VISVESVARAYA TECHNOLOGICAL UNIVERSITY

“JnanaSangama”, Belgaum -590014, Karnataka.

LAB RECORD

Bio Inspired Systems (23CS5BSBIS)

Submitted by

Shreya Bharamanna Patil(1BM23CS420)

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 **Shreya Bharamanna Patil (1BM23CS420),** 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 mention subject and the work prescribed for the said degree.

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

Index

Github Link:
https://github.com/PatilShreya22/BIS

Program 1

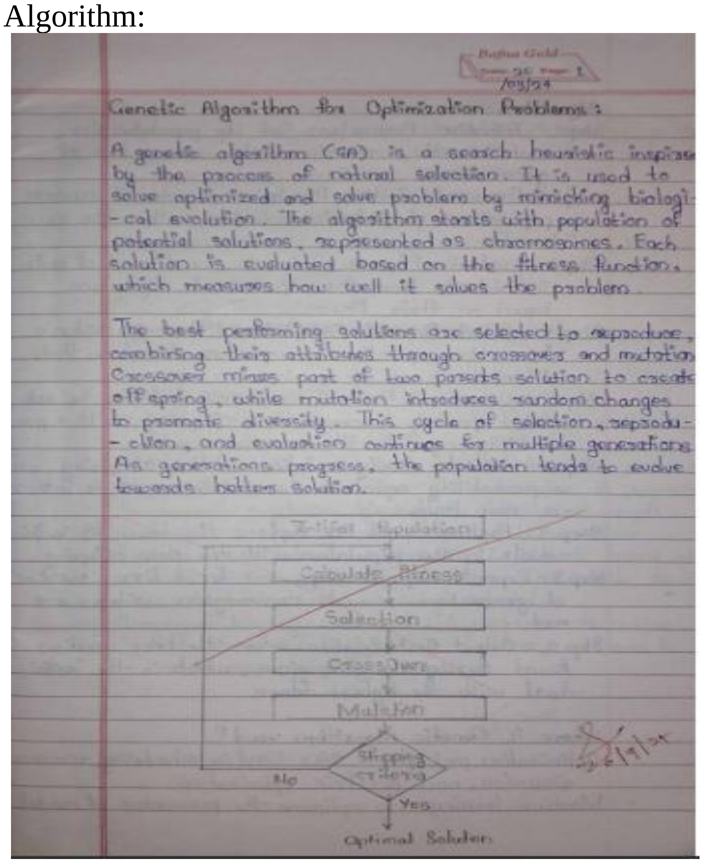
Genetic Algorithm for Optimization Problems

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:



Genetic Algorithm for Optimization Problems :

A genetic algorithm (GA) is a search heuristic inspired by the process of natural selection. It is used to solve optimized and solve problem by mimicking biological evolution. The algorithm starts with population of potential solutions, represented as chromosomes. Each solution is evaluated based on the fitness function, which measures how well it solves the problem.

The best performing solutions are selected to reproduce, combining their attributes through crossover and mutation. Crossover mixes part of two parents solution to create offspring, while mutation introduces random changes to promote diversity. This cycle of selection, reproduction, and evaluation continues for multiple generations. As generations progress, the population tends to evolve towards better solution.

Algorithm

Step 1: Initialize Parameters: Set the population size, mutation rate, crossover rate, and number of generations.

Step 2: Generate Initial Population: Create a random population of potential solutions within the given bounds.

Step 3: Evaluate Fitness: Calculate the fitness of each individuals from the population to reproduce, based on their fitness.

Step 4: Select Parents: Select the fittest individuals from the population to reproduce, based on their fitness.

Step 5: Crossover: Perform crossover between the selected parents to create new offspring, with a probability equal to the crossover rate.

Step 6: Mutate: Apply mutation to the offspring, with a probability equal to the mutation rate, to introduce new traits.

Step 7: Replace Least Fit: Replace the least fit individuals in the population with the new offspring.

Step 8: Repeat: Repeat steps 4-7 for a fixed number of generations or until convergence criteria are met.

Step 9: Output Best Solution: return the best solution found during the generations, which is the individual with the highest fitness.

Where is Genetic Algorithm used?

- Optimization problems: are used in scheduling, resource allocation, and portfolio optimization.
- Machine learning: to optimize the parameters of model such as neural networks and decision trees.
- Engineering design: Design of complex systems, such as electronic circuits, mechanical systems, and aerospace systems.
- Finance: to optimize portfolio management, risk management, and trading strategies.
- Computer networks: Optimize network routing, scheduling and resource allocation.

Optimizing Techniques:

- Selection: Selecting the fittest individuals from the population to reproduce.
- Crossover: Combining the genetic information of two parents to create a new offspring.
- Mutation: Randomly changing the genetic information of an individual to introduce new traits.
- Elitism: Preserving the best solutions from the previous generation to ensure that the population to progress.
- Tournament selection: Selecting the fittest individuals from a subset of the population to reproduce.
- Roulette wheel selection: Selecting individuals based on their fitness, with higher fitness individuals having a higher chance of being selected.
- Simulated binary crossover: A crossover technique that simulates the process of binary crossover to create new offspring.

```
import random
import numpy as np

def objective_function(x):
    return x**2 + 2*x + 1

def generate_initial_population(population_size, bounds):
    population = []
    for _ in range(population_size):
        x = random.uniform(bounds[0], bounds[1])
        population.append(x)
    return population

def evaluate_fitness(population):
    fitness = []
    for x in population:
        fitness.append(objective_function(x))
    return fitness

def selection(population, fitness, num_parents):
    parents = []
    for _ in range(num_parents):
        max_fitness_idx = np.argmax(fitness)
        parents.append(population[max_fitness_idx])
        fitness[max_fitness_idx] = -float("inf")
    return parents

def crossover(parents, crossover_rate):
    offspring = []
    for _ in range(len(offspring)):
        if random.random() < mutation_rate:
            offspring[i] += random.uniform(-0.1, 0.1)
                            * (bounds[1] - bounds[0])
            offspring[i] = max(bounds[0], min(offspring[i],
                                              bounds[1]))
    return offspring

def mutation(offspring, mutation_rate, bounds):
    offspring = []
    for _ in range(len(parents)//2):
        parent1, parent2 = random.sample(parents, 2)
        if random.random() < crossover_rate:
            x1, x2 = parent1, parent2
            x = offspring.append(x, offspring)
    return offspring

def genetic_algorithm(population_size, mutation_rate,
        crossover_rate, num_generations, bounds):
    population = generate_initial_population(population_
                 size, bounds)
    for generation in range(num_generations):
        fitness = evaluate_fitness(population)
        parents = selection(population, fitness,
                  population_size // 2)
        offspring = crossover(parents, crossover_rate)
        offspring = mutation(offspring, mutation_rate,
                    bounds)
        population = offspring + parents
    best_solution = min(population, key=objective_
                    function)
    return best_solution

# Set parameters
population_size = 100
mutation_rate = 0.01
crossover_rate = 0.5
num_generations = 100
bounds = (-10, 10)

best_solution = genetic_algorithm(population_size,
    mutation_rate, crossover_rate, num_generations,
    bounds)
print("Best solution :", best_solution)
```

