 else 0
                self.heuristic[j][i] = self.heuristic[i][j]

    def select_next_city(self, current_city, visited_cities):
        probabilities = np.zeros(self.num_cities)
        total_pheromone = 0.0

        # Calculate the transition probabilities
        for city in range(self.num_cities):
            if city not in visited_cities:
                pheromone = self.pheromone[current_city][city] ** self.alpha
                heuristic = self.heuristic[current_city][city] ** self.beta
                probabilities[city] = pheromone * heuristic
                total_pheromone += probabilities[city]
```

```
        # Normalize probabilities
        if total_pheromone == 0:
            return random.choice([city for city in range(self.num_cities) if city not in visited_cities])

        probabilities /= total_pheromone

        # Exploration vs Exploitation
        if random.random() < self.q0:
            # Exploitation: choose the city with the highest probability
            next_city = np.argmax(probabilities)
        else:
            # Exploration: choose based on probabilities
            next_city = np.random.choice(self.num_cities, p=probabilities)

        return next_city

    def update_pheromone(self, ants):
        # Evaporate pheromone
        self.pheromone *= (1 - self.rho)

        # Deposit pheromone based on the ants' solutions
        for ant in ants:
            pheromone_deposit = 1.0 / ant.tour_length
            for i in range(self.num_cities):
                current_city = ant.tour[i]
                next_city = ant.tour[(i + 1) % self.num_cities]
                self.pheromone[current_city][next_city] += pheromone_deposit
                self.pheromone[next_city][current_city] += pheromone_deposit

    def run(self):
        for iteration in range(self.num_iterations):
            ants = [Ant(self.num_cities, self) for _ in range(self.num_ants)]
            for ant in ants:
                ant.construct_solution()

            # Update pheromones
            self.update_pheromone(ants)

            # Update the best solution found so far
            for ant in ants:
                if ant.tour_length < self.best_tour_length:
                    self.best_tour_length = ant.tour_length
                    self.best_tour = ant.tour

            print(f"Iteration {iteration + 1}/{self.num_iterations}: Best Tour Length = {self.best_tour_length}")

        return self.best_tour, self.best_tour_length
```

```
# Ant class to simulate each ant's behavior
class Ant:
    def __init__(self, num_cities, aco_tsp):
        self.num_cities = num_cities
        self.aco_tsp = aco_tsp
        self.tour = []
        self.tour_length = 0.0

    def construct_solution(self):
        start_city = random.randint(0, self.num_cities - 1)
        self.tour = [start_city]
        self.tour_length = 0.0
        visited_cities = set(self.tour)

        current_city = start_city
        while len(self.tour) < self.num_cities:
            next_city = self.aco_tsp.select_next_city(current_city, visited_cities)
            self.tour.append(next_city)
            visited_cities.add(next_city)
            self.tour_length += self.aco_tsp.cities[current_city].distance(self.aco_tsp.cities[next_city])
            current_city = next_city

        # Add the return to the starting city
        self.tour_length += self.aco_tsp.cities[self.tour[-1]].distance(self.aco_tsp.cities[self.tour[0]])

# Example usage

if __name__ == "__main__":
    # Define cities (x, y coordinates)
    cities = [City(0, 0), City(1, 3), City(4, 3), City(6, 1), City(3, 0)]

    # Initialize and run ACO
    aco = ACO_TSP(cities=cities, num_ants=10, num_iterations=100, alpha=1.0, beta=2.0, rho=0.5, q0=0.9)
    best_tour, best_tour_length = aco.run()

    # Output the best tour and its length
    print("\nBest tour found:", best_tour)
    print("Best tour length:", best_tour_length)
```

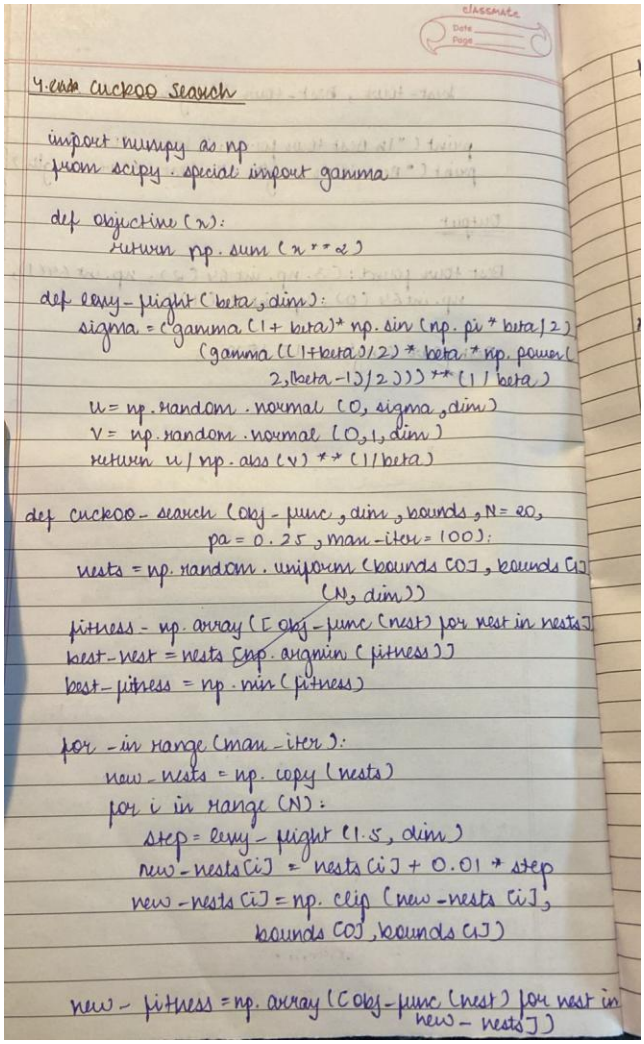
**Output:**

