VISVESVARAYA TECHNOLOGICAL UNIVERSITY

"JnanaSangama", Belgaum -590014, Karnataka.

LAB RECORD

Bio Inspired Systems (23CS5BSBIS)

Submitted by

Sirigireddy Pranav Reddy (1BM22CS281)

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 **Sirigireddy Pranav Reddy (1BM22CS281),** who is a bonafide student of **B.M.S. College of Engineering.** It is in partial fulfillment for the award of **Bachelor of Engineering in Computer Science and Engineering** of the Visvesvaraya Technological University, Belgaum. The Lab report has been approved as it satisfies the academic requirements of the above mentioned subject and the work prescribed for the said degree.

Saritha A N Assistant Professor Department of CSE, BMSCE	Dr. Kavitha Sooda Professor & HOD Department of CSE, BMSCE

Index

Github Link:

https://github.com/pranavreddy-123/BIS_LAB.git

Program 1

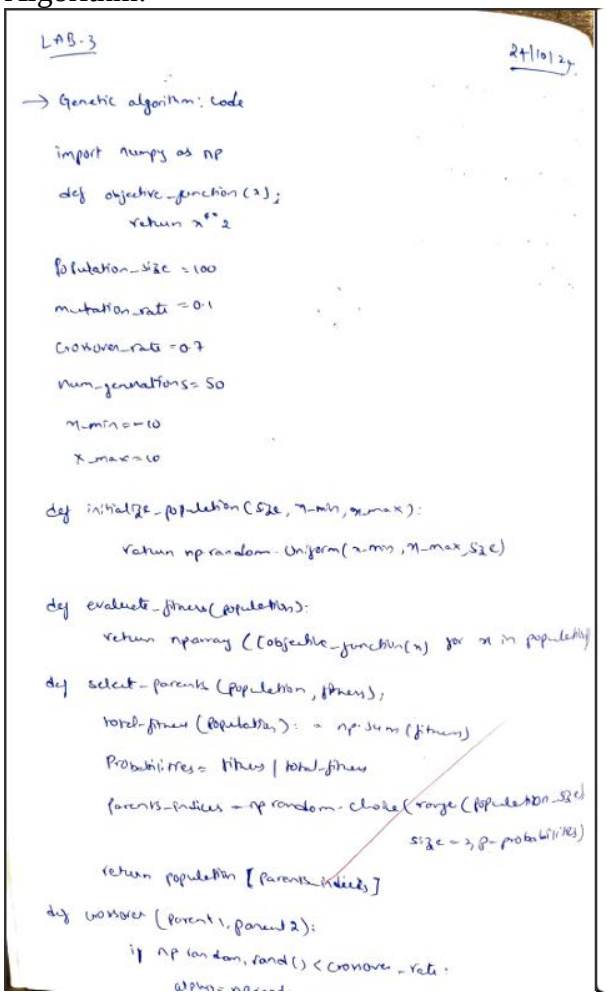
Genetic Algorithm for Optimization Problems:

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.

Implementation Steps:

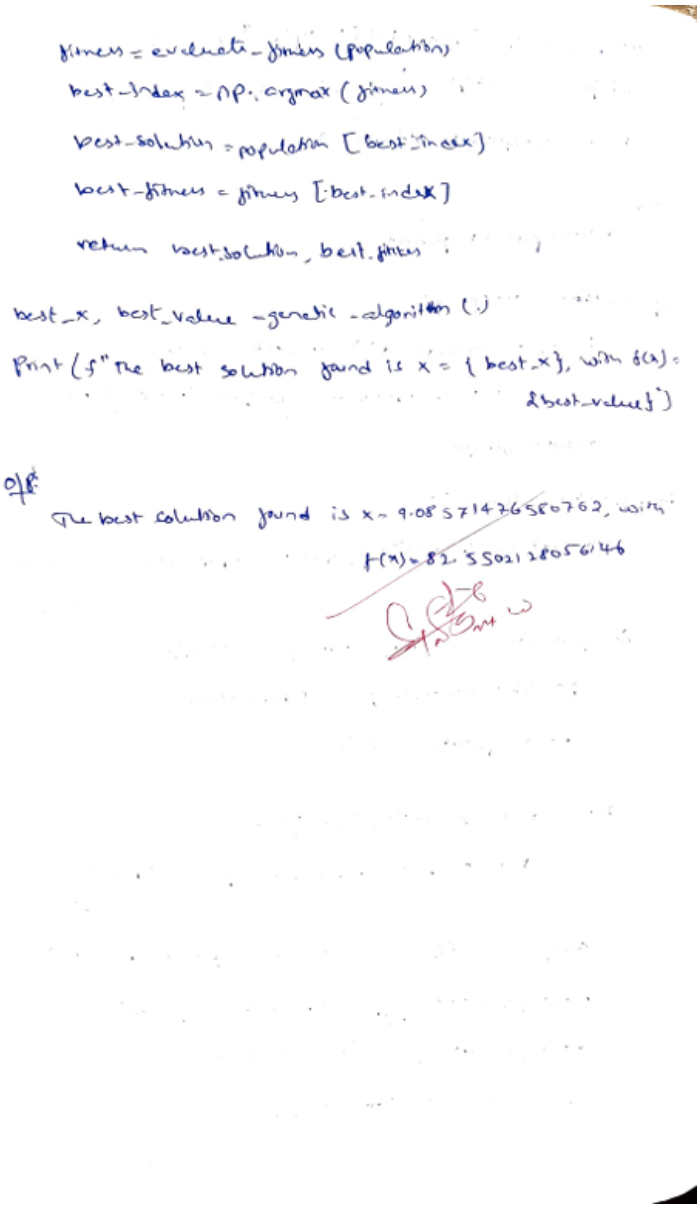
1. Define the Problem: Create a mathematical function to optimize.
2. Initialize Parameters: Set the population size, mutation rate, crossover rate, and number of generations.
3. Create Initial Population: Generate an initial population of potential solutions.
4. Evaluate Fitness: Evaluate the fitness of each individual in the population.
5. Selection: Select individuals based on their fitness to reproduce.
6. Crossover: Perform crossover between selected individuals to produce offspring.
7. Mutation: Apply mutation to the offspring to maintain genetic diversity.
8. Iteration: Repeat the evaluation, selection, crossover, and mutation processes for a fixed number of generations or until convergence criteria are met.
9. Output the Best Solution: Track and output the best solution found during the generations.

Algorithm:



```
LAB-3                                                    24/10/24

→ Genetic algorithm: code

import numpy as np
def objective_function(x):
    return x**2

population_size = 100
mutation_rate = 0.1
crossover_rate = 0.7
num_generations = 50
x_min = -10
x_max = 10

def initialize_population(size, x_min, x_max):
    return np.random.uniform(x_min, x_max, size)

def evaluate_fitness(population):
    return nparray([objective_function(x) for x in population])

def select_parents(population, fitness):
    total_fitness = np.sum(fitness)
    Probabilities = fitness / total_fitness
    parents_indices = np.random.choice(range(population_size),
                                       size=2, p=probabilities)
    return population[parents_indices]

def crossover(parent1, parent2):
    if np.random.rand() < crossover_rate:
```

```
        offspring1 = alpha * parent1 + (1-alpha) * parent2
        offspring2 = (1-alpha) * parent1 + alpha * parent2
        return offspring1, offspring2
    else:
        return parent1, parent2

def mutate(offspring):
    if np.random.random() < mutation_rate:
        return np.random.uniform(x_min, x_max)
    return offspring

def genetic_algorithm():
    population = initialize_population(population_size, x_min,
                                       x_max)
    for generation in range(num_generations):
        fitness = evaluate_fitness(population)
        new_population = []
        for _ in range(population_size // 2):
            parent1, parent2 = select_parents(population,
                                              fitness)
            offspring1, offspring2 = crossover(parent1, parent2)
            new_population.append(mutate(offspring1))
            new_population.append(mutate(offspring2))
        population = nparray(new_population)
```



Code:

```
import random

def fitness(x):
    return x**2

def initialize_population(pop_size, low, high):
    return [random.uniform(low, high) for _ in range(pop_size)]

def selection(population, fitness_values):
    total_fitness = sum(fitness_values)
    selection_probs = [f / total_fitness for f in fitness_values]
    return random.choices(population, weights=selection_probs, k=2)

def crossover(parent1, parent2):
    alpha = random.random()
```

```
    offspring1 = alpha * parent1 + (1 - alpha) * parent2
    offspring2 = alpha * parent2 + (1 - alpha) * parent1
    return offspring1, offspring2

def mutate(individual, mutation_rate, low, high):
    if random.random() < mutation_rate:
        return random.uniform(low, high)
    return individual

def genetic_algorithm(pop_size, generations, low, high, mutation_rate, crossover_rate):
    population = initialize_population(pop_size, low, high)
    best_solution = None
    best_fitness = float('-inf')

    for generation in range(generations):
        fitness_values = [fitness(ind) for ind in population]

        max_fitness = max(fitness_values)
        if max_fitness > best_fitness:
            best_fitness = max_fitness
            best_solution = population[fitness_values.index(max_fitness)]

        new_population = []
        while len(new_population) < pop_size:
            parent1, parent2 = selection(population, fitness_values)

            if random.random() < crossover_rate:
                offspring1, offspring2 = crossover(parent1, parent2)
            else:
                offspring1, offspring2 = parent1, parent2

            offspring1 = mutate(offspring1, mutation_rate, low, high)
            offspring2 = mutate(offspring2, mutation_rate, low, high)

            new_population.extend([offspring1, offspring2])

        population = new_population[:pop_size]

        print(f"Generation {generation+1}: Best fitness = {best_fitness:.4f}, Best solution =
{best_solution:.4f}")

    print(f"\nBest solution found: x = {best_solution:.4f}, f(x) = {best_fitness:.4f}")

population_size = 100
num_generations = 10
x_range_low = -10
x_range_high = 10
mutation_rate = 0.1
crossover_rate = 0.7
```

genetic_algorithm(population_size, num_generations, x_range_low, x_range_high, mutation_rate, crossover_rate)

Output:

```
Generation 1: Best fitness = 99.5858, Best solution = 9.9793
Generation 2: Best fitness = 99.5858, Best solution = 9.9793
Generation 3: Best fitness = 99.5858, Best solution = 9.9793
Generation 4: Best fitness = 99.5858, Best solution = 9.9793
Generation 5: Best fitness = 99.5858, Best solution = 9.9793
Generation 6: Best fitness = 99.5858, Best solution = 9.9793
Generation 7: Best fitness = 99.5858, Best solution = 9.9793
Generation 8: Best fitness = 99.5858, Best solution = 9.9793
Generation 9: Best fitness = 99.5858, Best solution = 9.9793
Generation 10: Best fitness = 99.5858, Best solution = 9.9793

Best solution found: x = 9.9793, f(x) = 99.5858
```

Program 2

Particle Swarm Optimization for Function Optimization:

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.