o/p : Best solution : 9.9988299884834992

Code:

```
#GENETIC ALGORITHM
import numpy as np
import random

# Define the fitness function
def fitness_function(x):
    return x ** 2

# Initialize parameters
population_size = 100
mutation_rate = 0.1
num_generations = 50
bounds = (-10, 10)

# Step 1: Create initial population
def create_initial_population(size, bounds):
    return [random.uniform(bounds[0], bounds[1]) for _ in range(size)]

# Step 2: Evaluate fitness of the population
def evaluate_population(population):
    return [fitness_function(individual) for individual in population]

# Step 3: Selection using roulette-wheel selection
def selection(population, fitness):
    total_fitness = sum(fitness)
    selection_probs = [f / total_fitness for f in fitness]
    return np.random.choice(population, size=2, p=selection_probs)

# Step 4: Crossover operation
def crossover(parent1, parent2):
    alpha = random.uniform(0, 1)
    offspring1 = alpha * parent1 + (1 - alpha) * parent2
    offspring2 = alpha * parent2 + (1 - alpha) * parent1
    return offspring1, offspring2

# Step 5: Mutation operation
def mutate(individual, bounds):
    if random.random() < mutation_rate:
        return random.uniform(bounds[0], bounds[1])
    return individual

# Main Genetic Algorithm loop
def genetic_algorithm(bounds):
    # Step 1: Create initial population
    population = create_initial_population(population_size, bounds)
```

```python
    best_solution = None
    best_fitness = float('-inf')

    for generation in range(num_generations):
        # Step 2: Evaluate fitness
        fitness = evaluate_population(population)

        # Track the best solution
        current_best_fitness = max(fitness)
        if current_best_fitness > best_fitness:
            best_fitness = current_best_fitness
            best_solution = population[fitness.index(current_best_fitness)]

        # Step 3: Create new population
        new_population = []

        while len(new_population) < population_size:
            parent1, parent2 = selection(population, fitness)
            offspring1, offspring2 = crossover(parent1, parent2)
            new_population.append(mutate(offspring1, bounds))
            new_population.append(mutate(offspring2, bounds))

        # Replace the old population with the new population
        population = new_population[:population_size]

    return best_solution, best_fitness

# Run the Genetic Algorithm
best_solution, best_fitness = genetic_algorithm(bounds)

print(f"Best Solution: x = {best_solution}")
print(f"Best Fitness: f(x) = {best_fitness}")
```

Output:

```
Best Solution: x = 9.97704555295002
Best Fitness: f(x) = 99.54143796563977
```

Program 2

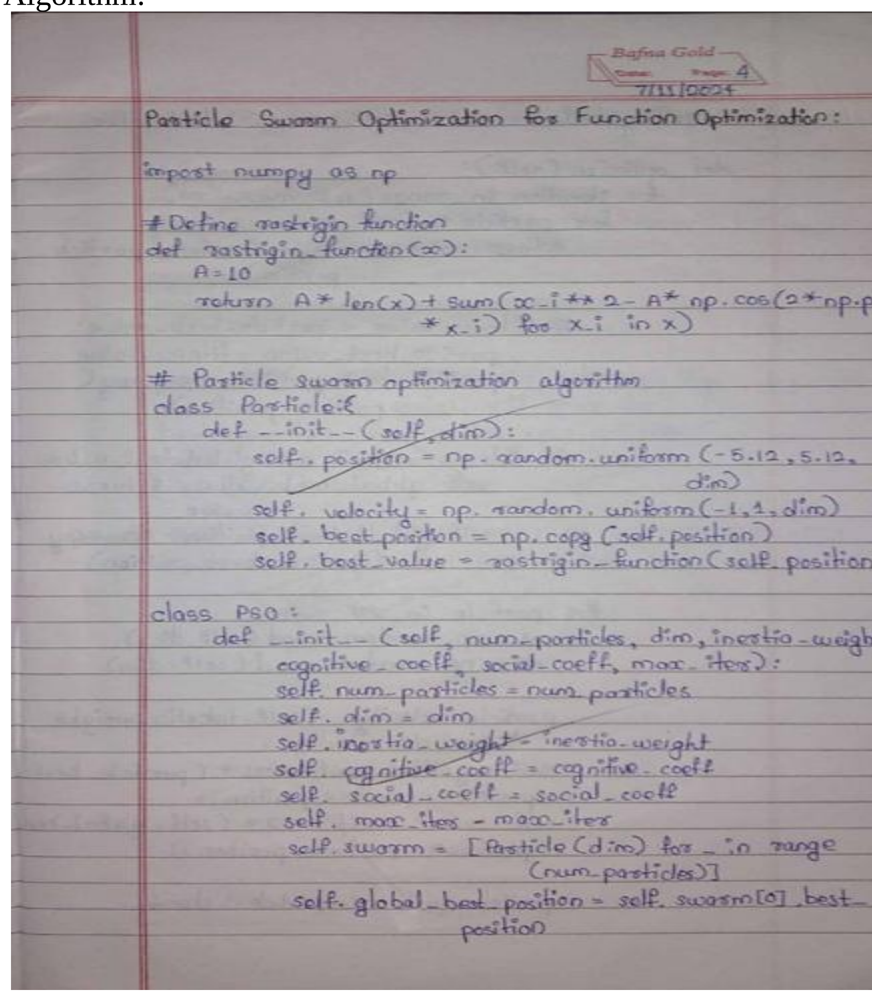
Particle Swarm Optimization for Function Optimization

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.

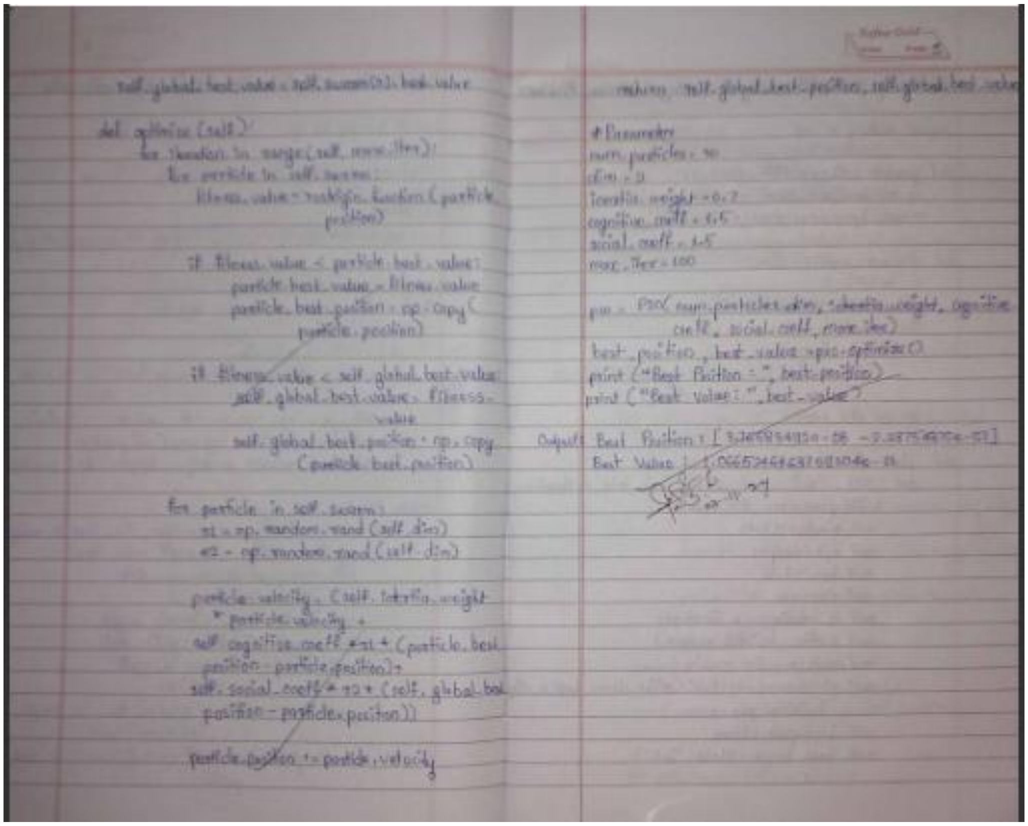
Algorithm:



Date: 7/11/2024 Page: 4

Particle Swarm Optimization for Function Optimization:

```
import numpy as np

#Define rastrigin function
def rastrigin_function(x):
    A=10
    return A * len(x) + sum(x_i ** 2 - A * np.cos(2*np.pi
           * x_i) for x_i in x)

# Particle swarm optimization algorithm
class Particle:{
    def __init__(self, dim):
        self.position = np.random.uniform(-5.12, 5.12,
                                          dim)
        self.velocity = np.random.uniform(-1, 1, dim)
        self.best_position = np.copy(self.position)
        self.best_value = rastrigin_function(self.position)

class PSO:
    def __init__(self, num_particles, dim, inertia_weight,
         cognitive_coeff, social_coeff, max_iters):
        self.num_particles = num_particles
        self.dim = dim
        self.inertia_weight = inertia_weight
        self.cognitive_coeff = cognitive_coeff
        self.social_coeff = social_coeff
        self.max_iters = max_iters
        self.swarm = [Particle(dim) for _ in range
                      (num_particles)]
        self.global_best_position = self.swarm[0].best_
            position
```



Code:
#PARTICLE SWARM OPTIMIZATION

```
import numpy as np

# Rastrigin function: A benchmark function for optimization problems
def rastrigin(x):
    A = 10
    # Calculate the Rastrigin function value based on the input vector x
    return A * len(x) + sum(x_i**2 - A * np.cos(2 * np.pi * x_i) for x_i in x)

# Particle Swarm Optimization class
class ParticleSwarmOptimizer:
    def __init__(self, func, n_particles, n_dimensions, n_iterations, inertia_weight=0.7,
cognitive_coeff=1.5, social_coeff=1.5, bounds=(-5.12, 5.12)):
        self.func = func  # The function to optimize
        self.n_particles = n_particles  # Number of particles in the swarm
        self.n_dimensions = n_dimensions  # Dimensions of the search space
        self.n_iterations = n_iterations  # Number of iterations for the optimization
        self.lower_bound, self.upper_bound = bounds  # Bounds for the search space

        # Initialize particle positions randomly within the specified bounds
        self.positions = np.random.uniform(self.lower_bound, self.upper_bound, (n_particles,
n_dimensions))
```

```
        # Initialize particle velocities randomly
        self.velocities = np.random.uniform(-1, 1, (n_particles, n_dimensions))
        # Personal best positions and scores for each particle
        self.pbest_positions = np.copy(self.positions)
        self.pbest_scores = np.array([func(p) for p in self.positions])  # Evaluate initial fitness
        # Global best position and score among all particles
        self.gbest_position = self.pbest_positions[np.argmin(self.pbest_scores)]
        self.gbest_score = np.min(self.pbest_scores)

    def optimize(self):
        # Main loop for the optimization process
        for _ in range(self.n_iterations):
            for i in range(self.n_particles):
                # Evaluate the fitness of the current position
                fitness = self.func(self.positions[i])
                # Update personal best if the current fitness is better
                if fitness < self.pbest_scores[i]:
                    self.pbest_scores[i] = fitness
                    self.pbest_positions[i] = self.positions[i]
                # Update global best if the current fitness is better
                if fitness < self.gbest_score:
                    self.gbest_score = fitness
                    self.gbest_position = self.positions[i]

            # Generate random coefficients for cognitive and social components
            r1, r2 = np.random.rand(self.n_dimensions), np.random.rand(self.n_dimensions)
            # Update velocities based on inertia, personal best, and global best
            self.velocities = (self.velocities * 0.7 +  # Inertia weight
                               1.5 * r1 * (self.pbest_positions - self.positions) +  # Cognitive component
                               1.5 * r2 * (self.gbest_position - self.positions))  # Social component
            # Update positions based on new velocities and clip to stay within bounds
            self.positions = np.clip(self.positions + self.velocities, self.lower_bound, self.upper_bound)

            # Print the best fitness found so far in this iteration
            print(f"Best Fitness: {self.gbest_score}")

        # Return the best position and score found after all iterations
        return self.gbest_position, self.gbest_score

# Create and run the optimizer
pso = ParticleSwarmOptimizer(func=rastrigin, n_particles=30, n_dimensions=2, n_iterations=100)
best_position, best_score = pso.optimize()

# Print the best position and corresponding fitness score found
print("\nBest Position Found:", best_position)
print("Best Fitness Score:", best_score)
```

Output:

```
Best Fitness: 7.523349690449162
Best Fitness: 5.479012944526062
Best Fitness: 5.479012944526062
Best Fitness: 5.479012944526062
Best Fitness: 5.479012944526062
Best Fitness: 5.479012944526062
Best Fitness: 5.479012944526062
Best Fitness: 5.479012944526062
Best Fitness: 5.35158483420342
Best Fitness: 4.23336222695108
Best Fitness: 2.3059731550465656
Best Fitness: 2.3059731550465656
Best Fitness: 2.3059731550465656
Best Fitness: 2.3059731550465656
Best Fitness: 2.2923779383497873
Best Fitness: 2.2923779383497873
Best Fitness: 2.2923779383497873
Best Fitness: 2.2923779383497873
Best Fitness: 2.2923779383497873
Best Fitness: 2.2923779383497873
Best Fitness: 2.2923779383497873
Best Fitness: 2.2923779383497873
Best Fitness: 2.2923779383497873
Best Fitness: 1.4393114014934305
Best Fitness: 1.4393114014934305
Best Fitness: 1.3002025319518147
Best Fitness: 1.3002025319518147
Best Fitness: 1.3002025319518147
```

```
Best Fitness: 0.00042484148907107055
Best Fitness: 0.00042484148907107055
Best Fitness: 0.00019896058490331825
Best Fitness: 0.00019896058490331825
Best Fitness: 0.00019896058490331825
Best Fitness: 0.00019896058490331825
Best Fitness: 0.00019896058490331825
Best Fitness: 0.00019896058490331825
Best Fitness: 9.185587018123442e-06
Best Fitness: 9.185587018123442e-06
Best Fitness: 9.185587018123442e-06
Best Fitness: 9.185587018123442e-06
Best Fitness: 9.185587018123442e-06
Best Fitness: 9.185587018123442e-06
Best Fitness: 9.185587018123442e-06
Best Fitness: 8.17354336390963e-06
Best Fitness: 8.17354336390963e-06
Best Fitness: 3.993851240835511e-06
Best Fitness: 3.993851240835511e-06
Best Fitness: 3.993851240835511e-06
Best Fitness: 3.993851240835511e-06
Best Fitness: 1.6462023069152565e-06
Best Fitness: 1.6462023069152565e-06
Best Fitness: 1.6462023069152565e-06
Best Fitness: 1.6462023069152565e-06
Best Fitness: 1.6462023069152565e-06
Best Fitness: 1.6462023069152565e-06
Best Fitness: 1.6462023069152565e-06
Best Fitness: 1.5764355048020207e-06

Best Position Found: [3.76308963e-05 8.08082678e-05]
Best Fitness Score: 1.5764355048020207e-06
```