```
Iteration 1/100: Best Tour Length = 15.15298244508295
Iteration 2/100: Best Tour Length = 15.15298244508295
Iteration 3/100: Best Tour Length = 15.15298244508295
Iteration 4/100: Best Tour Length = 15.15298244508295
Iteration 5/100: Best Tour Length = 15.15298244508295
Iteration 6/100: Best Tour Length = 15.15298244508295
Iteration 7/100: Best Tour Length = 15.15298244508295
Iteration 8/100: Best Tour Length = 15.15298244508295
Iteration 9/100: Best Tour Length = 15.15298244508295
Iteration 10/100: Best Tour Length = 15.15298244508295
Iteration 11/100: Best Tour Length = 15.15298244508295
Iteration 12/100: Best Tour Length = 15.15298244508295
Iteration 13/100: Best Tour Length = 15.15298244508295
Iteration 14/100: Best Tour Length = 15.15298244508295
Iteration 15/100: Best Tour Length = 15.15298244508295
Iteration 16/100: Best Tour Length = 15.15298244508295
Iteration 17/100: Best Tour Length = 15.15298244508295
Iteration 18/100: Best Tour Length = 15.15298244508295
Iteration 19/100: Best Tour Length = 15.15298244508295
Iteration 20/100: Best Tour Length = 15.15298244508295
Iteration 21/100: Best Tour Length = 15.15298244508295
Iteration 22/100: Best Tour Length = 15.15298244508295
Iteration 23/100: Best Tour Length = 15.15298244508295
Iteration 24/100: Best Tour Length = 15.15298244508295
Iteration 25/100: Best Tour Length = 15.15298244508295
Iteration 26/100: Best Tour Length = 15.15298244508295
Iteration 27/100: Best Tour Length = 15.15298244508295
Iteration 28/100: Best Tour Length = 15.15298244508295
Iteration 29/100: Best Tour Length = 15.15298244508295
Iteration 30/100: Best Tour Length = 15.15298244508295
Iteration 31/100: Best Tour Length = 15.15298244508295
Iteration 32/100: Best Tour Length = 15.15298244508295
Iteration 33/100: Best Tour Length = 15.15298244508295
Iteration 34/100: Best Tour Length = 15.15298244508295
Iteration 35/100: Best Tour Length = 15.15298244508295
Iteration 36/100: Best Tour Length = 15.15298244508295
Iteration 37/100: Best Tour Length = 15.15298244508295
Iteration 38/100: Best Tour Length = 15.15298244508295
Iteration 39/100: Best Tour Length = 15.15298244508295
Iteration 40/100: Best Tour Length = 15.15298244508295
Iteration 41/100: Best Tour Length = 15.15298244508295
Iteration 42/100: Best Tour Length = 15.15298244508295
Iteration 43/100: Best Tour Length = 15.15298244508295
Iteration 44/100: Best Tour Length = 15.15298244508295
Iteration 45/100: Best Tour Length = 15.15298244508295
Iteration 46/100: Best Tour Length = 15.15298244508295
Iteration 47/100: Best Tour Length = 15.15298244508295
Iteration 48/100: Best Tour Length = 15.15298244508295
Iteration 49/100: Best Tour Length = 15.15298244508295
Iteration 50/100: Best Tour Length = 15.15298244508295
Iteration 51/100: Best Tour Length = 15.15298244508295
Iteration 52/100: Best Tour Length = 15.15298244508295
Iteration 53/100: Best Tour Length = 15.15298244508295
Iteration 54/100: Best Tour Length = 15.15298244508295
Iteration 55/100: Best Tour Length = 15.15298244508295
Iteration 56/100: Best Tour Length = 15.15298244508295
Iteration 57/100: Best Tour Length = 15.15298244508295
Iteration 58/100: Best Tour Length = 15.15298244508295
Iteration 59/100: Best Tour Length = 15.15298244508295
Iteration 60/100: Best Tour Length = 15.15298244508295
Iteration 61/100: Best Tour Length = 15.15298244508295
Iteration 62/100: Best Tour Length = 15.15298244508295
Iteration 63/100: Best Tour Length = 15.15298244508295
Iteration 64/100: Best Tour Length = 15.15298244508295
Iteration 65/100: Best Tour Length = 15.15298244508295
Iteration 66/100: Best Tour Length = 15.15298244508295
Iteration 67/100: Best Tour Length = 15.15298244508295
Iteration 68/100: Best Tour Length = 15.15298244508295
Iteration 69/100: Best Tour Length = 15.15298244508295
Iteration 70/100: Best Tour Length = 15.15298244508295
Iteration 71/100: Best Tour Length = 15.15298244508295
Iteration 72/100: Best Tour Length = 15.15298244508295
Iteration 73/100: Best Tour Length = 15.15298244508295
Iteration 74/100: Best Tour Length = 15.15298244508295
Iteration 75/100: Best Tour Length = 15.15298244508295
Iteration 76/100: Best Tour Length = 15.15298244508295
Iteration 77/100: Best Tour Length = 15.15298244508295
Iteration 78/100: Best Tour Length = 15.15298244508295
Iteration 79/100: Best Tour Length = 15.15298244508295
Iteration 80/100: Best Tour Length = 15.15298244508295
Iteration 81/100: Best Tour Length = 15.15298244508295
Iteration 82/100: Best Tour Length = 15.15298244508295
Iteration 83/100: Best Tour Length = 15.15298244508295
Iteration 84/100: Best Tour Length = 15.15298244508295
Iteration 85/100: Best Tour Length = 15.15298244508295
Iteration 86/100: Best Tour Length = 15.15298244508295
Iteration 87/100: Best Tour Length = 15.15298244508295
Iteration 88/100: Best Tour Length = 15.15298244508295
Iteration 89/100: Best Tour Length = 15.15298244508295
Iteration 90/100: Best Tour Length = 15.15298244508295
Iteration 91/100: Best Tour Length = 15.15298244508295
Iteration 92/100: Best Tour Length = 15.15298244508295
Iteration 93/100: Best Tour Length = 15.15298244508295
Iteration 94/100: Best Tour Length = 15.15298244508295
Iteration 95/100: Best Tour Length = 15.15298244508295
Iteration 96/100: Best Tour Length = 15.15298244508295
Iteration 97/100: Best Tour Length = 15.15298244508295
Iteration 98/100: Best Tour Length = 15.15298244508295
Iteration 99/100: Best Tour Length = 15.15298244508295
Iteration 100/100: Best Tour Length = 15.15298244508295

Best tour found: [1, np.int64(2), np.int64(3), np.int64(4), np.int64(0)]
Best tour length: 15.15298244508295
```

**Program 4:**

Cuckoo Search (CS) is a nature-inspired optimization algorithm based on the brood parasitism of some cuckoo species. This behavior involves laying eggs in the nests of other birds, leading to the optimization of survival strategies. CS uses Lévy flights to generate new solutions, promoting global search capabilities and avoiding local minima. The algorithm is widely used for solving continuous optimization problems and has applications in various domains, including engineering design, machine learning, and data mining.

**Algorithm:**



```
4.Cuckoo Search

import numpy as np
from scipy.special import gamma

def objective(x):
    return np.sum(x**2)

def levy_flight(beta, dim):
    sigma = (gamma(1+beta)*np.sin(np.pi*beta/2)
             (gamma((1+beta)/2)*beta*np.power(
             2,(beta-1)/2)))**(1/beta)
    u = np.random.normal(0, sigma, dim)
    v = np.random.normal(0,1,dim)
    return u/np.abs(v)**(1/beta)

def cuckoo_search(obj_func, dim, bounds, N=20,
                  pa=0.25, max_iter=100):
    nests = np.random.uniform(bounds[0], bounds[1],
                              (N, dim))
    fitness = np.array([obj_func(nest) for nest in nests])
    best_nest = nests[np.argmin(fitness)]
    best_fitness = np.min(fitness)

    for _ in range(max_iter):
        new_nests = np.copy(nests)
        for i in range(N):
            step = levy_flight(1.5, dim)
            new_nests[i] = nests[i] + 0.01 * step
            new_nests[i] = np.clip(new_nests[i],
                                   bounds[0], bounds[1])

        new_fitness = np.array([obj_func(nest) for nest in
                                new_nests])
        for i in range(N):
            if new_fitness[i] < fitness[i]:
                nests[i] = new_nests[i]
                fitness[i] = new_fitness[i]

        best_nest_idx = np.argmin(fitness)
        best_nest = nests[best_nest_idx]
        best_fitness = fitness[best_nest_idx]

    return best_nest, best_fitness

dim = 10
bounds = [-5, 5]
best_nest, best_fitness = cuckoo_search(objective,
                                        dim, bounds)

print(f"Best Nest: {best_nest}")
print(f"Best fitness: {best_fitness}")
```

Output

```
Best Nest: [-2.7162  0.5995  1.1431 -0.7824
-0.3176 -0.2455  1.5899  2.4114 -4.0075  1.388]

Best fitness: 36.14801947
```

**Code:**

```
import numpy as np
from scipy.special import gamma

# Objective Function (for example, Sphere function)
def objective(x):
    return np.sum(x**2)

# Lévy flight function
def levy_flight(beta, dim):
    sigma = (gamma(1+beta)*np.sin(np.pi*beta/2) /
             (gamma((1+beta)/2)*beta*np.power(2, (beta-1)/2)))**(1/beta)
    u = np.random.normal(0, sigma, dim)
    v = np.random.normal(0, 1, dim)
    return u / np.abs(v)**(1/beta)

# Cuckoo Search algorithm
def cuckoo_search(obj_func, dim, bounds, N=20, pa=0.25, max_iter=100):
    nests = np.random.uniform(bounds[0], bounds[1], (N, dim))
    fitness = np.array([obj_func(nest) for nest in nests])
    best_nest = nests[np.argmin(fitness)]
    best_fitness = np.min(fitness)

    for _ in range(max_iter):
        # Generate new solutions using Lévy flights
        new_nests = np.copy(nests)
        for i in range(N):
            step = levy_flight(1.5, dim)  # Lévy exponent 1.5
            new_nests[i] = nests[i] + 0.01 * step
            new_nests[i] = np.clip(new_nests[i], bounds[0], bounds[1])  # Bound the new nest position

        # Evaluate new solutions
        new_fitness = np.array([obj_func(nest) for nest in new_nests])

        # Abandon worst nests and replace with new ones
        for i in range(N):
            if np.random.rand() < pa and new_fitness[i] < fitness[i]:
                nests[i] = new_nests[i]
                fitness[i] = new_fitness[i]

        # Update the best solution
        best_nest_idx = np.argmin(fitness)
        best_nest = nests[best_nest_idx]
        best_fitness = fitness[best_nest_idx]

    return best_nest, best_fitness
```

```
# Parameters
dim = 10
bounds = [-5, 5]
best_nest, best_fitness = cuckoo_search(objective, dim, bounds)