Implementation Steps:

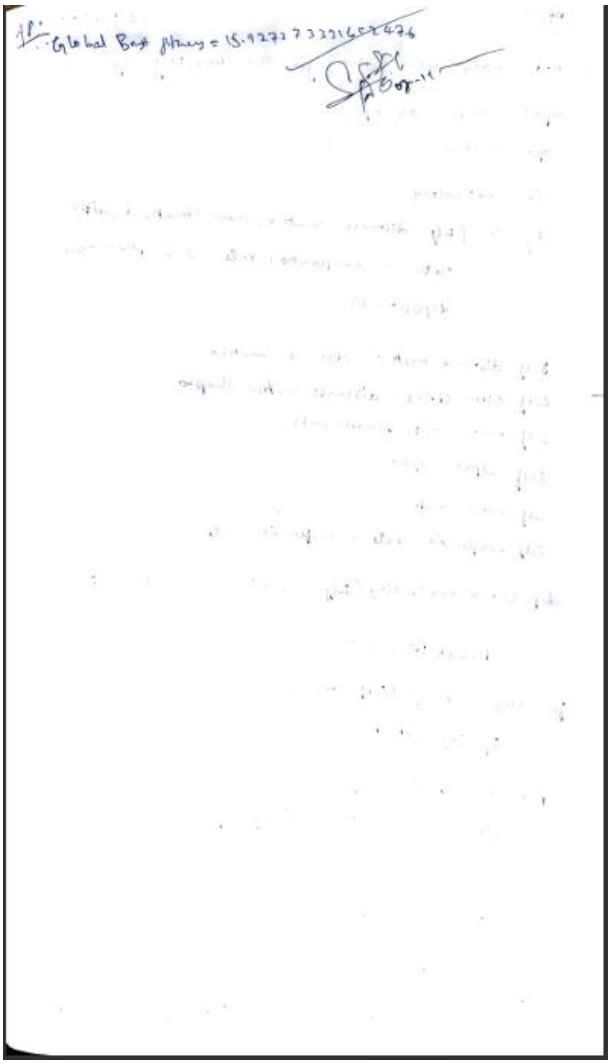
1. Define the Problem: Create a mathematical function to optimize.
2. Initialize Parameters: Set the number of particles, inertia weight, cognitive and social coefficients.
3. Initialize Particles: Generate an initial population of particles with random positions and velocities.
4. Evaluate Fitness: Evaluate the fitness of each particle based on the optimization function.
5. Update Velocities and Positions: Update the velocity and position of each particle based on its own best position and the global best position.
6. Iterate: Repeat the evaluation, updating, and position adjustment for a fixed number of iterations or until convergence criteria are met.
7. Output the Best Solution: Track and output the best solution found during the iterations.

Algorithm:

LAB-4 7/11/24

→ Particle Swarm optimization for function optimization: Code

```
import numpy as np
def rastrigin(x):
    A = 10
    return A*len(x) + sum(x**2 - A*np.cos(2*np.pi*x))

class particle:
    def _init_(self, dim, bounds):
        self.position = np.random.uniform(bounds[0], bound[1], dim)
        self.velocity = np.random.uniform(-1, 1, dim)
        self.best_position = np.copy(self.position)
        self.best_fitness = float('inf')

    def update_velocity(self, global_best_position, inertia_weight,
                        cognitive_weight, social_weight):
        r1, r2 = np.random.random(2)
        cognitive_velocity = cognitive_weight * r1 * (self.best_position)
        social_velocity = inertia_weight * self.velocity + cognitive_velocity)

    def update_position(self, bounds)
        self.position += self.velocity
        self.position = np.clip(self.position, bounds[0], bounds[1])

    def evaluate_fitness(self, fitness_function):
        fitness = fitness_function(self, position)
        if fitness < self.best_fitness:
            self.best_fitness = fitness
            self.best_position = np.copy(self.position)
        return fitness

def pso(num_particles, dim, bounds, fitness_function, num_iterations):
    particles = [particle(dim, bounds) for _ in range(num_particles)]
    inertia_weight = 0.7
    cognitive_weight = 1.5
    social_weight = 1.5
    global_best_position = None
    global_best_fitness = float('inf')
    for iteration in range(num_iterations):
        for particle in particles:
            fitness = particle.evaluate_fitness(fitness_function)
            if fitness < global_best_fitness:
                global < global_best_fitness:
                    global_best_fitness = fitness
                    global_best_position = np.copy(particle.position)
        for particle in particles:
            particle.update_velocity(global_best_position)
[illegible]
    [illegible]
    [illegible]
    best_position, best_fitness = pso
print([illegible])
print(f"Best [illegible]")
```



Code:

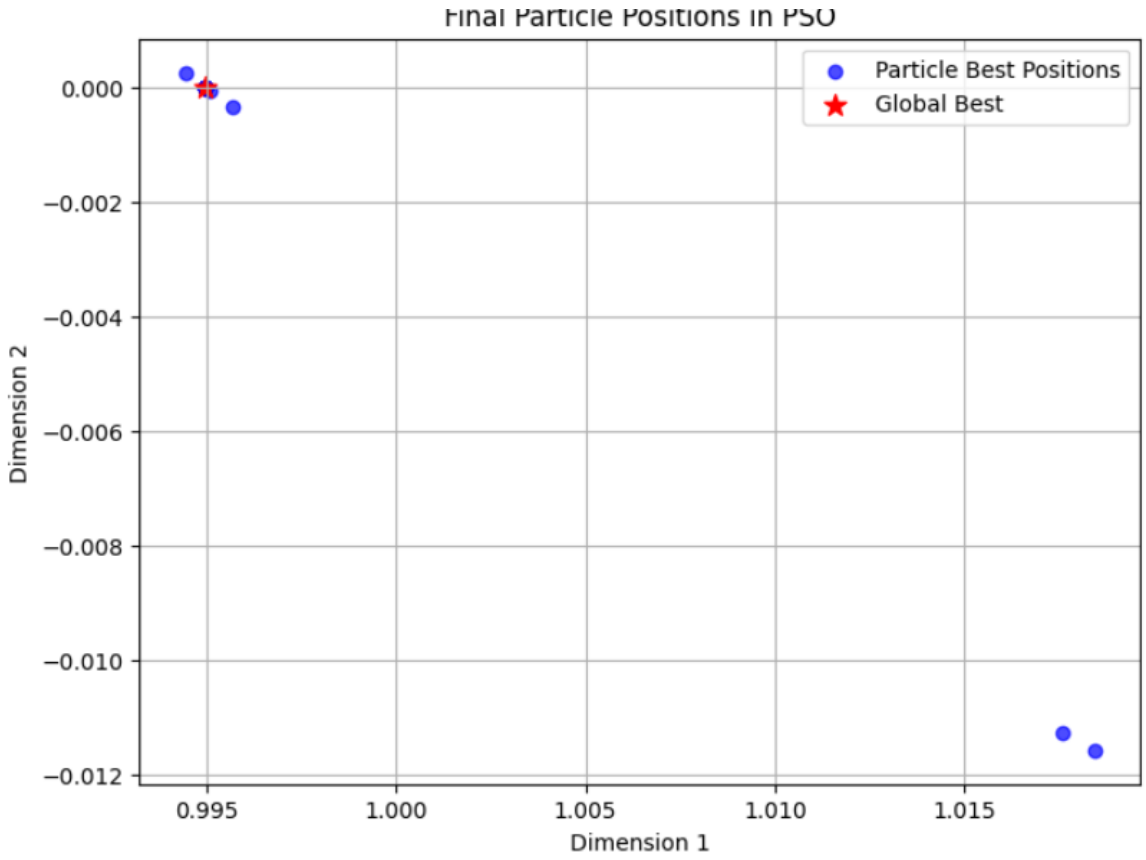
```
import numpy as np
import random
import matplotlib.pyplot as plt

def rastrigin(x):
    return 10 * len(x) + sum([(xi ** 2 - 10 * np.cos(2 * np.pi * xi)) for xi in x])

class Particle:
    def __init__(self, dim, bounds):
        self.position = np.random.uniform(bounds[0], bounds[1], dim)
        self.velocity = np.random.uniform(-1, 1, dim)
        self.best_position = np.copy(self.position)
        self.best_value = rastrigin(self.position)

    def evaluate(self):
        current_value = rastrigin(self.position)
        if current_value < self.best_value:
            self.best_value = current_value
```