<u>Program 3</u>

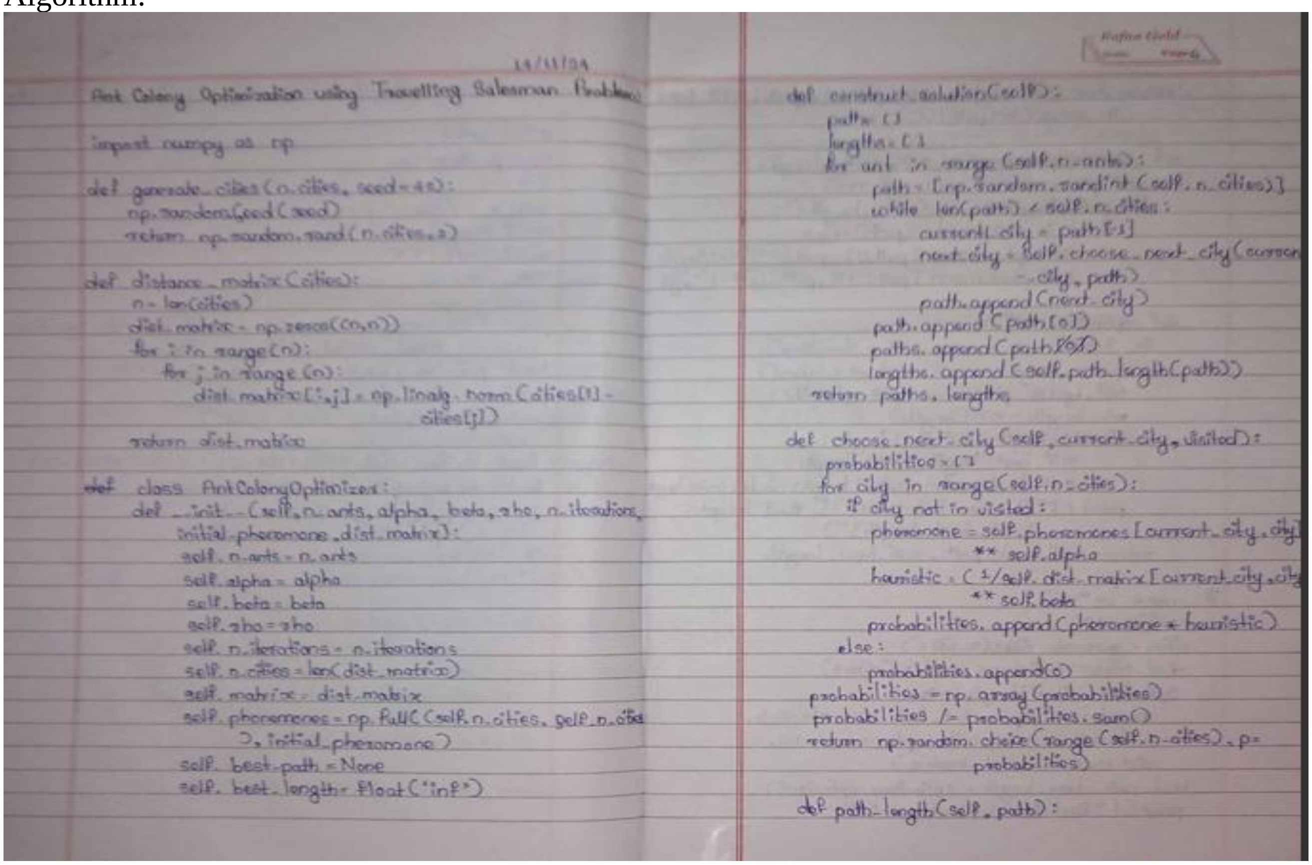
Ant Colony Optimization for the Traveling Salesman Problem

The foraging behaviour 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.

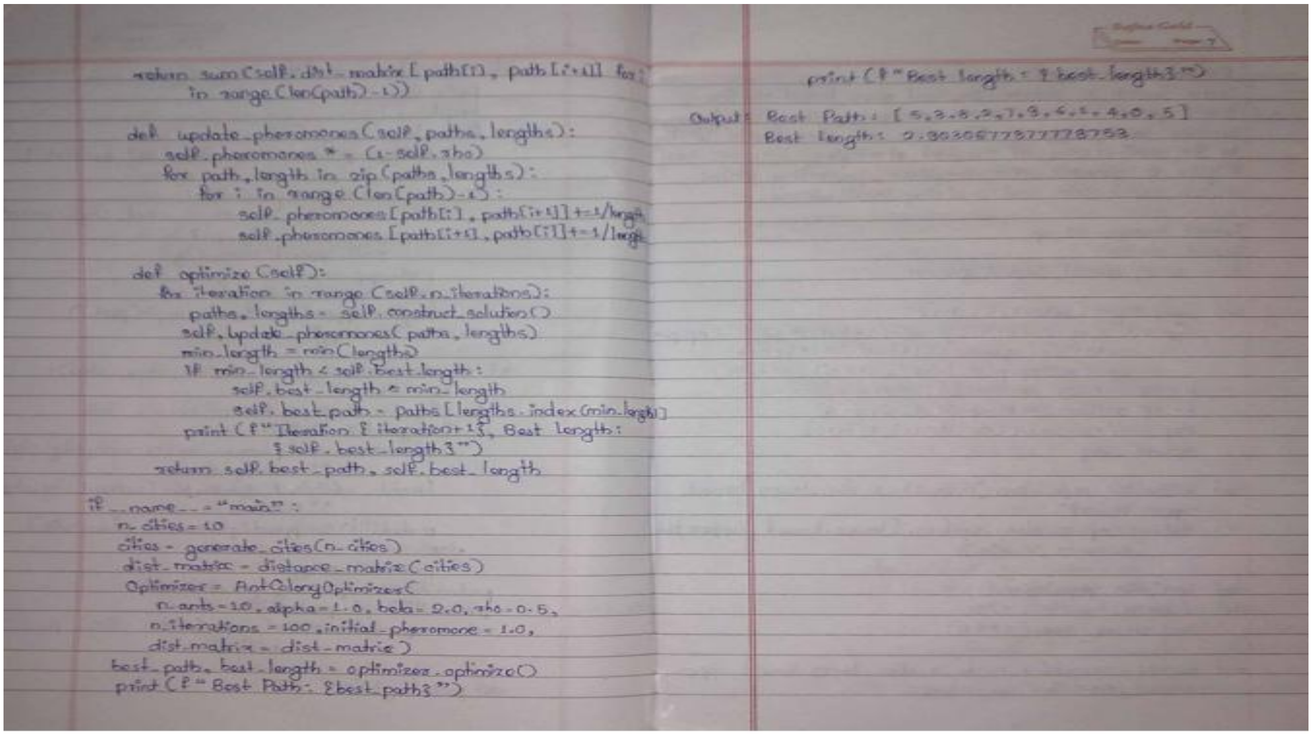
Implementation Steps:

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:



```
14/11/24
Ant Colony Optimization using Travelling Salesman Problem

import numpy as np

def generate_cities(n_cities, seed=42):
    np.random.seed(seed)
    return np.random.rand(n_cities, 2)

def distance_matrix(cities):
    n = len(cities)
    dist_matrix = np.zeros((n,n))
    for i in range(n):
        for j in range(n):
            dist_matrix[i,j] = np.linalg.norm(cities[i] -
                                    cities[j])
    return dist_matrix

class AntColonyOptimizer:
    def __init__(self, n_ants, alpha, beta, rho, n_iterations,
            initial_pheromone, dist_matrix):
        self.n_ants = n_ants
        self.alpha = alpha
        self.beta = beta
        self.rho = rho
        self.n_iterations = n_iterations
        self.n_cities = len(dist_matrix)
        self.matrix = dist_matrix
        self.pheromones = np.full((self.n_cities, self.n_cities
            ), initial_pheromone)
        self.best_path = None
        self.best_length = float("inf")

def construct_solution(self):
    paths = []
    lengths = []
    for ant in range(self.n_ants):
        path = [np.random.randint(self.n_cities)]
        while len(path) < self.n_cities:
            current_city = path[-1]
            next_city = self.choose_next_city(current
                    _city, path)
            path.append(next_city)
        path.append(path[0])
        paths.append(path)
        lengths.append(self.path_length(path))
    return paths, lengths

def choose_next_city(self, current_city, visited):
    probabilities = []
    for city in range(self.n_cities):
        if city not in visited:
            pheromone = self.pheromones[current_city, city]
                    ** self.alpha
            heuristic = (1/self.dist_matrix[current_city, city
                    ** self.beta
            probabilities.append(pheromone * heuristic)
        else:
            probabilities.append(0)
    probabilities = np.array(probabilities)
    probabilities /= probabilities.sum()
    return np.random.choice(range(self.n_cities), p=
                    probabilities)

def path_length(self, path):
```



```
return sum(self.dist_matrix[path[i], path[i+1]] for i
    in range(len(path)-1))

def update_pheromones(self, paths, lengths):
    self.pheromones *= (1-self.rho)
    for path, length in zip(paths, lengths):
        for i in range(len(path)-1):
            self.pheromones[path[i], path[i+1]] += 1/length
            self.pheromones[path[i+1], path[i]] += 1/length

def optimize(self):
    for iteration in range(self.n_iterations):
        paths, lengths = self.construct_solution()
        self.update_pheromones(paths, lengths)
        min_length = min(lengths)
        if min_length < self.best_length:
            self.best_length = min_length
            self.best_path = paths[lengths.index(min_length)]
        print(f"Iteration {iteration+1}, Best length:
                {self.best_length}")
    return self.best_path, self.best_length

if __name__ == "main":
    n_cities = 10
    cities = generate_cities(n_cities)
    dist_matrix = distance_matrix(cities)
    Optimizer = AntColonyOptimizer(
        n_ants=10, alpha=1.0, beta=2.0, rho=0.5,
        n_iterations=100, initial_pheromone=1.0,
        dist_matrix=dist_matrix)
    best_path, best_length = optimizer.optimize()
    print(f"Best Path: {best_path}")
    print(f"Best length: {best_length}")

Output: Best Path: [5, 3, 8, 2, 7, 9, 6, 1, 4, 0, 5]
        Best length: 2.3030677877778753
```

Code:
#ANT COLONY OPTIMIZATION

```
import random
import numpy as np

# Distance calculation (Euclidean distance)
def euclidean_distance(city1, city2):
    return np.sqrt((city1[0] - city2[0])**2 + (city1[1] - city2[1])**2)

# Ant Colony Optimization Algorithm
class ACO:
    def __init__(self, cities, num_ants=10, num_iterations=100, alpha=1.0, beta=2.0, rho=0.5, Q=100):
        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 information (distance)
        self.rho = rho           # Pheromone evaporation rate
        self.Q = Q               # Total pheromone deposited per ant per tour
        self.num_cities = len(cities)

        # Initialize pheromone matrix (for each pair of cities)
        self.pheromone = np.ones((self.num_cities, self.num_cities)) / self.num_cities
        self.distances = np.zeros((self.num_cities, self.num_cities))

        # Compute distance matrix
        for i in range(self.num_cities):
```

```
        for j in range(i + 1, self.num_cities):
            self.distances[i][j] = self.distances[j][i] = euclidean_distance(cities[i], cities[j])

def _choose_next_city(self, ant, visited):
    # Calculate the probability of moving to each city
    current_city = ant[-1]
    probabilities = []

    for i in range(self.num_cities):
        if i not in visited:
            pheromone = self.pheromone[current_city][i] ** self.alpha
            heuristic = (1.0 / self.distances[current_city][i]) ** self.beta
            probabilities.append(pheromone * heuristic)
        else:
            probabilities.append(0)

    # Normalize probabilities
    total = sum(probabilities)
    if total == 0:  # In case there's no valid path (shouldn't happen with good settings)
        return random.choice([i for i in range(self.num_cities) if i not in visited])

    probabilities = [prob / total for prob in probabilities]

    # Choose next city based on probabilities
    next_city = random.choices(range(self.num_cities), probabilities)[0]
    return next_city

def _construct_solution(self):
    # Each ant starts at a random city
    ant = [random.randint(0, self.num_cities - 1)]
    visited = set(ant)

    while len(ant) < self.num_cities:
        next_city = self._choose_next_city(ant, visited)
        ant.append(next_city)
        visited.add(next_city)

    # Return to the starting city
    ant.append(ant[0])

    return ant

def _evaluate_solution(self, solution):
    # Calculate the total distance of the tour
    total_distance = 0
    for i in range(len(solution) - 1):
        total_distance += self.distances[solution[i]][solution[i + 1]]
```

```
        return total_distance

    def _update_pheromone(self, all_solutions):
        # Initialize pheromone update matrix
        pheromone_delta = np.zeros((self.num_cities, self.num_cities))

        # For each solution, deposit pheromone
        for solution in all_solutions:
            tour_length = self._evaluate_solution(solution)
            for i in range(len(solution) - 1):
                pheromone_delta[solution[i]][solution[i + 1]] += self.Q / tour_length

        # Evaporate pheromone
        self.pheromone = (1 - self.rho) * self.pheromone + pheromone_delta

    def solve(self):
        best_solution = None
        best_distance = float('inf')

        for iteration in range(self.num_iterations):
            all_solutions = []

            # Each ant constructs a solution
            for ant in range(self.num_ants):
                solution = self._construct_solution()
                all_solutions.append(solution)
                tour_length = self._evaluate_solution(solution)

                # Update best solution if necessary
                if tour_length < best_distance:
                    best_solution = solution
                    best_distance = tour_length

            # Update pheromones based on solutions found
            self._update_pheromone(all_solutions)

            print(f"Iteration {iteration + 1}, Best Distance: {best_distance}")

        return best_solution, best_distance

# Function to take user input for cities
def get_user_input():
    num_cities = int(input("Enter the number of cities: "))
    cities = []

    print("Enter the coordinates of each city (x, y):")
    for i in range(num_cities):
```

```
        x, y = map(float, input(f"City {i+1}: ").split())
        cities.append((x, y))

    return cities

# Example usage:
if __name__ == "__main__":
    # Take user input for cities
    cities = get_user_input()

    # Take user input for ACO parameters
    num_ants = int(input("Enter the number of ants: "))
    num_iterations = int(input("Enter the number of iterations: "))
    alpha = float(input("Enter the value of alpha (pheromone importance): "))
    beta = float(input("Enter the value of beta (distance importance): "))
    rho = float(input("Enter the value of rho (pheromone evaporation rate): "))
    Q = float(input("Enter the value of Q (pheromone deposit per ant): "))

    # Create an instance of ACO and solve the problem
    aco = ACO(cities, num_ants, num_iterations, alpha, beta, rho, Q)
    best_solution, best_distance = aco.solve()

    print(f"\nBest Solution (Tour): {best_solution}")
    print(f"Best Distance: {best_distance}")
```