print(f"Best Nest: {best_nest}")
print(f"Best Fitness: {best_fitness}")
```

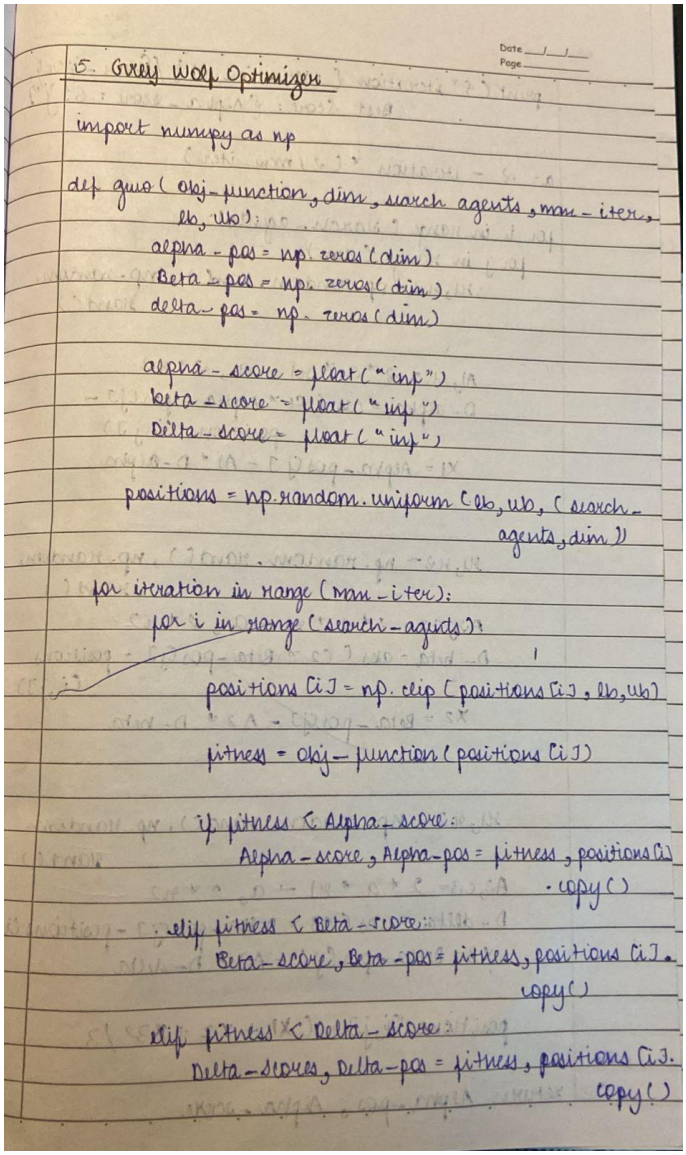
**Output:**

```
Best Nest: [ 2.7008626  -1.75838593 -2.58232104  0.74937546 -1.00344901 -0.26175236
 -2.21050897 -2.06340349  1.29407781  0.82913262]
Best Fitness: 30.19803175808211
```

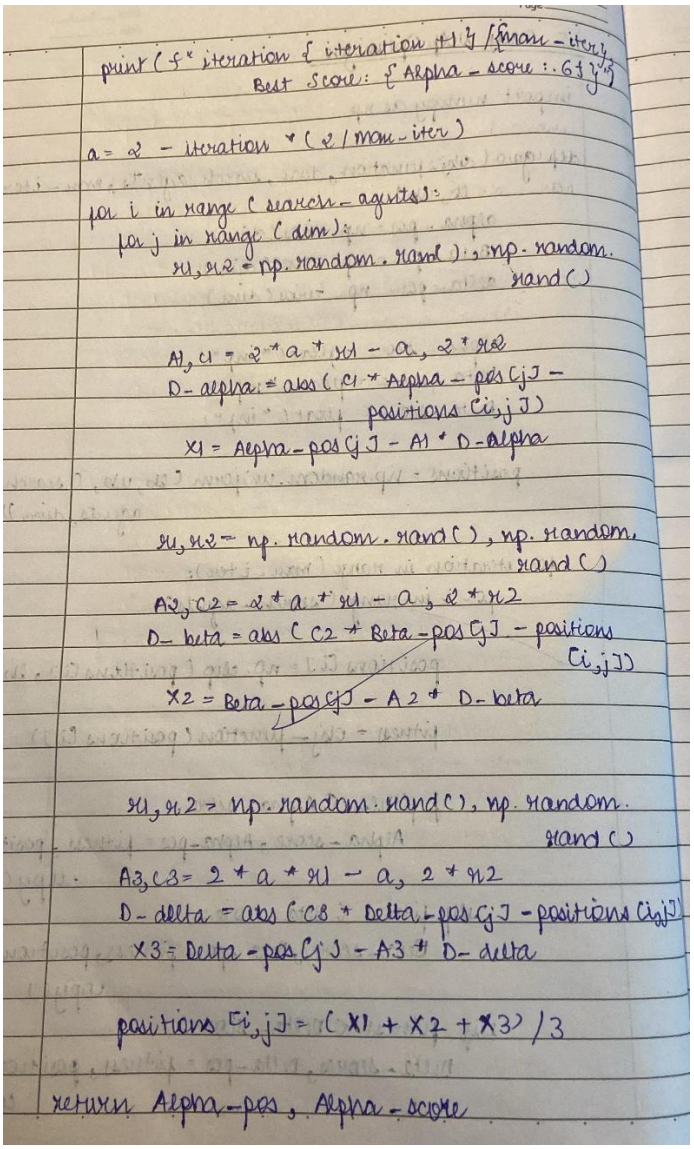
**Program 5:**

The Grey Wolf Optimizer (GWO) algorithm is a swarm intelligence algorithm inspired by the social hierarchy and hunting behavior of grey wolves. It mimics the leadership structure of alpha, beta, delta, and omega wolves and their collaborative hunting strategies. The GWO algorithm uses these social hierarchies to model the optimization process, where the alpha wolves guide the search process while beta and delta wolves assist in refining the search direction. This algorithm is effective for continuous optimization problems and has applications in engineering, data analysis, and machine learning.

**Algorithm:**



```
5. Grey Wolf Optimizer

import numpy as np

def gwo(obj_function, dim, search_agents, max_iter, lb, ub):
    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")

    positions = np.random.uniform(lb, ub, (search_agents, dim))

    for iteration in range(max_iter):
        for i in range(search_agents):

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

            fitness = obj_function(positions[i,j])

            if fitness < Alpha_score:
                Alpha_score, Alpha_pos = fitness, positions[i].copy()

            elif fitness < Beta_score:
                Beta_score, Beta_pos = fitness, positions[i].copy()

            elif fitness < Delta_score:
                Delta_score, Delta_pos = fitness, positions[i].copy()
```



```
            print("iteration {iteration+1}/{max_iter}")
            Best Score: {Alpha_score:.6f}")

        a = 2 - iteration * (2 / max_iter)

        for i in range(search_agents):
            for j in range(dim):
                r1, r2 = np.random.rand(), np.random.rand()

                A1, C1 = 2 * a * r1 - a, 2 * r2
                D_alpha = abs(C1 * Alpha_pos[j] - positions[i,j])
                X1 = Alpha_pos[j] - A1 * D_alpha

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

                A2, C2 = 2 * a * r1 - a, 2 * r2
                D_beta = abs(C2 * Beta_pos[j] - positions[i,j])
                X2 = Beta_pos[j] - A2 * D_beta

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

                A3, C3 = 2 * a * r1 - a, 2 * r2
                D_delta = abs(C3 * Delta_pos[j] - positions[i,j])
                X3 = Delta_pos[j] - A3 * D_delta

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

    return Alpha_pos, Alpha_score
```

Date
Page

```
def sphere_function(x):
    return np.sum(x**2)

dim = 5
search_agents = 30
max_iter = 50
lb, ub = -10, 10

best_position, best_score = gwo(sphere_function,
    dim, search_agents, max_iter, lb, ub)

print("Best position:", best_position)
print("Best score:", best_score)
```

Output

Best position : [-1.5212 -1.955 1.4038 -1.7118
1.7226]

Best score : 1.4007

**Code:**

```
import numpy as np

def gwo(obj_function, dim, search_agents, max_iter, lb, ub):
    # Initialize alpha, beta, and delta positions
    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")

    # Initialize the positions of search agents
    positions = np.random.uniform(lb, ub, (search_agents, dim))

    for iteration in range(max_iter):
        for i in range(search_agents):
            # Constrain positions within search space
            positions[i] = np.clip(positions[i], lb, ub)

            # Evaluate the fitness of each agent
            fitness = obj_function(positions[i])

            # Update Alpha, Beta, and Delta
            if fitness < Alpha_score:
                Alpha_score, Alpha_pos = fitness, positions[i].copy()
            elif fitness < Beta_score:
                Beta_score, Beta_pos = fitness, positions[i].copy()
            elif fitness < Delta_score:
                Delta_score, Delta_pos = fitness, positions[i].copy()

        # Print the current best score at each iteration
        print(f"Iteration {iteration + 1}/{max_iter}, Best Score: {Alpha_score:.6f}")

        # Update the position of each search agent
        a = 2 - iteration * (2 / max_iter)  # Linearly decreases from 2 to 0

        for i in range(search_agents):
            for j in range(dim):
                r1, r2 = np.random.rand(), np.random.rand()

                A1, C1 = 2 * a * r1 - a, 2 * r2
                D_alpha = abs(C1 * Alpha_pos[j] - positions[i, j])
                X1 = Alpha_pos[j] - A1 * D_alpha