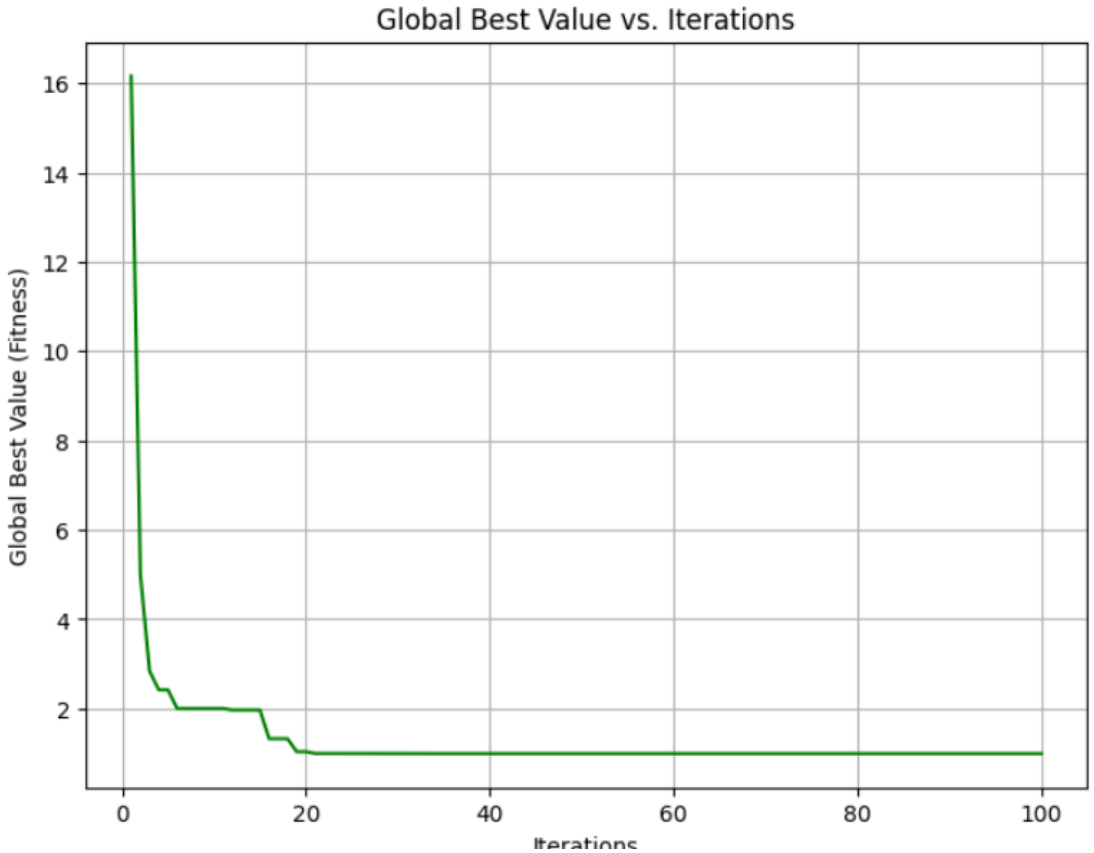
```
            self.best_position = np.copy(self.position)

def pso(dim, bounds, num_particles=30, max_iter=100, w=0.5, c1=1.5, c2=1.5):
    particles = [Particle(dim, bounds) for _ in range(num_particles)]

    global_best_position = None
    global_best_value = float('inf')

    best_values_over_iterations = []

    for iter in range(max_iter):
        for particle in particles:
            particle.evaluate()
            if particle.best_value < global_best_value:
                global_best_value = particle.best_value
                global_best_position = np.copy(particle.best_position)

        best_values_over_iterations.append(global_best_value)

        for particle in particles:
            inertia = w * particle.velocity
            cognitive = c1 * np.random.random() * (particle.best_position - particle.position)
            social = c2 * np.random.random() * (global_best_position - particle.position)

            particle.velocity = inertia + cognitive + social
            particle.position = particle.position + particle.velocity

            particle.position = np.clip(particle.position, bounds[0], bounds[1])

        if (iter+1) % 10 == 0:
            print(f"Iteration {iter+1}/{max_iter}, Global Best Value: {global_best_value}")

    return global_best_position, global_best_value, particles, best_values_over_iterations

if __name__ == "__main__":
    dim = 2
    bounds = [-5.12, 5.12]

    best_position, best_value, particles, best_values_over_iterations = pso(dim, bounds,
num_particles=30, max_iter=100)

    print("\nFinal Best Position:", best_position)
    print("Final Best Value:", best_value)

    fig, ax = plt.subplots(figsize=(8, 6))

    final_best_positions = np.array([particle.best_position for particle in particles])

    ax.scatter(final_best_positions[:, 0], final_best_positions[:, 1], color='blue', label="Particle Best
```

```
Positions", alpha=0.7)

    ax.scatter(best_position[0], best_position[1], color='red', label="Global Best", s=100, marker='*')

    ax.set_title("Final Particle Positions in PSO")
    ax.set_xlabel("Dimension 1")
    ax.set_ylabel("Dimension 2")
    ax.legend()
    plt.grid(True)
    plt.show()

    plt.figure(figsize=(8, 6))
    plt.plot(range(1, len(best_values_over_iterations) + 1), best_values_over_iterations, color='green')
    plt.title("Global Best Value vs. Iterations")
    plt.xlabel("Iterations")
    plt.ylabel("Global Best Value (Fitness)")
    plt.grid(True)
    plt.show()
```

Output:

```
Iteration 10/100, Global Best Value: 2.003250667292207
Iteration 20/100, Global Best Value: 1.0371970536607833
Iteration 30/100, Global Best Value: 0.9965161455248861
Iteration 40/100, Global Best Value: 0.9949848667711656
Iteration 50/100, Global Best Value: 0.9949610887864182
Iteration 60/100, Global Best Value: 0.994959059938175
Iteration 70/100, Global Best Value: 0.9949590571109823
Iteration 80/100, Global Best Value: 0.9949590570934674
Iteration 90/100, Global Best Value: 0.9949590570932898
Iteration 100/100, Global Best Value: 0.9949590570932898

Final Best Position: [ 9.94958638e-01 -8.70961770e-10]
Final Best Value: 0.9949590570932898
```


Final Particle Positions in PSO
Particle Best Positions
Global Best
0.000
−0.002
−0.004
−0.006
−0.008
−0.010
−0.012
Dimension 2
0.995
1.000
1.005
1.010
1.015
Dimension 1


Global Best Value vs. Iterations
16
14
12
10
8
6
4
2
Global Best Value (Fitness)
0
20
40
60
80
100
Iterations

Program 3

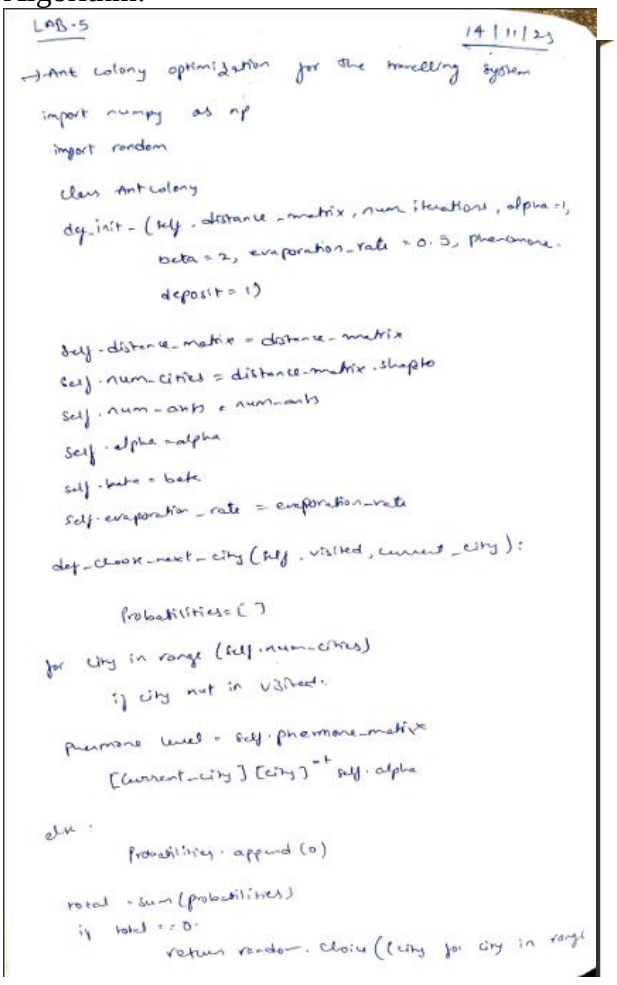
Ant Colony Optimization for the Traveling Salesman Problem:

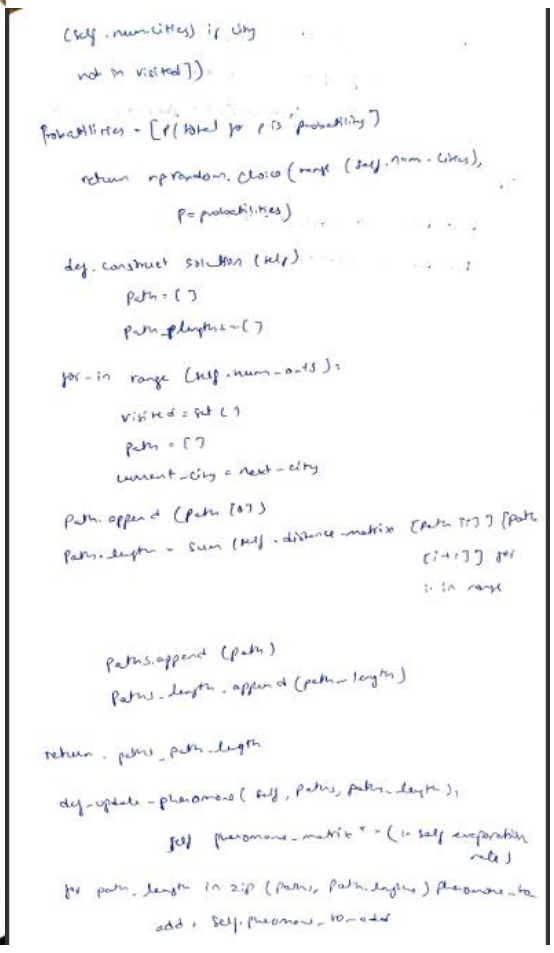
The foraging behavior of ants has inspired the development of optimization algorithms that can solve complex problems such as the Traveling Salesman Problem (TSP). Ant Colony Optimization (ACO) simulates the way ants find the shortest path between food sources and their nest. Implement the ACO algorithm 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.

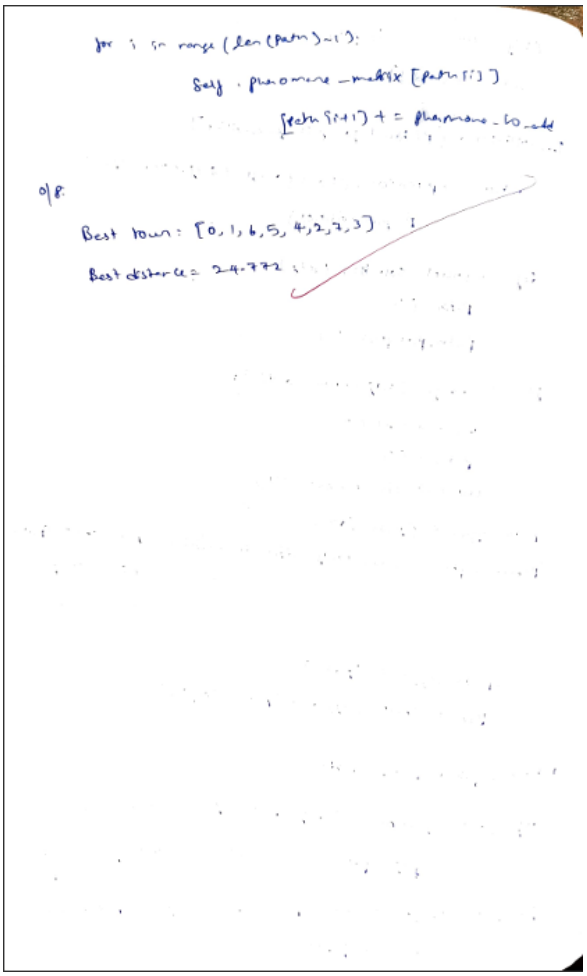
Implementation Steps:

1. Define the Problem: Create a set of cities with their coordinates.
2. Initialize Parameters: Set the number of ants, the importance of pheromone (alpha), the importance of heuristic information (beta), the evaporation rate (rho), and the initial pheromone value.
3. Construct Solutions: Each ant constructs a solution by probabilistically choosing the next city based on pheromone trails and heuristic information.
4. Update Pheromones: After all ants have constructed their solutions, update the pheromone trails based on the quality of the solutions found.
5. Iterate: Repeat the construction and updating process for a fixed number of iterations or until convergence criteria are met.
6. Output the Best Solution: Keep track of and output the best solution found during the iterations.