Output:

```
Enter the number of cities: 5
Enter the coordinates of each city (x, y):
City 1: 0 0
City 2: 1 3
City 3: 4 3
City 4: 6 1
City 5: 3 0
Enter the number of ants: 10
Enter the number of iterations: 100
Enter the value of alpha (pheromone importance): 1.0
Enter the value of beta (distance importance): 2.0
Enter the value of rho (pheromone evaporation rate): 0.5
Enter the value of Q (pheromone deposit per ant): 100
```

```
Best Solution (Tour): [1, 0, 4, 3, 2, 1]
Best Distance: 15.15298244508295
```

Program 4

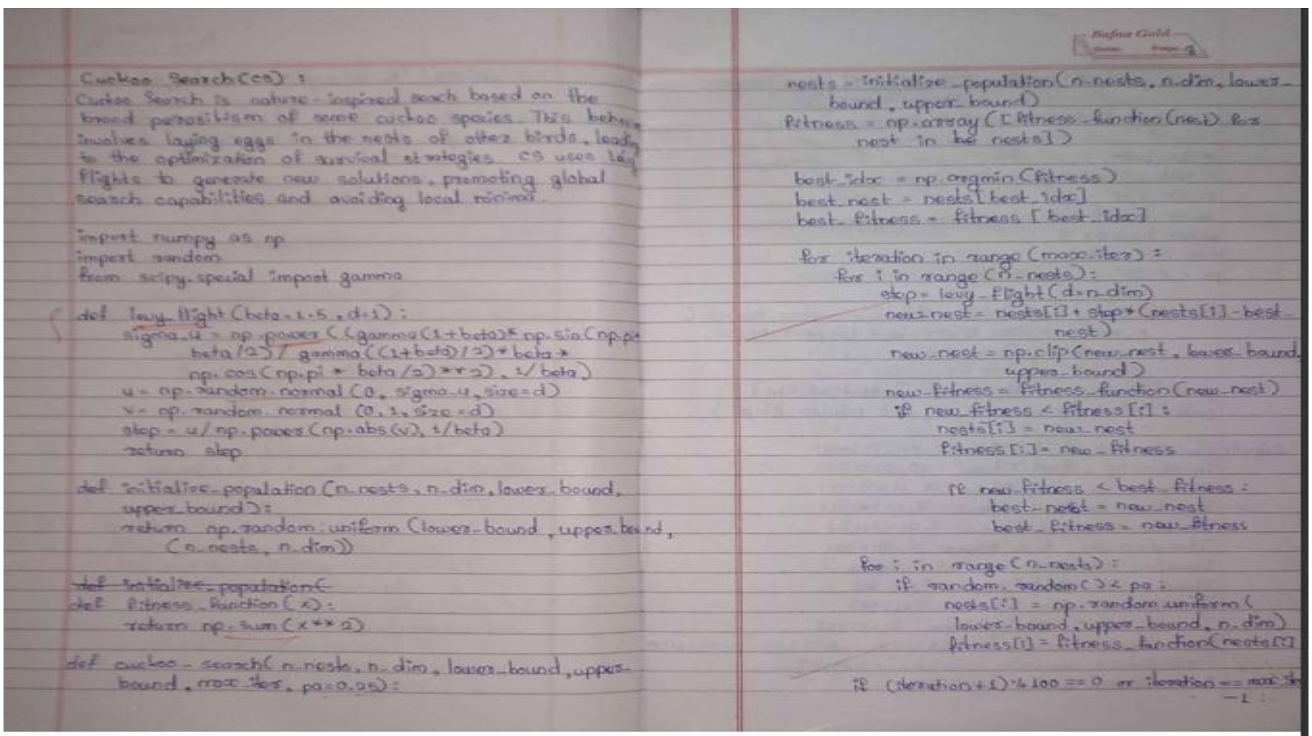
Cuckoo Search (CS)

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.

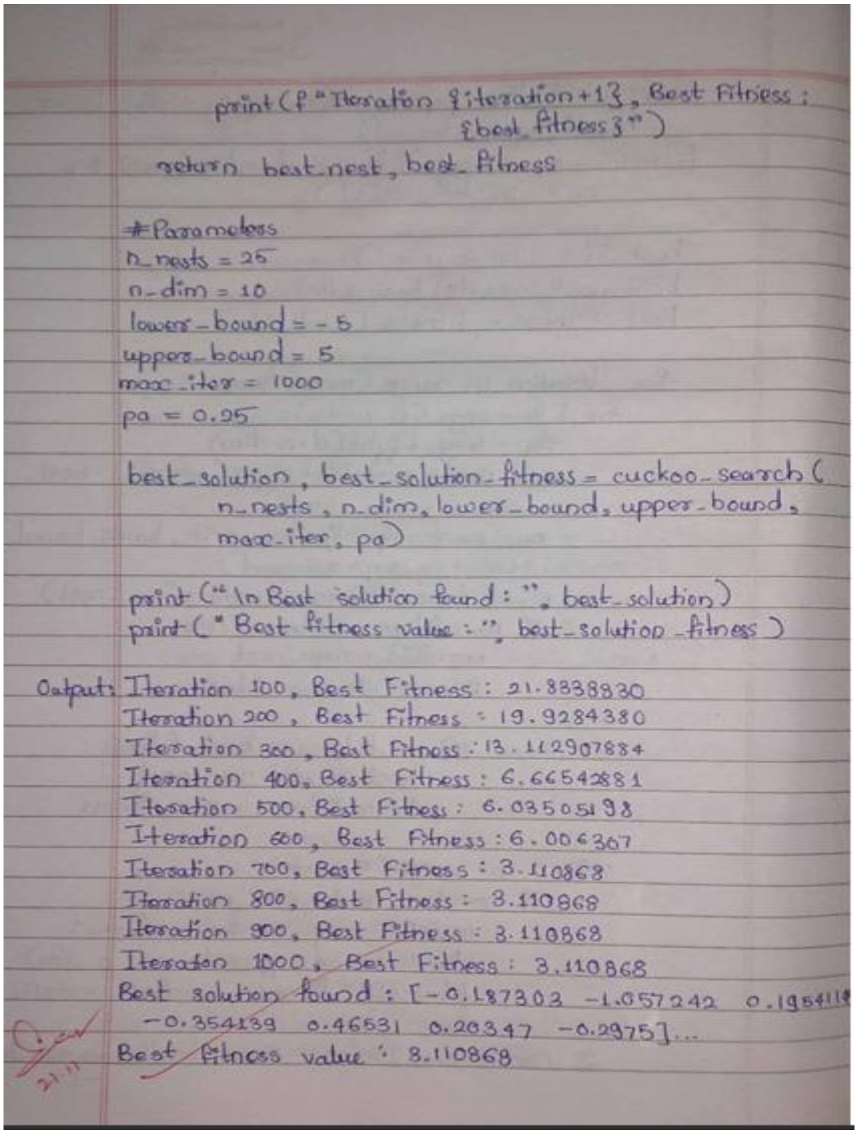
Algorithm:



Cuckoo Search (CS) :

Cuckoo Search is nature-inspired search 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.