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

```
            A2, C2 = 2 * a * r1 - a, 2 * r2
            D_beta = abs(C2 * Beta_pos[j] - positions[i, j])
            X2 = Beta_pos[j] - A2 * D_beta

            r1, r2 = np.random.rand(), np.random.rand()
            A3, C3 = 2 * a * r1 - a, 2 * r2
            D_delta = abs(C3 * Delta_pos[j] - positions[i, j])
            X3 = Delta_pos[j] - A3 * D_delta

            positions[i, j] = (X1 + X2 + X3) / 3  # Average of Alpha, Beta, Delta

    return Alpha_pos, Alpha_score

# Example: Optimization of the Sphere function
def sphere_function(x):
    return np.sum(x**2)

# Parameters
dim = 5                 # Dimensionality
search_agents = 30      # Number of wolves
max_iter = 50           # Maximum iterations
lb, ub = -10, 10        # Search space boundaries

best_position, best_score = gwo(sphere_function, dim, search_agents, max_iter, lb, ub)
print("Best Position:", best_position)
print("Best Score:", best_score)
```

**Output:**

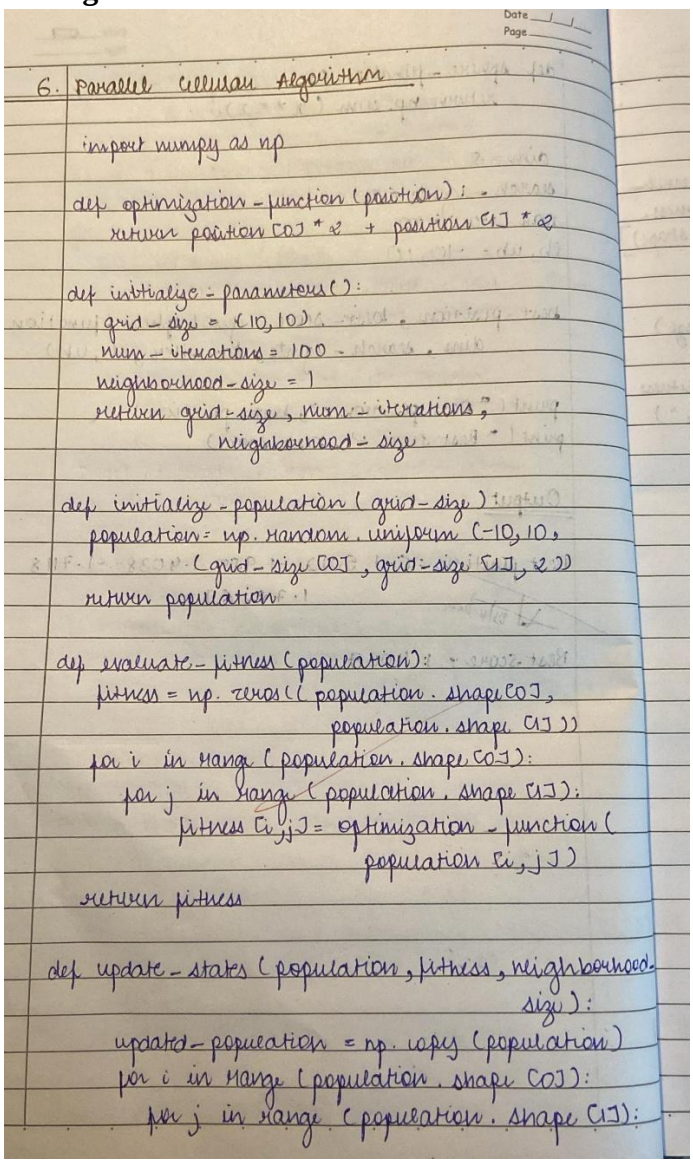
```
Wolf.py"
Iteration 1/50, Best Score: 52.846173
Iteration 2/50, Best Score: 32.167372
Iteration 3/50, Best Score: 12.209762
Iteration 4/50, Best Score: 7.146341
Iteration 5/50, Best Score: 1.346182
Iteration 6/50, Best Score: 0.768308
Iteration 7/50, Best Score: 0.598152
Iteration 8/50, Best Score: 0.190601
Iteration 9/50, Best Score: 0.046695
Iteration 10/50, Best Score: 0.028743
Iteration 11/50, Best Score: 0.017310
Iteration 12/50, Best Score: 0.005724
Iteration 13/50, Best Score: 0.002007
Iteration 14/50, Best Score: 0.000538
Iteration 15/50, Best Score: 0.000153
Iteration 16/50, Best Score: 0.000120
Iteration 17/50, Best Score: 0.000037
Iteration 18/50, Best Score: 0.000010
Iteration 19/50, Best Score: 0.000004
Iteration 20/50, Best Score: 0.000001
Iteration 21/50, Best Score: 0.000001
Iteration 22/50, Best Score: 0.000000
Iteration 23/50, Best Score: 0.000000
Iteration 24/50, Best Score: 0.000000
Iteration 25/50, Best Score: 0.000000
Iteration 26/50, Best Score: 0.000000
Iteration 27/50, Best Score: 0.000000
Iteration 28/50, Best Score: 0.000000
Iteration 29/50, Best Score: 0.000000
Iteration 30/50, Best Score: 0.000000
Iteration 31/50, Best Score: 0.000000
Iteration 32/50, Best Score: 0.000000
Iteration 33/50, Best Score: 0.000000
Iteration 34/50, Best Score: 0.000000
Iteration 35/50, Best Score: 0.000000
Iteration 36/50, Best Score: 0.000000
Iteration 37/50, Best Score: 0.000000
Iteration 38/50, Best Score: 0.000000
Iteration 39/50, Best Score: 0.000000
Iteration 40/50, Best Score: 0.000000
Iteration 41/50, Best Score: 0.000000
```

```
Iteration 42/50, Best Score: 0.000000
Iteration 43/50, Best Score: 0.000000
Iteration 44/50, Best Score: 0.000000
Iteration 45/50, Best Score: 0.000000
Iteration 46/50, Best Score: 0.000000
Iteration 47/50, Best Score: 0.000000
Iteration 48/50, Best Score: 0.000000
Iteration 49/50, Best Score: 0.000000
Iteration 50/50, Best Score: 0.000000
Best Position: [-3.72988510e-06 -5.99873502e-06  5.93390507e-06  4.25127629e-06
  5.78267284e-06]
Best Score: 1.3662074932044545e-10
```

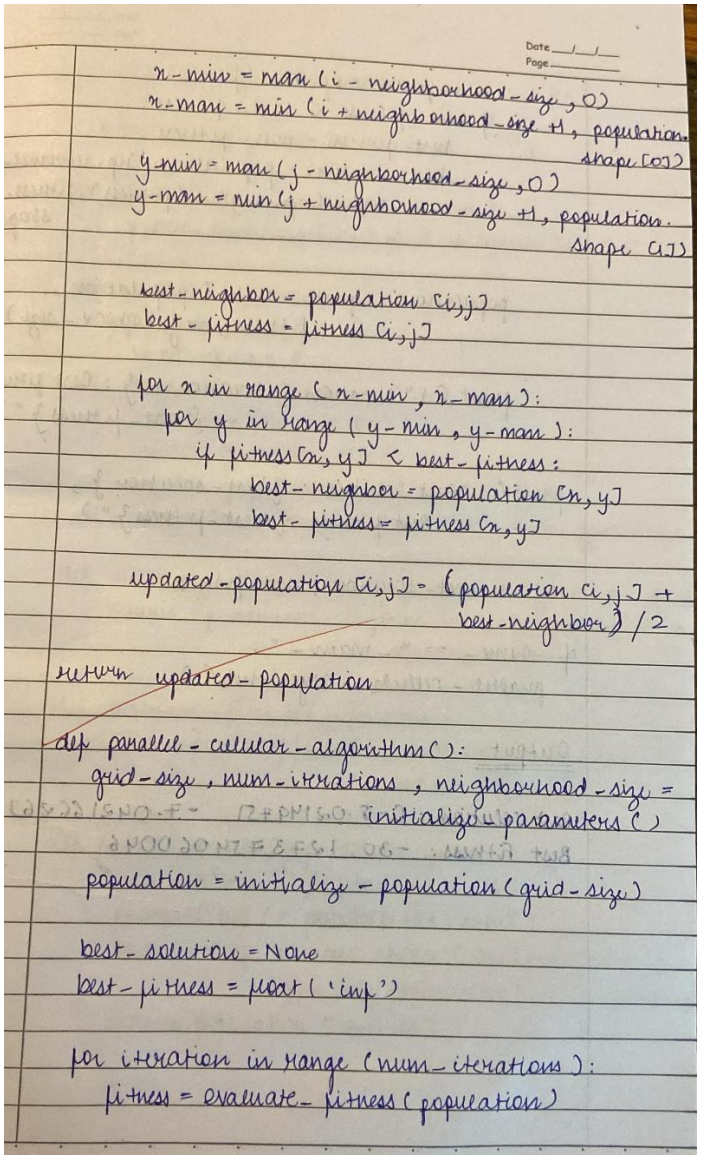
**Program 6:**

Parallel Cellular Algorithms are inspired by the functioning of biological cells that operate in a highly parallel and distributed manner. These algorithms leverage the principles of cellular automata and parallel computing to solve complex optimization problems efficiently. Each cell represents a potential solution and interacts with its neighbors to update its state based on predefined rules. This interaction models the diffusion of information across the cellular grid, enabling the algorithm to explore the search space effectively. Parallel Cellular Algorithms are particularly suitable for large-scale optimization problems and can be implemented on parallel computing architectures for enhanced performance.