Algorithm:







Code:

```
import numpy as np
import matplotlib.pyplot as plt

# 1. Define the Problem: Create a set of cities with their coordinates
cities = np.array([
    [0, 0],  # City 0
    [1, 5],  # City 1
    [5, 1],  # City 2
    [6, 4],  # City 3
    [7, 8],  # City 4
])

# Calculate the distance matrix between each pair of cities
def calculate_distances(cities):
    num_cities = len(cities)
    distances = np.zeros((num_cities, num_cities))

    for i in range(num_cities):
        for j in range(num_cities):
            distances[i][j] = np.linalg.norm(cities[i] - cities[j])

    return distances
```

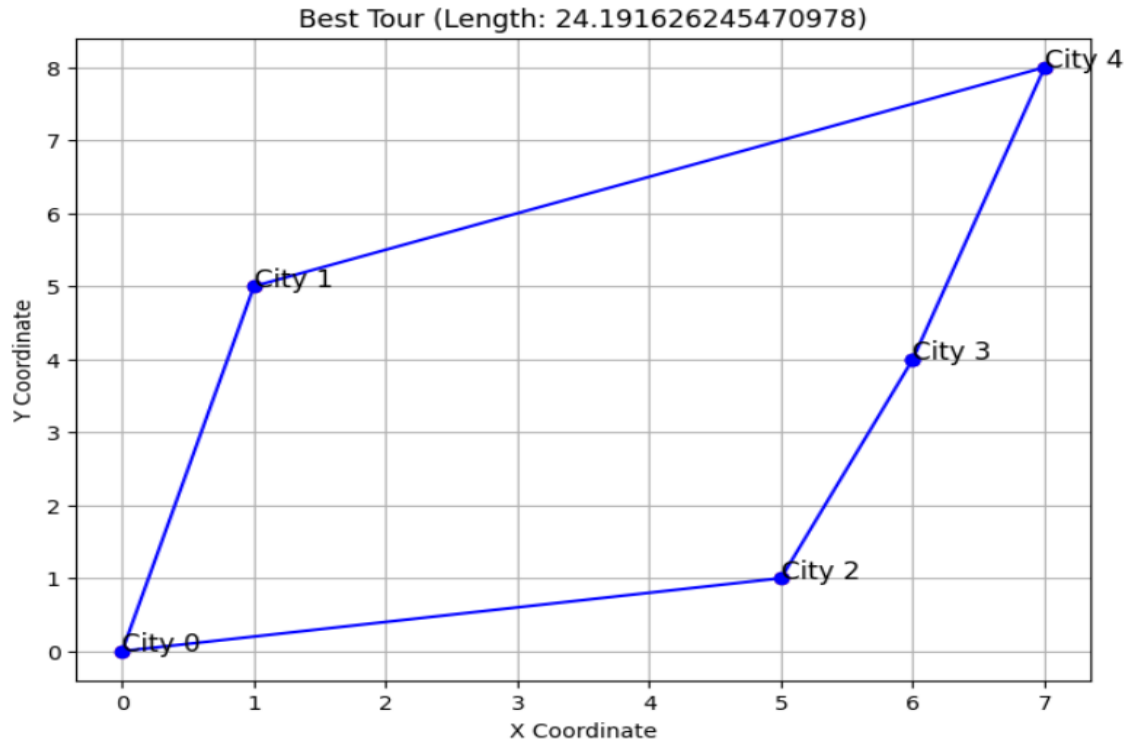
```
distances = calculate_distances(cities)

# 2. Initialize Parameters
num_ants = 10
num_cities = len(cities)
alpha = 1.0  # Influence of pheromone
beta = 5.0   # Influence of heuristic (inverse distance)
rho = 0.5    # Evaporation rate
num_iterations = 10
initial_pheromone = 1.0

# Pheromone matrix initialization
pheromone = np.ones((num_cities, num_cities)) * initial_pheromone

# 3. Heuristic information (Inverse of distance)
def heuristic(distances):
    with np.errstate(divide='ignore'):  # Ignore division by zero
        return 1 / distances

eta = heuristic(distances)

# 4. Choose next city probabilistically based on pheromone and heuristic info
def choose_next_city(pheromone, eta, visited):
    probs = []
    for j in range(num_cities):
        if j not in visited:
            pheromone_ij = pheromone[visited[-1], j] ** alpha
            heuristic_ij = eta[visited[-1], j] ** beta
            probs.append(pheromone_ij * heuristic_ij)
        else:
            probs.append(0)
    probs = np.array(probs)
    return np.random.choice(range(num_cities), p=probs / probs.sum())

# Construct solution for a single ant
def construct_solution(pheromone, eta):
    tour = [np.random.randint(0, num_cities)]
    while len(tour) < num_cities:
        next_city = choose_next_city(pheromone, eta, tour)
        tour.append(next_city)
    return tour

# 5. Update pheromones after all ants have constructed their tours
def update_pheromones(pheromone, all_tours, distances, best_tour):
    pheromone *= (1 - rho)  # Evaporate pheromones

    # Add pheromones for each ant's tour
    for tour in all_tours:
```

```
        tour_length = sum([distances[tour[i], tour[i + 1]] for i in range(-1, num_cities - 1)])
        for i in range(-1, num_cities - 1):
            pheromone[tour[i], tour[i + 1]] += 1.0 / tour_length

    # Increase pheromones on the best tour
    best_length = sum([distances[best_tour[i], best_tour[i + 1]] for i in range(-1, num_cities - 1)])
    for i in range(-1, num_cities - 1):
        pheromone[best_tour[i], best_tour[i + 1]] += 1.0 / best_length

# 6. Main ACO Loop: Iterate over multiple iterations to find the best solution
def run_aco(distances, num_iterations):
    pheromone = np.ones((num_cities, num_cities)) * initial_pheromone
    best_tour = None
    best_length = float('inf')

    for iteration in range(num_iterations):
        all_tours = [construct_solution(pheromone, eta) for _ in range(num_ants)]
        all_lengths = [sum([distances[tour[i], tour[i + 1]] for i in range(-1, num_cities - 1)]) for tour in
all_tours]

        current_best_length = min(all_lengths)
        current_best_tour = all_tours[all_lengths.index(current_best_length)]

        if current_best_length < best_length:
            best_length = current_best_length
            best_tour = current_best_tour

        update_pheromones(pheromone, all_tours, distances, best_tour)

        print(f"Iteration {iteration + 1}, Best Length: {best_length}")

    return best_tour, best_length

# Run the ACO algorithm
best_tour, best_length = run_aco(distances, num_iterations)

# 7. Output the Best Solution
print(f"Best Tour: {best_tour}")
print(f"Best Tour Length: {best_length}")

# 8. Plot the Best Route
def plot_route(cities, best_tour):
    plt.figure(figsize=(8, 6))
    for i in range(len(cities)):
        plt.scatter(cities[i][0], cities[i][1], color='red')
        plt.text(cities[i][0], cities[i][1], f"City {i}", fontsize=12)

    # Plot the tour as lines connecting the cities
    tour_cities = np.array([cities[i] for i in best_tour] + [cities[best_tour[0]]])  # Complete the loop by
```

returning to the start
```
    plt.plot(tour_cities[:, 0], tour_cities[:, 1], linestyle='-', marker='o', color='blue')

    plt.title(f"Best Tour (Length: {best_length})")
    plt.xlabel("X Coordinate")
    plt.ylabel("Y Coordinate")
    plt.grid(True)
    plt.show()

# Call the plot function
plot_route(cities, best_tour)
```

output:

```
Iteration 1, Best Length: 24.191626245470978
Iteration 2, Best Length: 24.191626245470978
Iteration 3, Best Length: 24.191626245470978
Iteration 4, Best Length: 24.191626245470978
Iteration 5, Best Length: 24.191626245470978
Iteration 6, Best Length: 24.191626245470978
Iteration 7, Best Length: 24.191626245470978
Iteration 8, Best Length: 24.191626245470978
Iteration 9, Best Length: 24.191626245470978
Iteration 10, Best Length: 24.191626245470978
Best Tour: [3, 4, 1, 0, 2]
Best Tour Length: 24.191626245470978
```