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

def levy_flight(beta=1.5, d=1):
    sigma_u = np.power((gamma(1+beta)* np.sin(np.pi*
            beta/2))/ gamma((1+beta)/2)* beta *
            np.cos(np.pi * beta/2)**2), 1/beta)
    u = np.random.normal(0, sigma_u, size=d)
    v = np.random.normal(0, 1, size=d)
    step = u/np.power(np.abs(v), 1/beta)
    return step

def initialize_population(n_nests, n_dim, lower_bound,
    upper_bound):
    return np.random.uniform(lower_bound, upper_bound,
        (n_nests, n_dim))

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

def cuckoo_search(n_nests, n_dim, lower_bound, upper_
    bound, max_iter, pa=0.25):

nests = initialize_population(n_nests, n_dim, lower_
    bound, upper_bound)
fitness = np.array([fitness_function(nest) for
    nest in nests])

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

for iteration in range(max_iter):
    for i in range(n_nests):
        step = levy_flight(d=n_dim)
        new_nest = nests[i] + step*(nests[i] - best_
                        nest)
        new_nest = np.clip(new_nest, lower_bound,
                    upper_bound)
        new_fitness = fitness_function(new_nest)
        if new_fitness < fitness[i]:
            nests[i] = new_nest
            fitness[i] = new_fitness

            if new_fitness < best_fitness:
                best_nest = new_nest
                best_fitness = new_fitness

    for i in range(n_nests):
        if random.random() < pa:
            nests[i] = np.random.uniform(
            lower_bound, upper_bound, n_dim)
            fitness[i] = fitness_function(nests[i]

    if (iteration+1) % 100 == 0 or iteration == max_iter
                                                -1 :
```



```
    print(f"Iteration {iteration+1}, Best Fitness:
                          {best_fitness}")
  return best_nest, best_fitness

#Parameters
n_nests = 25
n_dim = 10
lower_bound = -5
upper_bound = 5
max_iter = 1000
pa = 0.25

best_solution, best_solution_fitness = cuckoo_search(
        n_nests, n_dim, lower_bound, upper_bound,
        max_iter, pa)

print("\n Best solution found:", best_solution)
print("Best fitness value:", best_solution_fitness)
```

Output: Iteration 100, Best Fitness: 21.8838830
Iteration 200, Best Fitness : 19.9284380
Iteration 300, Best Fitness : 13.112907884
Iteration 400, Best Fitness : 6.66542881
Iteration 500, Best Fitness : 6.03505198
Iteration 600, Best Fitness : 6.006307
Iteration 700, Best Fitness : 3.110868
Iteration 800, Best Fitness : 3.110868
Iteration 900, Best Fitness : 3.110868
Iteration 1000, Best Fitness : 3.110868
Best solution found : [-0.187303 -1.057242 0.195411
-0.354139 0.46531 0.20347 -0.2975]...
Best fitness value : 3.110868

21.11

Code:
#CUCKOO SEARCH

```
import numpy as np
import math  # Import the standard math module
def levy_flight(Lambda):
    sigma = (math.gamma(1 + Lambda) * math.sin(math.pi * Lambda / 2) /
             (math.gamma((1 + Lambda) / 2) * Lambda * 2 ** ((Lambda - 1) / 2))) ** (1 / Lambda)
    u = np.random.normal(0, sigma, 1)
    v = np.random.normal(0, 1, 1)
    step = u / abs(v) ** (1 / Lambda)
    return step

def cuckoo_search(obj_function, bounds, n=25, pa=0.25, max_iter=100):
    # Initialize nests
    dim = len(bounds)
```

```
    nests = np.random.rand(n, dim)
    for i in range(dim):
        nests[:, i] = nests[:, i] * (bounds[i][1] - bounds[i][0]) + bounds[i][0]
    fitness = np.array([obj_function(nest) for nest in nests])

    # Start optimization
    for _ in range(max_iter):
        for i in range(n):
            # Generate a new solution via Levy flight
            new_nest = nests[i] + levy_flight(1.5) * np.random.randn(dim)
            # Apply bounds
            new_nest = np.clip(new_nest, [b[0] for b in bounds], [b[1] for b in bounds])
            new_fitness = obj_function(new_nest)
            # Update if new solution is better
            if new_fitness < fitness[i]:
                nests[i] = new_nest
                fitness[i] = new_fitness

        # Abandon some nests and create new ones
        abandon_idx = np.random.rand(n) < pa
        for i in np.where(abandon_idx)[0]:
            nests[i] = np.random.rand(dim) * (np.array([b[1] for b in bounds]) - np.array([b[0] for b in bounds])) + np.array([b[0] for b in bounds])
            fitness[i] = obj_function(nests[i])

    # Return the best solution
    best_idx = np.argmin(fitness)
    return nests[best_idx], fitness[best_idx]

# Example usage: Minimize f(x) = x^2
def objective(x):
    return sum(xi**2 for xi in x)

bounds = [(-10, 10), (-10, 10)]  # 2D problem
best_solution, best_fitness = cuckoo_search(objective, bounds)
print("Best Solution:", best_solution)
print("Best Fitness:", best_fitness)
```

Output:

```
Best Solution: [-0.14023741  0.59049343]
Best Fitness: 0.36834901994989167
```

Program 5

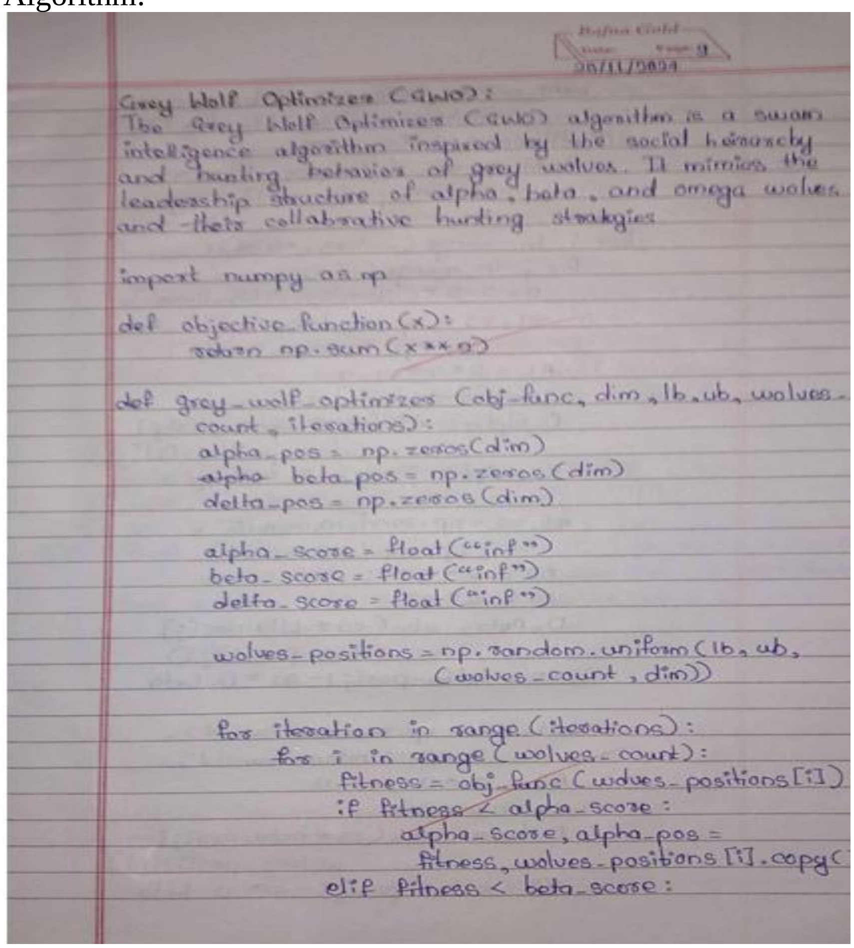
Grey Wolf Optimizer (GWO)

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:

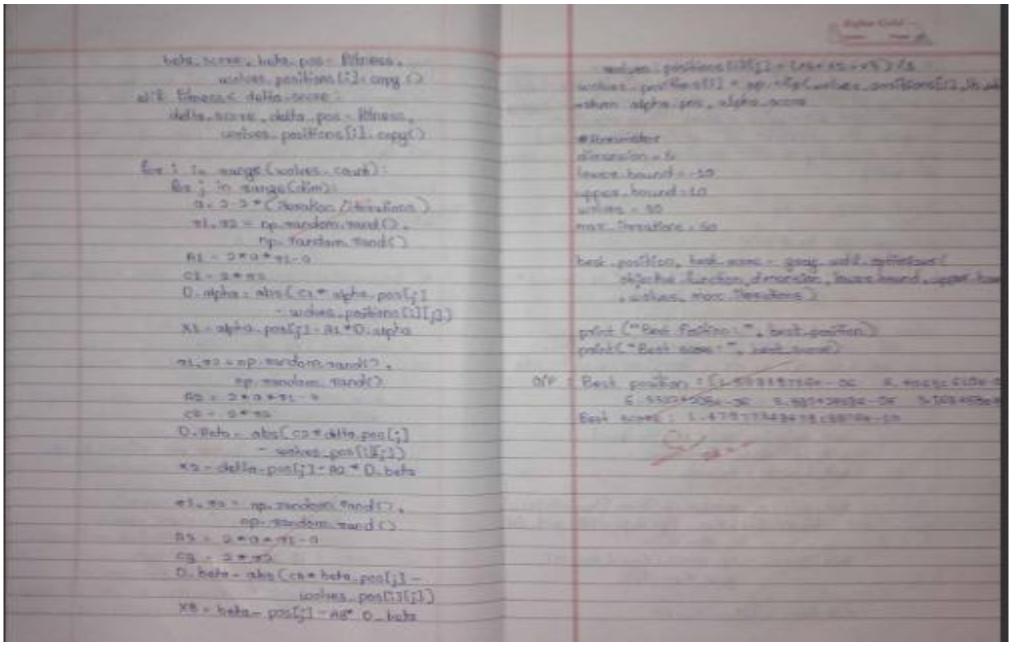


26/11/2024

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, and omega wolves and their collaborative hunting strategies