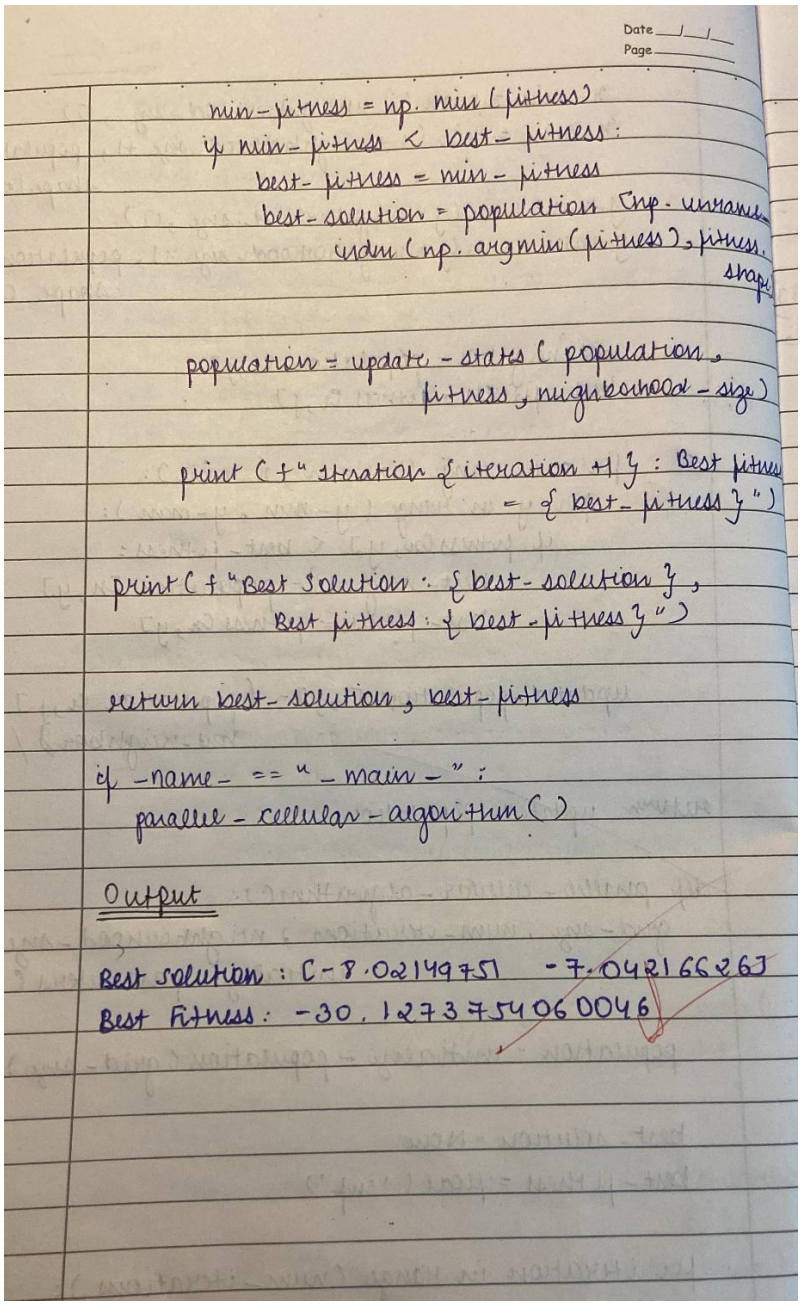
**Algorithm:**



```
6. Parallel Cellular Algorithm

import numpy as np

def optimization_function(position):
    return position[0]**2 + position[1]**2

def initialize_parameters():
    grid_size = (10,10)
    num_iterations = 100
    neighborhood_size = 1
    return grid_size, num_iterations,
           neighborhood_size

def initialize_population(grid_size):
    population = np.random.uniform(-10,10,
        (grid_size[0], grid_size[1], 2))
    return population

def evaluate_fitness(population):
    fitness = np.zeros((population.shape[0],
                        population.shape[1]))
    for i in range(population.shape[0]):
        for j in range(population.shape[1]):
            fitness[i,j] = optimization_function(
                population[i,j])
    return fitness

def update_states(population, fitness, neighborhood_
                  size):
    updated_population = np.copy(population)
    for i in range(population.shape[0]):
        for j in range(population.shape[1]):
```



```
            x_min = max(i - neighborhood_size, 0)
            x_max = min(i + neighborhood_size +1, population.
                                                  shape[0])
            y_min = max(j - neighborhood_size, 0)
            y_max = min(j + neighborhood_size +1, population.
                                                  shape[1])

            best_neighbor = population[i,j]
            best_fitness = fitness[i,j]

            for x in range(x_min, x_max):
                for y in range(y_min, y_max):
                    if fitness[x,y] < best_fitness:
                        best_neighbor = population[x,y]
                        best_fitness = fitness[x,y]

            updated_population[i,j] = (population[i,j] +
                                       best_neighbor) / 2
    return updated_population

def parallel_cellular_algorithm():
    grid_size, num_iterations, neighborhood_size =
        initialize_parameters()
    population = initialize_population(grid_size)

    best_solution = None
    best_fitness = float('inf')

    for iteration in range(num_iterations):
        fitness = evaluate_fitness(population)
```




**Code:**

```
import numpy as np

def optimization_function(position):
    return position[0]*2 + position[1]*2

def initialize_parameters():
    grid_size = (10, 10)
    num_iterations = 100
    neighborhood_size = 1
    return grid_size, num_iterations, neighborhood_size
```

```
def initialize_population(grid_size):
    population = np.random.uniform(-10, 10, (grid_size[0], grid_size[1], 2))
    return population

def evaluate_fitness(population):
    fitness = np.zeros((population.shape[0], population.shape[1]))
    for i in range(population.shape[0]):
        for j in range(population.shape[1]):
            fitness[i, j] = optimization_function(population[i, j])
    return fitness

def update_states(population, fitness, neighborhood_size):
    updated_population = np.copy(population)
    for i in range(population.shape[0]):
        for j in range(population.shape[1]):
            x_min = max(i - neighborhood_size, 0)
            x_max = min(i + neighborhood_size + 1, population.shape[0])
            y_min = max(j - neighborhood_size, 0)
            y_max = min(j + neighborhood_size + 1, population.shape[1])
            best_neighbor = population[i, j]
            best_fitness = fitness[i, j]
            for x in range(x_min, x_max):
                for y in range(y_min, y_max):
                    if fitness[x, y] < best_fitness:
                        best_neighbor = population[x, y]
                        best_fitness = fitness[x, y]
            updated_population[i, j] = (population[i, j] + best_neighbor) / 2
    return updated_population

def parallel_cellular_algorithm():
    grid_size, num_iterations, neighborhood_size = initialize_parameters()
    population = initialize_population(grid_size)
    best_solution = None
    best_fitness = float('inf')
    for iteration in range(num_iterations):
        fitness = evaluate_fitness(population)
        min_fitness = np.min(fitness)
        if min_fitness < best_fitness:
            best_fitness = min_fitness
            best_solution = population[np.unravel_index(np.argmin(fitness), fitness.shape)]
        population = update_states(population, fitness, neighborhood_size)
        print(f"Iteration {iteration + 1}: Best Fitness = {best_fitness}")
    print(f"Best Solution: {best_solution}, Best Fitness: {best_fitness}")
    return best_solution, best_fitness

if __name__ == "__main__":
    parallel_cellular_algorithm()
```