Program 4

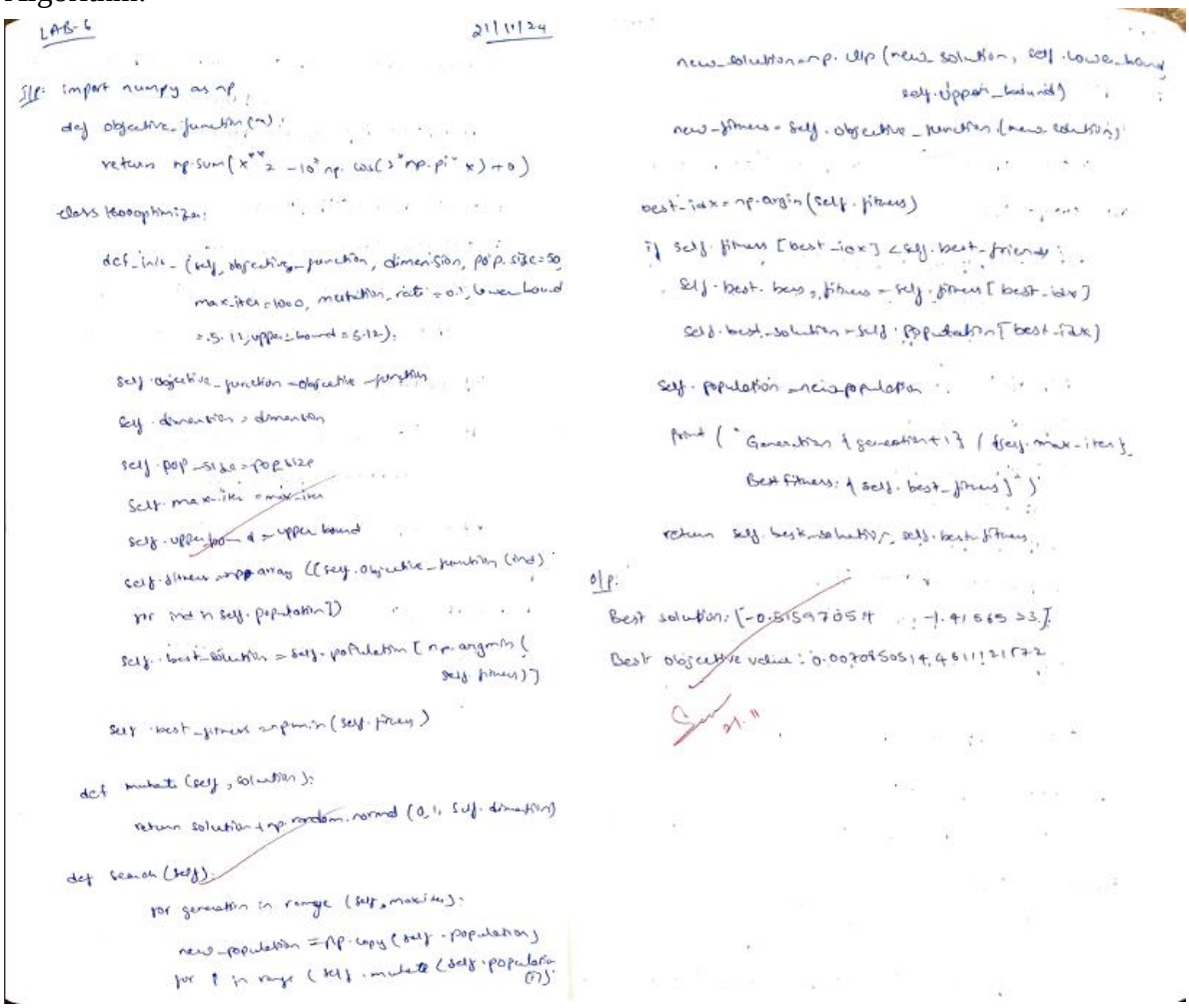
Cuckoo Search (CS):

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.

Implementation Steps:

1. Define the Problem: Create a mathematical function to optimize.
2. Initialize Parameters: Set the number of nests, the probability of discovery, and the number of iterations.
3. Initialize Population: Generate an initial population of nests with random positions.
4. Evaluate Fitness: Evaluate the fitness of each nest based on the optimization function.
5. Generate New Solutions: Create new solutions via Lévy flights.
6. Abandon Worst Nests: Abandon a fraction of the worst nests and replace them with new random positions.
7. Iterate: Repeat the evaluation, updating, and replacement process for a fixed number of iterations or until convergence criteria are met.
8. Output the Best Solution: Track and output the best solution found during the iterations.

Algorithm:



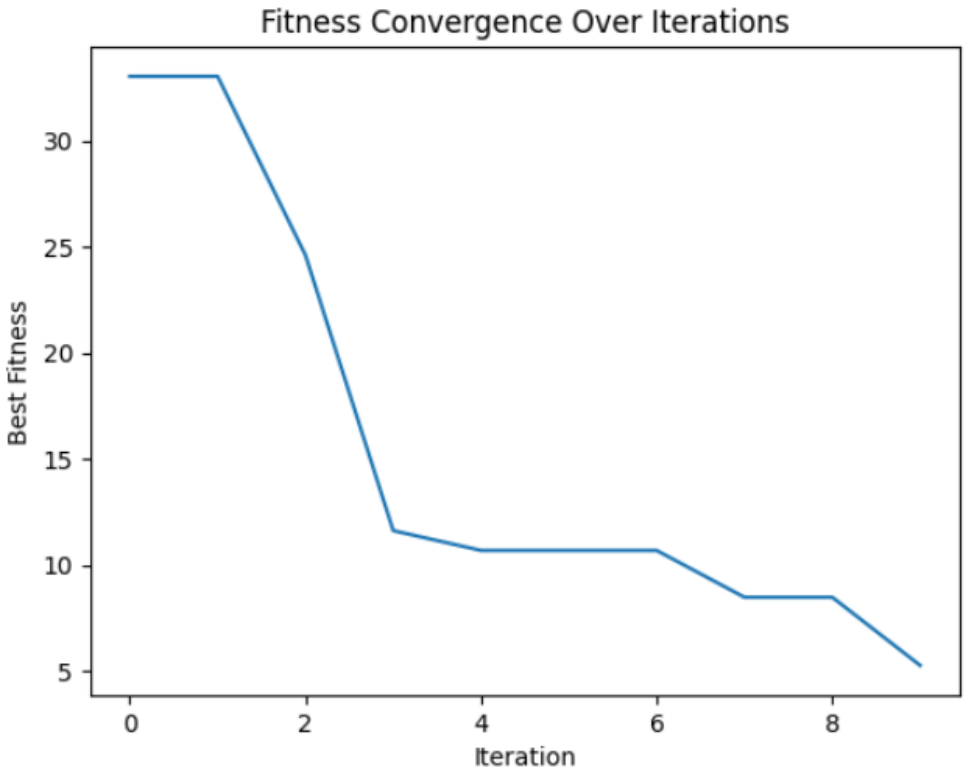
Code:

```
import numpy as np
import random
import math
import matplotlib.pyplot as plt

# Define a sample function to optimize (Sphere function in this case)
def objective_function(x):
    return np.sum(x ** 2)

# Lévy flight function
def levy_flight(Lambda):
    sigma_u = (math.gamma(1 + Lambda) * np.sin(np.pi * Lambda / 2) /
               (math.gamma((1 + Lambda) / 2) * Lambda * 2 ** ((Lambda - 1) / 2))) ** (1 / Lambda)
    sigma_v = 1
    u = np.random.normal(0, sigma_u, size=1)
    v = np.random.normal(0, sigma_v, size=1)
    step = u / (abs(v) ** (1 / Lambda))
    return step

# Cuckoo Search algorithm
def cuckoo_search(num_nests=25, num_iterations=100, discovery_rate=0.25, dim=5, lower_bound=-
10, upper_bound=10):
    # Initialize nests
    nests = np.random.uniform(lower_bound, upper_bound, (num_nests, dim))
    fitness = np.array([objective_function(nest) for nest in nests])

    # Get the current best nest
    best_nest_idx = np.argmin(fitness)
    best_nest = nests[best_nest_idx].copy()
    best_fitness = fitness[best_nest_idx]

    Lambda = 1.5  # Parameter for Lévy flights
    fitness_history = []  # To track fitness at each iteration

    for iteration in range(num_iterations):
        # Generate new solutions via Lévy flight
        for i in range(num_nests):
            step_size = levy_flight(Lambda)
            new_solution = nests[i] + step_size * (nests[i] - best_nest)
            new_solution = np.clip(new_solution, lower_bound, upper_bound)
            new_fitness = objective_function(new_solution)

            # Replace nest if new solution is better
            if new_fitness < fitness[i]:
                nests[i] = new_solution
                fitness[i] = new_fitness

        # Discover some nests with probability 'discovery_rate'
```

```
        random_nests = np.random.choice(num_nests, int(discovery_rate * num_nests), replace=False)
        for nest_idx in random_nests:
            nests[nest_idx] = np.random.uniform(lower_bound, upper_bound, dim)
            fitness[nest_idx] = objective_function(nests[nest_idx])

        # Update the best nest
        current_best_idx = np.argmin(fitness)
        if fitness[current_best_idx] < best_fitness:
            best_fitness = fitness[current_best_idx]
            best_nest = nests[current_best_idx].copy()

        # Store fitness for plotting
        fitness_history.append(best_fitness)

        # Print the best solution at each iteration (optional)
        print(f"Iteration {iteration+1}/{num_iterations}, Best Fitness: {best_fitness}")

    # Plot fitness convergence graph
    plt.plot(fitness_history)
    plt.title('Fitness Convergence Over Iterations')
    plt.xlabel('Iteration')
    plt.ylabel('Best Fitness')
    plt.show()

    # Return the best solution found
    return best_nest, best_fitness

# Example usage
best_nest, best_fitness = cuckoo_search(num_nests=30, num_iterations=10, dim=10, lower_bound=-
5, upper_bound=5)
print("Best Solution:", best_nest)
print("Best Fitness:", best_fitness)
```

Output:

```
Iteration 1/10, Best Fitness: 33.041281203083585
Iteration 2/10, Best Fitness: 33.041281203083585
Iteration 3/10, Best Fitness: 24.61474034339304
Iteration 4/10, Best Fitness: 11.62274110008269
Iteration 5/10, Best Fitness: 10.689701522637932
Iteration 6/10, Best Fitness: 10.689701522637932
Iteration 7/10, Best Fitness: 10.689701522637932
Iteration 8/10, Best Fitness: 8.483040606104721
Iteration 9/10, Best Fitness: 8.483040606104721
Iteration 10/10, Best Fitness: 5.27254818921324
```




```
Best Solution: [-0.44074699  0.44475909 -0.40497755 -1.1444419  -0.79762137  0.46740521
 -0.91064972 -1.00122337  0.38893795 -0.7543568 ]
Best Fitness: 5.27254818921324
```