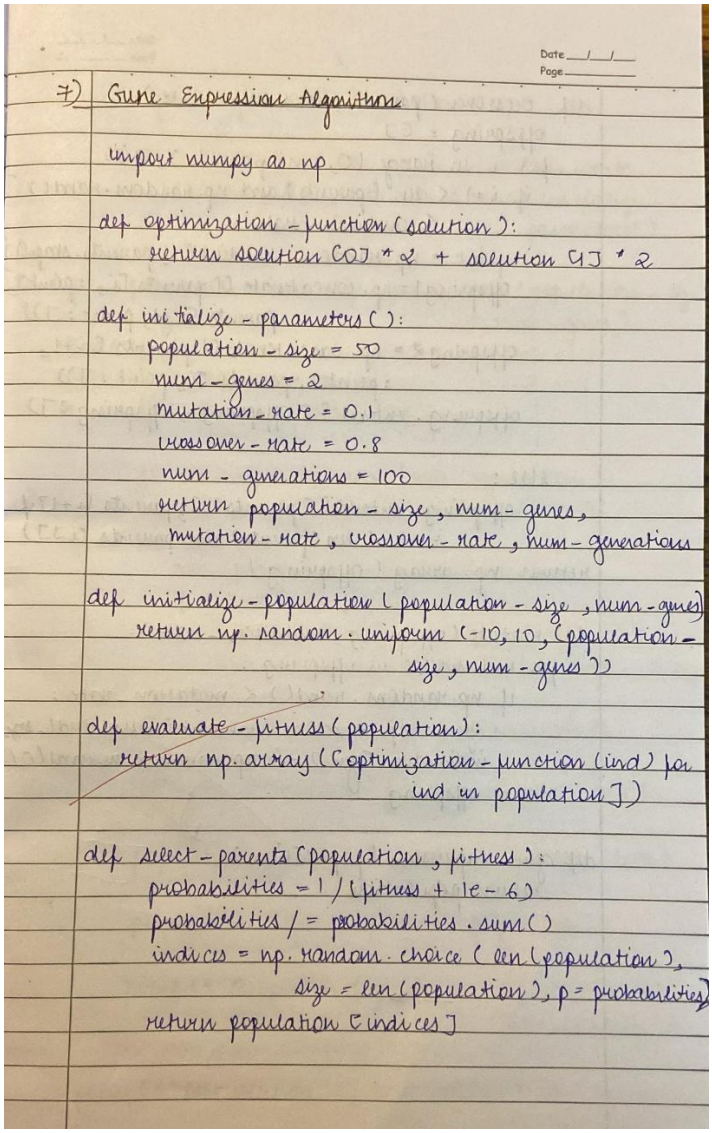
**Output:**

```
Iteration 1: Best Fitness = -33.73159832550755
Iteration 2: Best Fitness = -33.73159832550755
Iteration 3: Best Fitness = -33.73159832550755
Iteration 4: Best Fitness = -33.73159832550755
Iteration 5: Best Fitness = -33.73159832550755
Iteration 6: Best Fitness = -33.73159832550755
Iteration 7: Best Fitness = -33.73159832550755
Iteration 8: Best Fitness = -33.73159832550755
Iteration 9: Best Fitness = -33.73159832550755
Iteration 10: Best Fitness = -33.73159832550755
Iteration 11: Best Fitness = -33.73159832550755
Iteration 12: Best Fitness = -33.73159832550755
Iteration 13: Best Fitness = -33.73159832550755
Iteration 14: Best Fitness = -33.73159832550755
Iteration 15: Best Fitness = -33.73159832550755
Iteration 16: Best Fitness = -33.73159832550755
Iteration 17: Best Fitness = -33.73159832550755
Iteration 18: Best Fitness = -33.73159832550755
Iteration 19: Best Fitness = -33.73159832550755
Iteration 20: Best Fitness = -33.73159832550755
Iteration 21: Best Fitness = -33.73159832550755
Iteration 22: Best Fitness = -33.73159832550755
Iteration 23: Best Fitness = -33.73159832550755
Iteration 24: Best Fitness = -33.73159832550755
Iteration 25: Best Fitness = -33.73159832550755
Iteration 26: Best Fitness = -33.73159832550755
Iteration 27: Best Fitness = -33.73159832550755
Iteration 28: Best Fitness = -33.73159832550755
Iteration 29: Best Fitness = -33.73159832550755
Iteration 30: Best Fitness = -33.73159832550755
Iteration 31: Best Fitness = -33.73159832550755
Iteration 32: Best Fitness = -33.73159832550755
Iteration 33: Best Fitness = -33.73159832550755
Iteration 34: Best Fitness = -33.73159832550755
Iteration 35: Best Fitness = -33.73159832550755
Iteration 36: Best Fitness = -33.73159832550755
Iteration 37: Best Fitness = -33.73159832550755
Iteration 38: Best Fitness = -33.73159832550755
Iteration 39: Best Fitness = -33.73159832550755
Iteration 40: Best Fitness = -33.73159832550755
Iteration 41: Best Fitness = -33.73159832550755
Iteration 42: Best Fitness = -33.73159832550755
Iteration 43: Best Fitness = -33.73159832550755
Iteration 44: Best Fitness = -33.73159832550755
Iteration 45: Best Fitness = -33.73159832550755
Iteration 46: Best Fitness = -33.73159832550755
Iteration 47: Best Fitness = -33.73159832550755
Iteration 48: Best Fitness = -33.73159832550755
Iteration 49: Best Fitness = -33.73159832550755
Iteration 50: Best Fitness = -33.73159832550755
Iteration 51: Best Fitness = -33.73159832550755
Iteration 52: Best Fitness = -33.73159832550755
Iteration 53: Best Fitness = -33.73159832550755
Iteration 54: Best Fitness = -33.73159832550755
Iteration 55: Best Fitness = -33.73159832550755
Iteration 56: Best Fitness = -33.73159832550755
Iteration 57: Best Fitness = -33.73159832550755
Iteration 58: Best Fitness = -33.73159832550755
Iteration 59: Best Fitness = -33.73159832550755
Iteration 60: Best Fitness = -33.73159832550755
Iteration 61: Best Fitness = -33.73159832550755
Iteration 62: Best Fitness = -33.73159832550755
Iteration 63: Best Fitness = -33.73159832550755
Iteration 64: Best Fitness = -33.73159832550755
Iteration 65: Best Fitness = -33.73159832550755
Iteration 66: Best Fitness = -33.73159832550755
Iteration 67: Best Fitness = -33.73159832550755
Iteration 68: Best Fitness = -33.73159832550755
Iteration 69: Best Fitness = -33.73159832550755
Iteration 70: Best Fitness = -33.73159832550755
Iteration 71: Best Fitness = -33.73159832550755
Iteration 72: Best Fitness = -33.73159832550755
Iteration 73: Best Fitness = -33.73159832550755
Iteration 74: Best Fitness = -33.73159832550755
Iteration 75: Best Fitness = -33.73159832550755
Iteration 76: Best Fitness = -33.73159832550755
Iteration 77: Best Fitness = -33.73159832550755
Iteration 78: Best Fitness = -33.73159832550755
Iteration 79: Best Fitness = -33.73159832550755
Iteration 80: Best Fitness = -33.73159832550755
Iteration 81: Best Fitness = -33.73159832550755
Iteration 82: Best Fitness = -33.73159832550755
Iteration 83: Best Fitness = -33.73159832550755
Iteration 84: Best Fitness = -33.73159832550755
Iteration 85: Best Fitness = -33.73159832550755
Iteration 86: Best Fitness = -33.73159832550755
Iteration 87: Best Fitness = -33.73159832550755
Iteration 88: Best Fitness = -33.73159832550755
Iteration 89: Best Fitness = -33.73159832550755
Iteration 90: Best Fitness = -33.73159832550755
Iteration 91: Best Fitness = -33.73159832550755
Iteration 92: Best Fitness = -33.73159832550755
Iteration 93: Best Fitness = -33.73159832550755
Iteration 94: Best Fitness = -33.73159832550755
Iteration 95: Best Fitness = -33.73159832550755
Iteration 96: Best Fitness = -33.73159832550755
Iteration 97: Best Fitness = -33.73159832550755
Iteration 98: Best Fitness = -33.73159832550755
Iteration 99: Best Fitness = -33.73159832550755
Iteration 100: Best Fitness = -33.73159832550755
Best Solution: [-9.75031539 -7.11548377], Best Fitness: -33.73159832550755
```

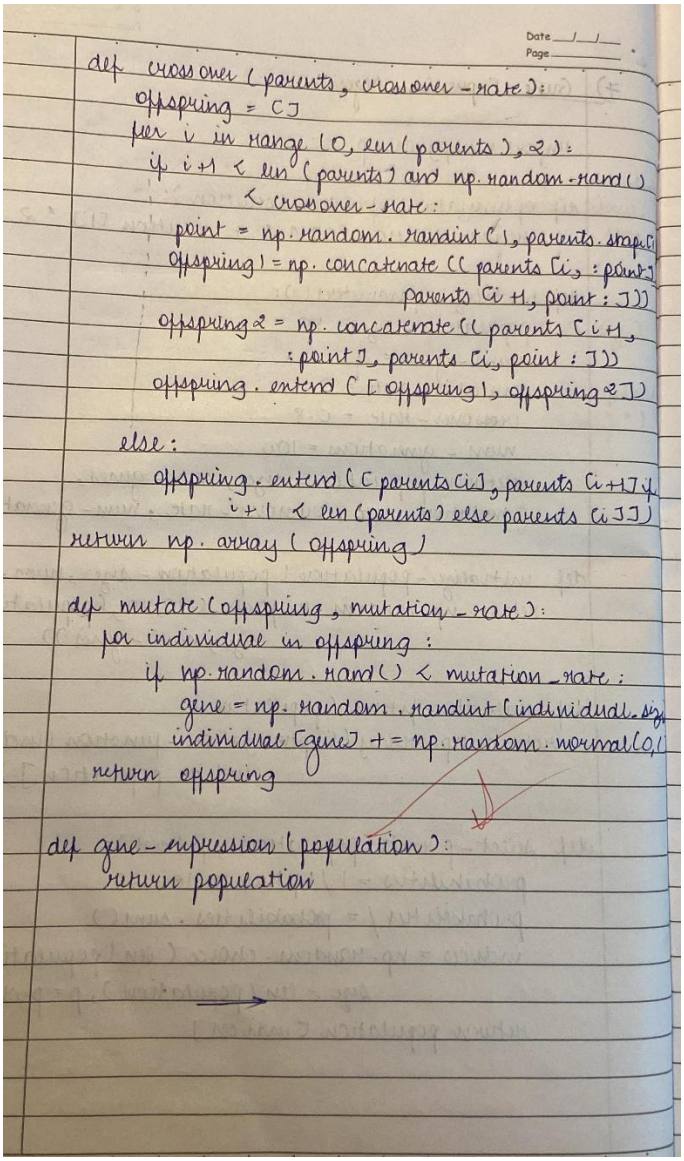
**Program 7:**

Gene Expression Algorithms (GEA) are inspired by the biological process of gene expression in living organisms. This process involves the translation of genetic information encoded in DNA into functional proteins. In GEA, solutions to optimization problems are encoded in a manner similar to genetic sequences. The algorithm evolves these solutions through selection, crossover, mutation, and gene expression to find optimal or near-optimal solutions. GEA is effective for solving complex optimization problems in various domains, including engineering, data analysis, and machine learning.

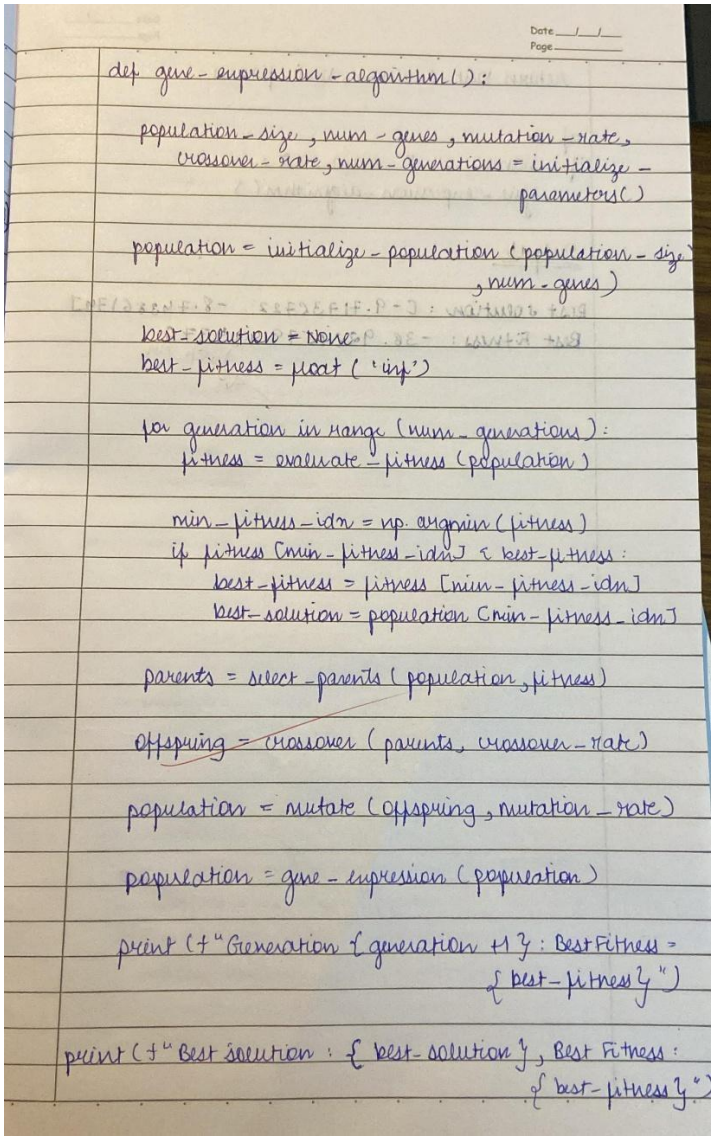
**Algorithm:**



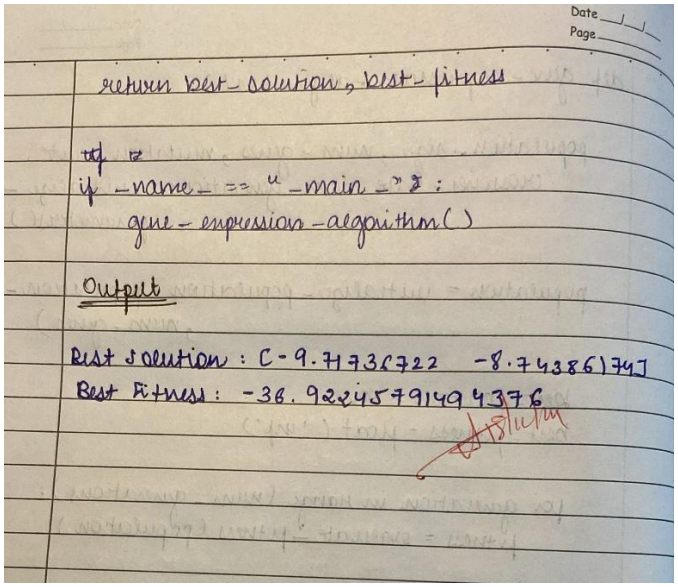
```
7) Gene Expression Algorithm

import numpy as np

def optimization_function(solution):
    return solution[0] ** 2 + solution[1] * 2

def initialize_parameters():
    population_size = 50
    num_genes = 2
    mutation_rate = 0.1
    crossover_rate = 0.8
    num_generations = 100
    return population_size, num_genes,
    mutation_rate, crossover_rate, num_generations

def initialize_population(population_size, num_genes):
    return np.random.uniform(-10, 10, (population_size,
                                       num_genes))

def evaluate_fitness(population):
    return np.array([optimization_function(ind) for
                     ind in population])

def select_parents(population, fitness):
    probabilities = 1/(fitness + 1e-6)
    probabilities /= probabilities.sum()
    indices = np.random.choice(len(population),
                               size=len(population), p=probabilities)
    return population[indices]
```



```
def crossover(parents, crossover_rate):
    offspring = []
    for i in range(0, len(parents), 2):
        if i+1 < len(parents) and np.random.rand()
                < crossover_rate:
            point = np.random.randint(1, parents.shape[1]
            offspring1 = np.concatenate((parents[i], parents[i+1]
                                         [:point], parents[i+1, point:]))
            offspring2 = np.concatenate((parents[i+1],
                                         [:point], parents[i, point:]))
            offspring.extend([offspring1, offspring2])

        else:
            offspring.extend([parents[i], parents[i+1] if
                              i+1 < len(parents) else parents[i]])
    return np.array(offspring)

def mutate(offspring, mutation_rate):
    for individual in offspring:
        if np.random.rand() < mutation_rate:
            gene = np.random.randint(individual.shape
            individual[gene] += np.random.normal(0,
    return offspring

def gene_expression(population):
    return population
```



```
def gene_expression_algorithm():

    population_size, num_genes, mutation_rate,
        crossover_rate, num_generations = initialize_
                                          parameters()

    population = initialize_population(population_size
                                       , num_genes)
    best_solution = None
    best_fitness = float('inf')

    for generation in range(num_generations):
        fitness = evaluate_fitness(population)

        min_fitness_idx = np.argmin(fitness)
        if fitness[min_fitness_idx] < best_fitness:
            best_fitness = fitness[min_fitness_idx]
            best_solution = population[min_fitness_idx]

        parents = select_parents(population, fitness)

        offspring = crossover(parents, crossover_rate)

        population = mutate(offspring, mutation_rate)

        population = gene_expression(population)

        print(f"Generation {generation+1}: Best Fitness =
                                        {best_fitness}")

    print(f"Best solution: {best_solution}, Best Fitness:
                                        {best_fitness}")
```



```
    return best_solution, best_fitness

if __name__ == "__main__":
    gene_expression_algorithm()
```

Output

```
Best solution: [-9.71736722  -8.74386174]
Best Fitness: -36.92245791494376
```