Program 5

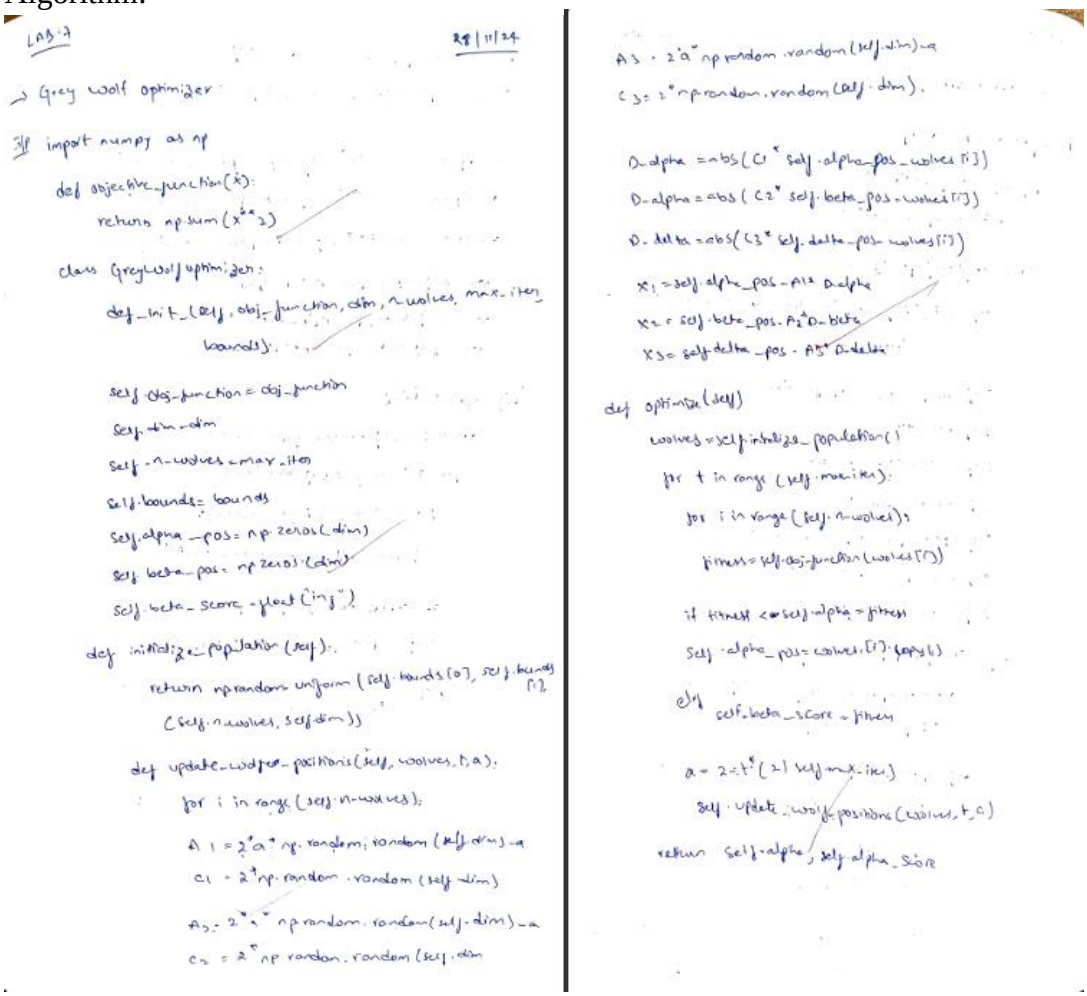
Grey Wolf Optimizer (GWO):

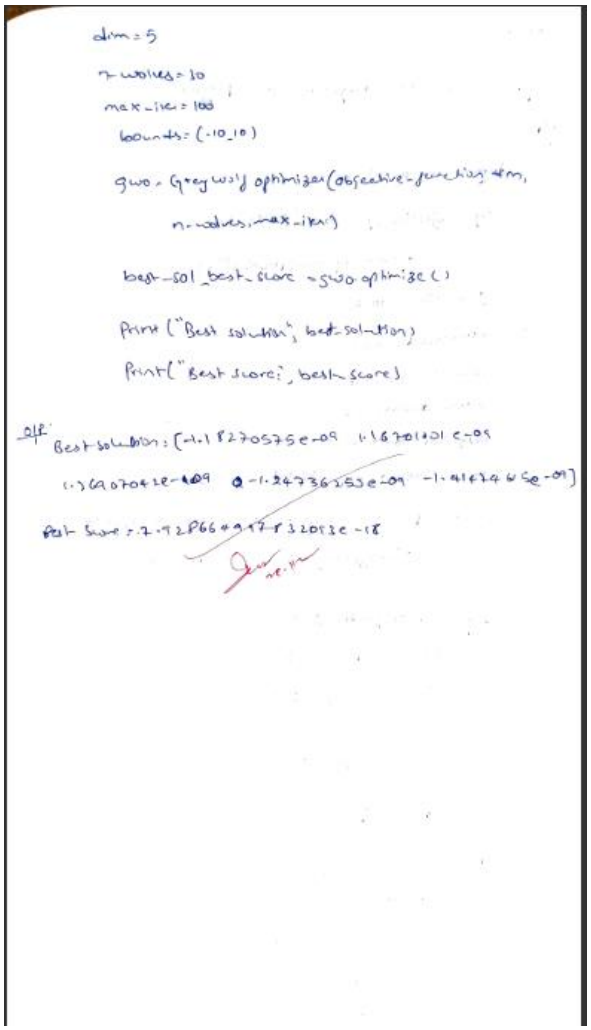
The Grey Wolf Optimizer (GWO) algorithm is a swarm intelligence algorithm inspired by the social hierarchy and hunting behavior of grey wolves. It mimics the leadership structure of alpha, beta, delta, and omega wolves and their collaborative hunting strategies. The GWO algorithm uses these social hierarchies 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.

Implementation Steps:

1. Define the Problem: Create a mathematical function to optimize.
2. Initialize Parameters: Set the number of wolves and the number of iterations.
3. Initialize Population: Generate an initial population of wolves with random positions.
4. Evaluate Fitness: Evaluate the fitness of each wolf based on the optimization function.
5. Update Positions: Update the positions of the wolves based on the positions of alpha, beta, and delta wolves.
6. Iterate: Repeat the evaluation and position updating process for a fixed number of iterations or until convergence criteria are met.
7. Output the Best Solution: Track and output the best solution found during the iterations.

Algorithm:





Code:

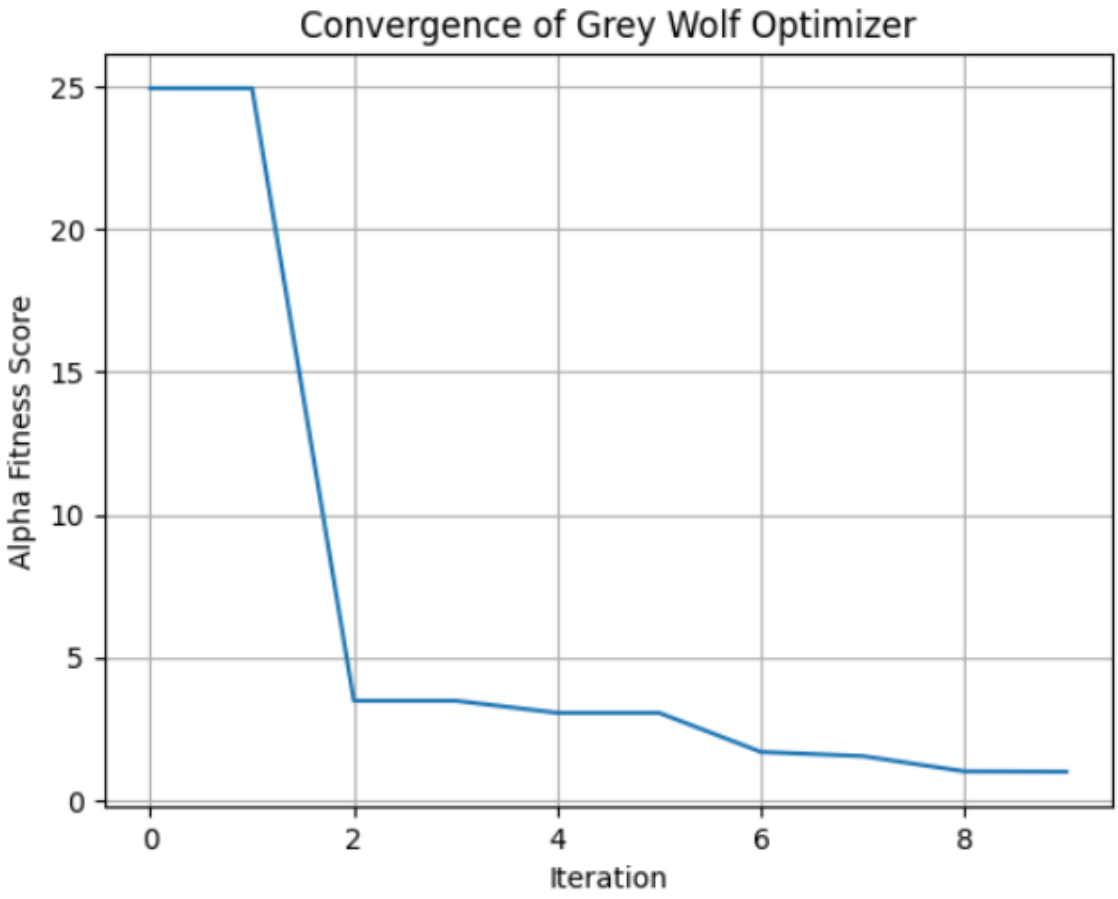
```
import numpy as np
import matplotlib.pyplot as plt

# Step 1: Define the Problem (a mathematical function to optimize)
def objective_function(x):
    return np.sum(x**2)  # Example: Sphere function (minimize sum of squares)

# Step 2: Initialize Parameters
num_wolves = 5  # Number of wolves in the pack
num_dimensions = 2  # Number of dimensions (for the optimization problem)
num_iterations = 10  # Number of iterations
lb = -10  # Lower bound of search space
ub = 10  # Upper bound of search space

# Step 3: Initialize Population (Generate initial positions randomly)
wolves = np.random.uniform(lb, ub, (num_wolves, num_dimensions))

# Initialize alpha, beta, delta wolves
alpha_pos = np.zeros(num_dimensions)
beta_pos = np.zeros(num_dimensions)
```

```
delta_pos = np.zeros(num_dimensions)

alpha_score = float('inf')  # Best (alpha) score
beta_score = float('inf')   # Second best (beta) score
delta_score = float('inf')  # Third best (delta) score

# To store the alpha score over iterations for graphing
alpha_score_history = []

# Step 4: Evaluate Fitness and assign Alpha, Beta, Delta wolves
def evaluate_fitness():
    global alpha_pos, beta_pos, delta_pos, alpha_score, beta_score, delta_score

    for wolf in wolves:
        fitness = objective_function(wolf)

        # Update Alpha, Beta, Delta wolves based on fitness
        if fitness < alpha_score:
            delta_score = beta_score
            delta_pos = beta_pos.copy()

            beta_score = alpha_score
            beta_pos = alpha_pos.copy()

            alpha_score = fitness
            alpha_pos = wolf.copy()
        elif fitness < beta_score:
            delta_score = beta_score
            delta_pos = beta_pos.copy()

            beta_score = fitness
            beta_pos = wolf.copy()
        elif fitness < delta_score:
            delta_score = fitness
            delta_pos = wolf.copy()

# Step 5: Update Positions
def update_positions(iteration):
    a = 2 - iteration * (2 / num_iterations)  # a decreases linearly from 2 to 0

    for i in range(num_wolves):
        for j in range(num_dimensions):
            r1 = np.random.random()
            r2 = np.random.random()

            # Position update based on alpha
            A1 = 2 * a * r1 - a
            C1 = 2 * r2
            D_alpha = abs(C1 * alpha_pos[j] - wolves[i, j])
```

```
            X1 = alpha_pos[j] - A1 * D_alpha

            # Position update based on beta
            r1 = np.random.random()
            r2 = np.random.random()
            A2 = 2 * a * r1 - a
            C2 = 2 * r2
            D_beta = abs(C2 * beta_pos[j] - wolves[i, j])
            X2 = beta_pos[j] - A2 * D_beta

            # Position update based on delta
            r1 = np.random.random()
            r2 = np.random.random()
            A3 = 2 * a * r1 - a
            C3 = 2 * r2
            D_delta = abs(C3 * delta_pos[j] - wolves[i, j])
            X3 = delta_pos[j] - A3 * D_delta

            # Update wolf position
            wolves[i, j] = (X1 + X2 + X3) / 3

            # Apply boundary constraints
            wolves[i, j] = np.clip(wolves[i, j], lb, ub)

# Step 6: Iterate (repeat evaluation and position updating)
for iteration in range(num_iterations):
    evaluate_fitness()  # Evaluate fitness of each wolf
    update_positions(iteration)  # Update positions based on alpha, beta, delta

    # Record the alpha score for this iteration
    alpha_score_history.append(alpha_score)

    # Optional: Print current best score
    print(f"Iteration {iteration+1}/{num_iterations}, Alpha Score: {alpha_score}")

# Step 7: Output the Best Solution
print("Best Solution:", alpha_pos)
print("Best Solution Fitness:", alpha_score)

# Plotting the convergence graph
plt.plot(alpha_score_history)
plt.title('Convergence of Grey Wolf Optimizer')
plt.xlabel('Iteration')
plt.ylabel('Alpha Fitness Score')
plt.grid(True)
plt.show()
```

Output:

```
Iteration 1/10, Alpha Score: 24.938603997415413
Iteration 2/10, Alpha Score: 24.938603997415413
Iteration 3/10, Alpha Score: 3.478306502607043
Iteration 4/10, Alpha Score: 3.478306502607043
Iteration 5/10, Alpha Score: 3.0526022091841627
Iteration 6/10, Alpha Score: 3.0526022091841627
Iteration 7/10, Alpha Score: 1.6838080429555806
Iteration 8/10, Alpha Score: 1.5380015669091764
Iteration 9/10, Alpha Score: 1.0036157784249133
Iteration 10/10, Alpha Score: 0.9922915488635977
Best Solution: [ 0.82264201 -0.56173987]
Best Solution Fitness: 0.9922915488635977
```



Program 6

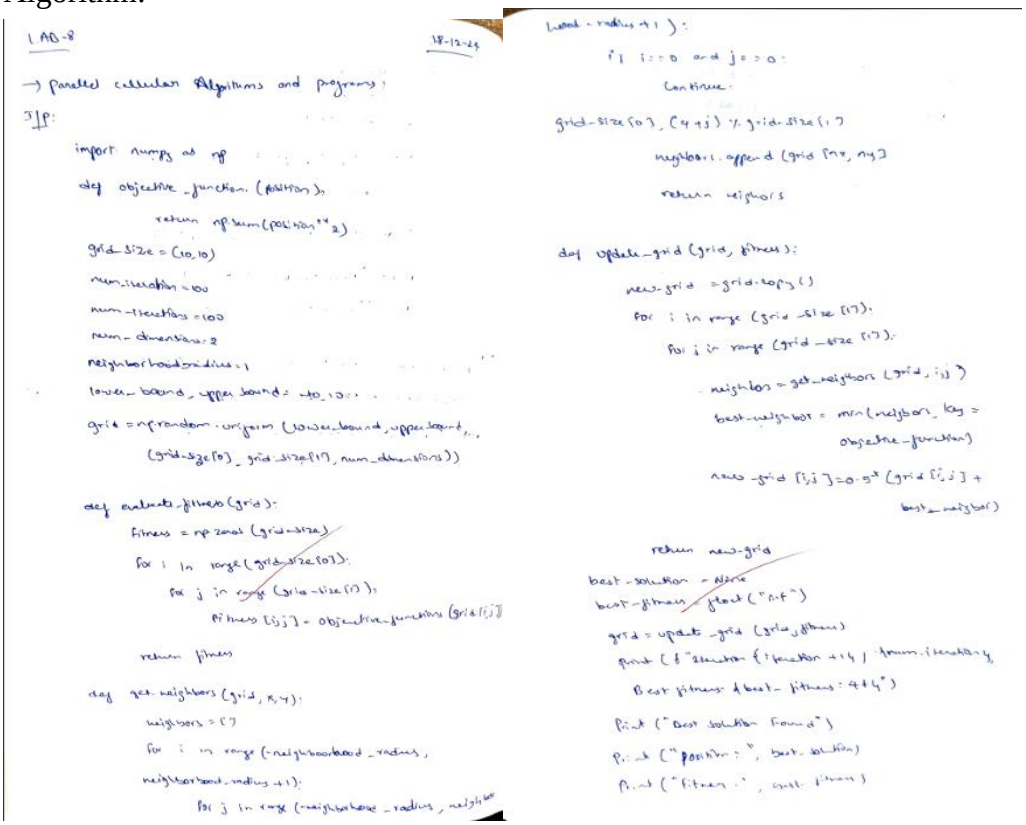
Parallel Cellular Algorithms and Programs:

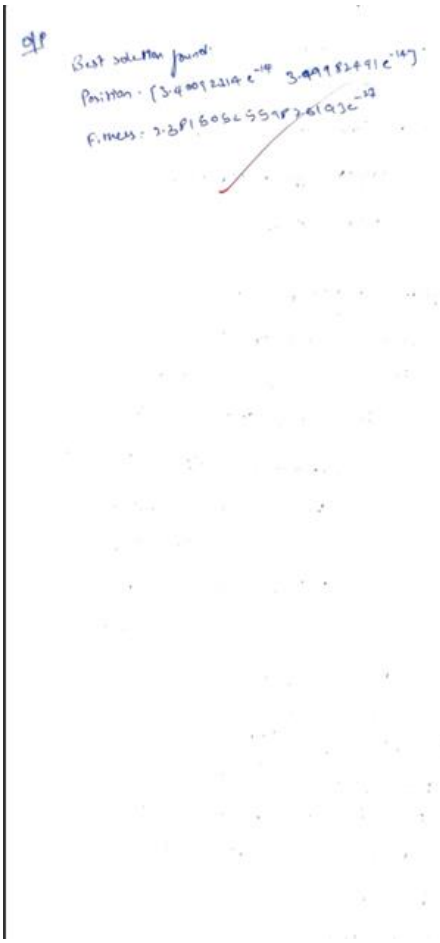
Parallel Cellular Algorithms are inspired by the functioning of biological cells that operate in a highly parallel and distributed manner. These algorithms leverage the principles of cellular automata and parallel computing to solve complex optimization problems efficiently. Each cell represents a potential 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.

Implementation Steps:

1. Define the Problem: Create a mathematical function to optimize.
2. Initialize Parameters: Set the number of cells, grid size, neighborhood structure, and number of iterations.
3. Initialize Population: Generate an initial population of cells with random positions in the solution space.
4. Evaluate Fitness: Evaluate the fitness of each cell based on the optimization function.
5. Update States: Update the state of each cell based on the states of its neighboring cells and predefined update rules.
6. Iterate: Repeat the evaluation and state updating process for a fixed number of iterations or until convergence criteria are met.
7. Output the Best Solution: Track and output the best solution found during the iterations.

Algorithm:





Code:

```
import numpy as np

def sphere_function(position):
    """
    Objective function to minimize.
    Sphere Function: f(x) = sum(x_i^2)
    """
    return np.sum(position**2)

def initialize_population(grid_size, solution_dim, lower_bound, upper_bound):
    """
    Initialize the cellular grid with random positions in the solution space.
    Each cell is assigned a random position (vector).
    """
    grid = np.random.uniform(lower_bound, upper_bound, size=(grid_size, grid_size, solution_dim))
    return grid

def evaluate_fitness(grid):
    """
    Evaluate the fitness of each cell in the grid based on the optimization function.
    """
    fitness = np.apply_along_axis(sphere_function, 2, grid)
    return fitness
```

```
def get_neighbors(grid, i, j):
    """
    Get the neighboring cells of cell (i, j) in the grid.
    Wraps around the grid edges (toroidal topology).
    """
    neighbors = []
    grid_size = len(grid)
    for di in [-1, 0, 1]:
        for dj in [-1, 0, 1]:
            if di != 0 or dj != 0:  # Exclude the cell itself
                ni, nj = (i + di) % grid_size, (j + dj) % grid_size
                neighbors.append(grid[ni, nj])
    return np.array(neighbors)

def update_states(grid, fitness, learning_rate):
    """
    Update the state (position) of each cell based on the neighbors and predefined rules.
    Each cell moves towards the best position in its neighborhood.
    """
    grid_size, _, solution_dim = grid.shape
    new_grid = np.copy(grid)
    for i in range(grid_size):
        for j in range(grid_size):
            neighbors = get_neighbors(grid, i, j)
            neighbor_fitness = np.array([sphere_function(n) for n in neighbors])
            best_neighbor = neighbors[np.argmin(neighbor_fitness)]
            # Move cell slightly towards the best neighbor's position
            new_grid[i, j] += learning_rate * (best_neighbor - grid[i, j])
    return new_grid
def parallel_cellular_algorithm(
    grid_size=10, solution_dim=2, lower_bound=-5.0, upper_bound=5.0,
    iterations=100, learning_rate=0.1):
    """
    Main function to execute the Parallel Cellular Algorithm.
    """
    # Step 1: Initialize population
    grid = initialize_population(grid_size, solution_dim, lower_bound, upper_bound)
    best_solution = None
    best_fitness = float('inf')

    for iteration in range(iterations):
        # Step 2: Evaluate fitness
        fitness = evaluate_fitness(grid)

        # Track the best solution
        min_idx = np.unravel_index(np.argmin(fitness), fitness.shape)
        current_best = grid[min_idx]
        current_fitness = fitness[min_idx]
```

```
        if current_fitness < best_fitness:
            best_solution = current_best
            best_fitness = current_fitness

        # Step 3: Update states
        grid = update_states(grid, fitness, learning_rate)

        # Print iteration progress
        print(f"Iteration {iteration+1}/{iterations}: Best Fitness = {best_fitness:.5f}")

    # Step 4: Output the best solution
    print("\nOptimization Complete.")
    print(f"Best Solution: {best_solution}")
    print(f"Best Fitness: {best_fitness:.5f}")

# Run the algorithm
if __name__ == "__main__":
    parallel_cellular_algorithm(grid_size=10, solution_dim=2, iterations=10, learning_rate=0.2)
```