**Code:**

```
import numpy as np

# Define the optimization problem (e.g., f(x, y) = x^2 + y^2)
def optimization_function(solution):
    return solution[0]**2 + solution[1]**2  # Ensure non-negative fitness values

# Initialize parameters
def initialize_parameters():
    population_size = 50
    num_genes = 2  # Dimensionality of the solution
    mutation_rate = 0.1
    crossover_rate = 0.8
    num_generations = 100
    return population_size, num_genes, mutation_rate, crossover_rate, num_generations
```

```
# Initialize population
def initialize_population(population_size, num_genes):
    return np.random.uniform(-10, 10, (population_size, num_genes))

# Evaluate fitness
def evaluate_fitness(population):
    return np.array([optimization_function(ind) for ind in population])

# Selection
def select_parents(population, fitness):
    # Ensure fitness values are positive
    if (fitness < 0).any():
        fitness = fitness - fitness.min() + 1e-6  # Shift fitness values to positive

    probabilities = 1 / (fitness + 1e-6)  # Avoid division by zero
    probabilities /= probabilities.sum()
    indices = np.random.choice(len(population), size=len(population), p=probabilities)
    return population[indices]

# Crossover
def crossover(parents, crossover_rate):
    offspring = []
    for i in range(0, len(parents), 2):
        if i + 1 < len(parents) and np.random.rand() < crossover_rate:
            point = np.random.randint(1, parents.shape[1])
            offspring1 = np.concatenate((parents[i, :point], parents[i + 1, point:]))
            offspring2 = np.concatenate((parents[i + 1, :point], parents[i, point:]))
            offspring.extend([offspring1, offspring2])
        else:
            offspring.extend([parents[i], parents[i + 1] if i + 1 < len(parents) else parents[i]])
    return np.array(offspring)

# Mutation
def mutate(offspring, mutation_rate):
    for individual in offspring:
        if np.random.rand() < mutation_rate:
            gene = np.random.randint(individual.size)
            individual[gene] += np.random.normal(0, 1)  # Gaussian mutation
    return offspring

# Gene expression (identity mapping in this example)
def gene_expression(population):
    return population

# Gene Expression Algorithm
def gene_expression_algorithm():
```

```
    # Step 1: Initialize parameters
    population_size, num_genes, mutation_rate, crossover_rate, num_generations = initialize_parameters()

    # Step 2: Initialize population
    population = initialize_population(population_size, num_genes)

    best_solution = None
    best_fitness = float('inf')

    for generation in range(num_generations):
        # Step 3: Evaluate fitness
        fitness = evaluate_fitness(population)

        # Track the best solution
        min_fitness_idx = np.argmin(fitness)
        if fitness[min_fitness_idx] < best_fitness:
            best_fitness = fitness[min_fitness_idx]
            best_solution = population[min_fitness_idx]

        # Step 4: Selection
        parents = select_parents(population, fitness)

        # Step 5: Crossover
        offspring = crossover(parents, crossover_rate)

        # Step 6: Mutation
        population = mutate(offspring, mutation_rate)

        # Step 7: Gene Expression
        population = gene_expression(population)

        print(f"Generation {generation + 1}: Best Fitness = {best_fitness}")

    print(f"Best Solution: {best_solution}, Best Fitness: {best_fitness}")
    return best_solution, best_fitness

# Run the algorithm
if __name__ == "__main__":
    gene_expression_algorithm()
```

**Output:**

```
ExpressionAlgo.py"
Generation 1: Best Fitness = 7.335674378931877
Generation 2: Best Fitness = 1.6696060041558665
Generation 3: Best Fitness = 0.9765300327854515
Generation 4: Best Fitness = 0.7699785344806228
Generation 5: Best Fitness = 0.4571262488569531
Generation 6: Best Fitness = 0.2800979511228152
Generation 7: Best Fitness = 0.15463434474953341
Generation 8: Best Fitness = 0.15463434474953341
Generation 9: Best Fitness = 0.15463434474953341
Generation 10: Best Fitness = 0.11872056711255587
Generation 11: Best Fitness = 0.11872056711255587
Generation 12: Best Fitness = 0.11872056711255587
Generation 13: Best Fitness = 0.06394983810233248
Generation 14: Best Fitness = 0.0357736278570954
Generation 15: Best Fitness = 0.0357736278570954
Generation 16: Best Fitness = 0.009625052105338682
Generation 17: Best Fitness = 0.009625052105338682
Generation 18: Best Fitness = 0.009625052105338682
Generation 19: Best Fitness = 0.009625052105338682
Generation 20: Best Fitness = 0.009625052105338682
Generation 21: Best Fitness = 0.009625052105338682
Generation 22: Best Fitness = 0.009625052105338682
Generation 23: Best Fitness = 0.0059593600267104346
Generation 24: Best Fitness = 0.0059593600267104346
Generation 25: Best Fitness = 0.0007126018075639387
Generation 26: Best Fitness = 0.0007126018075639387
Generation 27: Best Fitness = 0.0007126018075639387
Generation 28: Best Fitness = 0.0007126018075639387
Generation 29: Best Fitness = 0.0007126018075639387
Generation 30: Best Fitness = 0.0007126018075639387
Generation 31: Best Fitness = 0.0007126018075639387
Generation 32: Best Fitness = 0.00019904270249505076
Generation 33: Best Fitness = 0.00019904270249505076
Generation 34: Best Fitness = 0.00019904270249505076
Generation 35: Best Fitness = 0.00019904270249505076
Generation 36: Best Fitness = 0.00019904270249505076
Generation 37: Best Fitness = 0.00019904270249505076
Generation 38: Best Fitness = 0.00019904270249505076
Generation 39: Best Fitness = 0.00019904270249505076
Generation 40: Best Fitness = 0.00019904270249505076
Generation 41: Best Fitness = 0.00019904270249505076
Generation 42: Best Fitness = 0.00019904270249505076
Generation 43: Best Fitness = 0.00019904270249505076
Generation 44: Best Fitness = 0.00019904270249505076
Generation 45: Best Fitness = 3.114273529653314e-06
Generation 46: Best Fitness = 3.114273529653314e-06
Generation 47: Best Fitness = 3.114273529653314e-06
Generation 48: Best Fitness = 3.114273529653314e-06
Generation 49: Best Fitness = 3.114273529653314e-06
Generation 50: Best Fitness = 3.114273529653314e-06
Generation 51: Best Fitness = 3.114273529653314e-06
Generation 52: Best Fitness = 3.114273529653314e-06
Generation 53: Best Fitness = 3.114273529653314e-06
Generation 54: Best Fitness = 3.114273529653314e-06
Generation 55: Best Fitness = 3.114273529653314e-06
Generation 56: Best Fitness = 3.114273529653314e-06
Generation 57: Best Fitness = 3.114273529653314e-06
Generation 58: Best Fitness = 3.114273529653314e-06
Generation 59: Best Fitness = 3.114273529653314e-06
Generation 60: Best Fitness = 3.114273529653314e-06
Generation 61: Best Fitness = 3.114273529653314e-06
Generation 62: Best Fitness = 3.114273529653314e-06
Generation 63: Best Fitness = 3.114273529653314e-06
Generation 64: Best Fitness = 3.114273529653314e-06
Generation 65: Best Fitness = 3.114273529653314e-06
Generation 66: Best Fitness = 3.114273529653314e-06
Generation 67: Best Fitness = 3.114273529653314e-06
Generation 68: Best Fitness = 3.114273529653314e-06
Generation 69: Best Fitness = 3.114273529653314e-06
Generation 70: Best Fitness = 3.114273529653314e-06
Generation 71: Best Fitness = 3.114273529653314e-06
Generation 72: Best Fitness = 3.114273529653314e-06
Generation 73: Best Fitness = 3.114273529653314e-06
Generation 74: Best Fitness = 3.114273529653314e-06
Generation 75: Best Fitness = 3.114273529653314e-06
Generation 76: Best Fitness = 3.114273529653314e-06
Generation 77: Best Fitness = 3.114273529653314e-06
Generation 78: Best Fitness = 3.114273529653314e-06
Generation 79: Best Fitness = 3.114273529653314e-06
Generation 80: Best Fitness = 3.114273529653314e-06
Generation 81: Best Fitness = 3.114273529653314e-06
Generation 82: Best Fitness = 3.114273529653314e-06
Generation 83: Best Fitness = 3.114273529653314e-06
Generation 84: Best Fitness = 3.114273529653314e-06
Generation 85: Best Fitness = 3.114273529653314e-06
Generation 86: Best Fitness = 3.114273529653314e-06
Generation 87: Best Fitness = 3.114273529653314e-06
Generation 88: Best Fitness = 3.114273529653314e-06
Generation 89: Best Fitness = 3.114273529653314e-06
Generation 90: Best Fitness = 3.114273529653314e-06
Generation 91: Best Fitness = 3.114273529653314e-06
Generation 92: Best Fitness = 3.114273529653314e-06
Generation 93: Best Fitness = 3.114273529653314e-06
Generation 94: Best Fitness = 3.114273529653314e-06
Generation 95: Best Fitness = 3.114273529653314e-06
Generation 96: Best Fitness = 3.114273529653314e-06
Generation 97: Best Fitness = 3.114273529653314e-06
Generation 98: Best Fitness = 3.114273529653314e-06
Generation 99: Best Fitness = 3.114273529653314e-06
Generation 100: Best Fitness = 3.114273529653314e-06
Best Solution: [-0.00063346 -0.00164712], Best Fitness: 3.114273529653314e-06
```