Output:

```
Iteration 1/10: Best Fitness = 0.34823
Iteration 2/10: Best Fitness = 0.19787
Iteration 3/10: Best Fitness = 0.04693
Iteration 4/10: Best Fitness = 0.01438
Iteration 5/10: Best Fitness = 0.01100
Iteration 6/10: Best Fitness = 0.00318
Iteration 7/10: Best Fitness = 0.00318
Iteration 8/10: Best Fitness = 0.00318
Iteration 9/10: Best Fitness = 0.00318
Iteration 10/10: Best Fitness = 0.00318

Optimization Complete.
Best Solution: [-0.05362323  0.01746463]
Best Fitness: 0.00318
```

Program 7

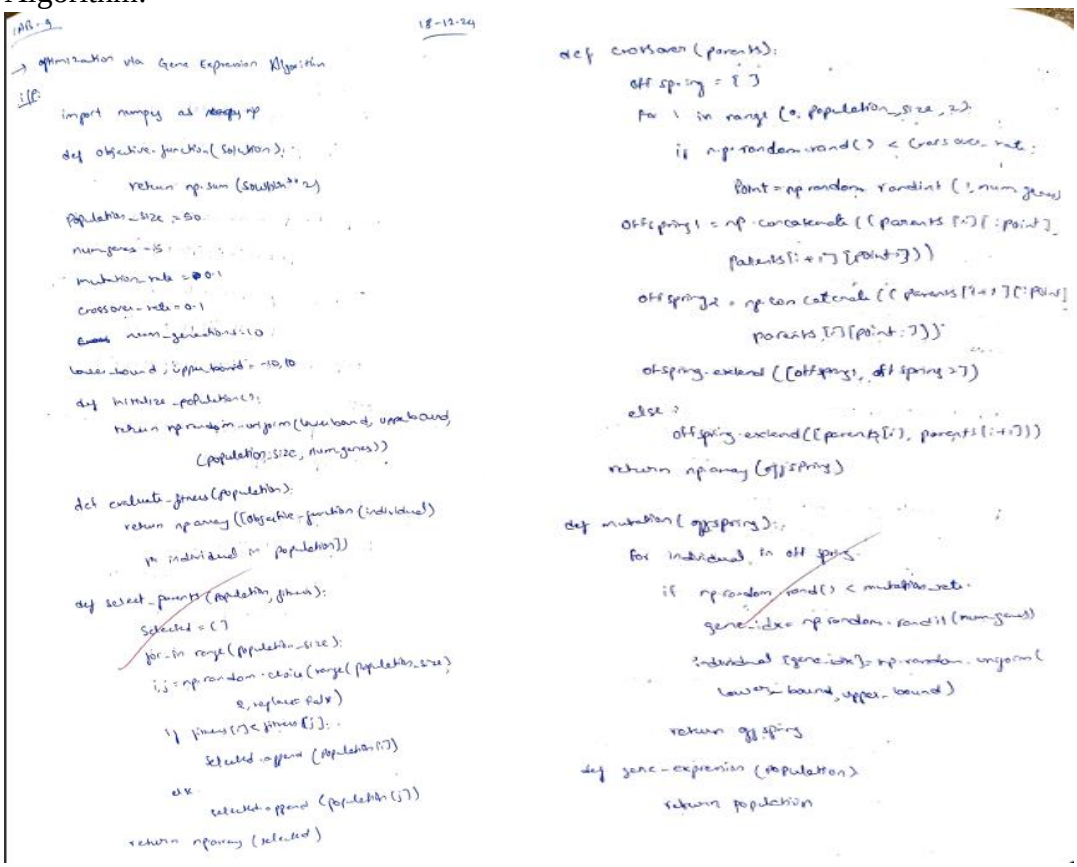
Optimization via Gene Expression Algorithms:

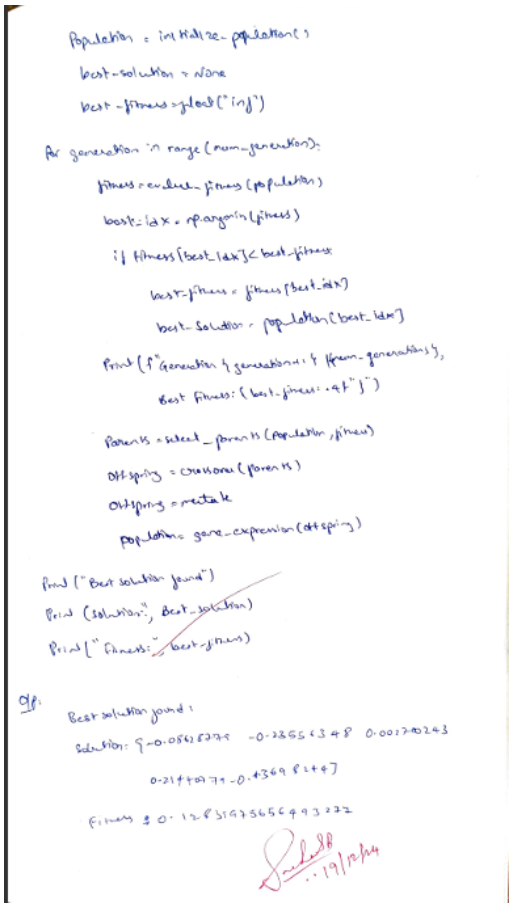
Gene Expression Algorithms (GEA) are inspired by the biological process of gene expression in living organisms. This process involves the translation of genetic information encoded in DNA into functional proteins. In GEA, solutions to optimization problems are encoded in a manner similar to genetic sequences. 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.

Implementation Steps:

1. Define the Problem: Create a mathematical function to optimize.
2. Initialize Parameters: Set the population size, number of genes, mutation rate, crossover rate, and number of generations.
3. Initialize Population: Generate an initial population of random genetic sequences.
4. Evaluate Fitness: Evaluate the fitness of each genetic sequence based on the optimization function.
5. Selection: Select genetic sequences based on their fitness for reproduction.
6. Crossover: Perform crossover between selected sequences to produce offspring.
7. Mutation: Apply mutation to the offspring to introduce variability.
8. Gene Expression: Translate genetic sequences into functional solutions.
9. Iterate: Repeat the selection, crossover, mutation, and gene expression processes for a fixed number of generations or until convergence criteria are met.
10. Output the Best Solution: Track and output the best solution found during the iterations.

Algorithm:





Code:

```
import numpy as np
import random

# Define the Rastrigin function (a well-known benchmark for optimization)
def rastrigin(x):
    A = 10
    return A * len(x) + sum([(xi**2 - A * np.cos(2 * np.pi * xi)) for xi in x])

# Initialize population
def initialize_population(pop_size, num_genes, lower_bound, upper_bound):
    population = np.random.uniform(lower_bound, upper_bound, (pop_size, num_genes))
    return population

# Evaluate fitness of the population
def evaluate_fitness(population):
    fitness = np.array([rastrigin(individual) for individual in population])
    return fitness

# Selection: Tournament selection
def tournament_selection(population, fitness, tournament_size=3):
    selected = []
    for _ in range(len(population)):
```

```
        tournament_indices = np.random.choice(len(population), tournament_size, replace=False)
        tournament_fitness = fitness[tournament_indices]
        winner_idx = tournament_indices[np.argmin(tournament_fitness)] # Minimize the Rastrigin
function
        selected.append(population[winner_idx])
    return np.array(selected)

# Crossover: One-point crossover
def crossover(parent1, parent2):
    crossover_point = np.random.randint(1, len(parent1) - 1)
    child1 = np.concatenate((parent1[:crossover_point], parent2[crossover_point:]))
    child2 = np.concatenate((parent2[:crossover_point], parent1[crossover_point:]))
    return child1, child2

# Mutation: Random mutation
def mutate(child, mutation_rate, lower_bound, upper_bound):
    for i in range(len(child)):
        if np.random.rand() < mutation_rate:
            child[i] = np.random.uniform(lower_bound, upper_bound)
    return child

# Gene expression (mapping genes to real values, already done by direct mapping in this case)
# You can modify this step based on the problem's domain (i.e., gene representation and translation)
# Main GEA function
def gene_expression_algorithm(pop_size, num_genes, lower_bound, upper_bound, mutation_rate,
crossover_rate, num_generations):
    # Step 1: Initialize Population
    population = initialize_population(pop_size, num_genes, lower_bound, upper_bound)

    # Step 2: Iterate for a fixed number of generations
    best_solution = None
    best_fitness = float('inf')

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

        # Step 4: 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 5: Selection
        selected_population = tournament_selection(population, fitness)

        # Step 6: Crossover and Mutation
        new_population = []
        for i in range(0, pop_size, 2):
```

```
            parent1 = selected_population[i]
            parent2 = selected_population[i+1] if i+1 < pop_size else selected_population[0] # Ensuring
even number of parents

            # Perform crossover
            if np.random.rand() < crossover_rate:
                child1, child2 = crossover(parent1, parent2)
            else:
                child1, child2 = parent1, parent2 # No crossover, just pass parents

            # Apply mutation
            child1 = mutate(child1, mutation_rate, lower_bound, upper_bound)
            child2 = mutate(child2, mutation_rate, lower_bound, upper_bound)

            # Add the children to the new population
            new_population.extend([child1, child2])

        # Update population with new generation
        population = np.array(new_population[:pop_size]) # Ensure population size remains constant

    return best_solution, best_fitness

# Set parameters
pop_size = 100                # Population size
num_genes = 10                # Number of genes (dimensions of the problem)
lower_bound = -5.12           # Lower bound of the search space
upper_bound = 5.12            # Upper bound of the search space
mutation_rate = 0.1           # Mutation rate
crossover_rate = 0.8          # Crossover rate
num_generations = 500         # Number of generations

# Run the Gene Expression Algorithm
best_solution, best_fitness = gene_expression_algorithm(pop_size, num_genes, lower_bound,
upper_bound, mutation_rate, crossover_rate, num_generations)

# Output the results
print("Best Solution:", best_solution)
print("Best Fitness:", best_fitness)
```

Output:

```
Best Solution: [ 0.01956405  0.00271381 -0.00243719  0.00141388 -0.02586832  0.00105932
  0.01769152 -1.03340239 -0.02943199 -0.04696745]
Best Fitness: 2.166804134722355
```