```
import numpy as np

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

def grey_wolf_optimizer(obj_func, dim, lb, ub, wolves_
    count, iterations):
    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")

    wolves_positions = np.random.uniform(lb, ub,
                       (wolves_count, dim))

    for iteration in range(iterations):
        for i in range(wolves_count):
            fitness = obj_func(wolves_positions[i])
            if fitness < alpha_score:
                alpha_score, alpha_pos =
                fitness, wolves_positions[i].copy(
            elif fitness < beta_score:
```



Code:

```
#Grey Wolf Optimizer (GWO)
import numpy as np

def objective_function(x):
    """Example objective function: Sphere function."""
    return sum(x**2)

def initialize_population(dim, n_wolves, bounds):
    """Initialize the positions of the wolves randomly within the given bounds."""
    return np.random.uniform(bounds[0], bounds[1], (n_wolves, dim))

def gwo(objective_function, bounds, dim, n_wolves, n_iterations):

    # Initialize population
    wolves = initialize_population(dim, n_wolves, bounds)
    fitness = np.apply_along_axis(objective_function, 1, wolves)

    # Initialize alpha, beta, and delta
    alpha, beta, delta = np.argsort(fitness)[:3]
    alpha_pos, alpha_score = wolves[alpha], fitness[alpha]
    beta_pos, beta_score = wolves[beta], fitness[beta]
    delta_pos, delta_score = wolves[delta], fitness[delta]

    # Main optimization loop
    for iteration in range(n_iterations):
        a = 2 - 2 * (iteration / n_iterations)  # Linearly decreasing a

        for i in range(n_wolves):
            for j in range(dim):
```

```python
            # Update each wolf's position
            r1, r2 = np.random.rand(), np.random.rand()
            A1, C1 = 2 * a * r1 - a, 2 * r2
            D_alpha = abs(C1 * alpha_pos[j] - wolves[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] - wolves[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] - wolves[i, j])
            X3 = delta_pos[j] - A3 * D_delta

            # Average position update
            wolves[i, j] = (X1 + X2 + X3) / 3.0

        # Enforce bounds
        wolves[i, :] = np.clip(wolves[i, :], bounds[0], bounds[1])

    # Evaluate fitness and update alpha, beta, delta
    fitness = np.apply_along_axis(objective_function, 1, wolves)
    sorted_indices = np.argsort(fitness)
    alpha, beta, delta = sorted_indices[:3]
    alpha_pos, alpha_score = wolves[alpha], fitness[alpha]
    beta_pos, beta_score = wolves[beta], fitness[beta]
    delta_pos, delta_score = wolves[delta], fitness[delta]

return alpha_pos, alpha_score

# Example usage
dim = 5  # Number of dimensions
bounds = (-10, 10)  # Search space bounds
n_wolves = 30  # Number of wolves
n_iterations = 100  # Number of iterations

best_solution, best_score = gwo(objective_function, bounds, dim, n_wolves, n_iterations)
print(f"Best solution: {best_solution}")
print(f"Best score: {best_score}")
```

Output:

```
Best solution: [-1.48263895e-11 -1.24732979e-11  1.51277899e-11  1.54330567e-11
  1.16834722e-11]
Best score: 9.78937775690888e-22
```

Program 6

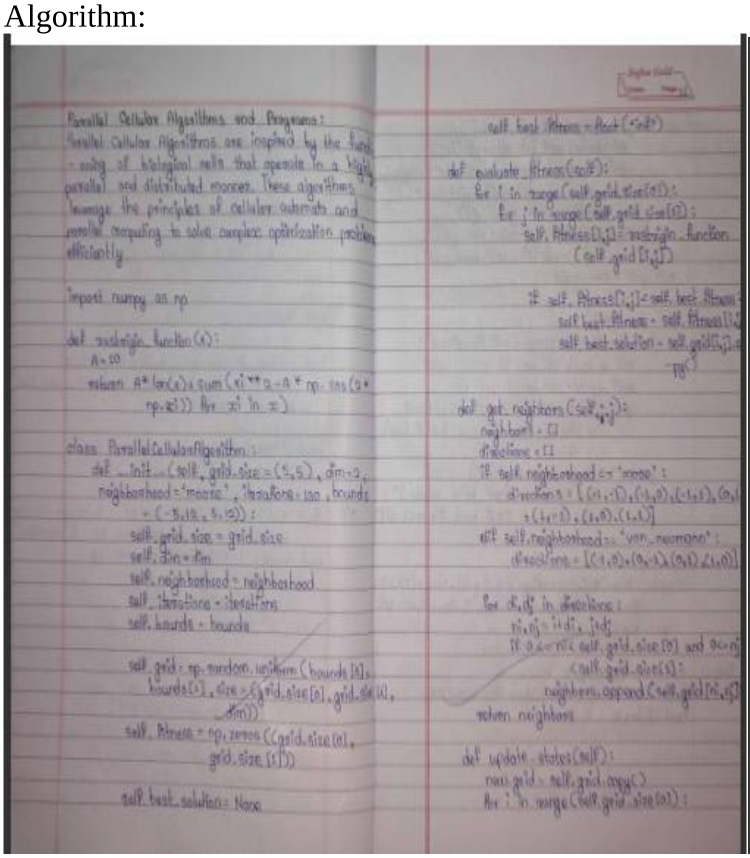
Parallel Cellular Algorithms and Programs

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:

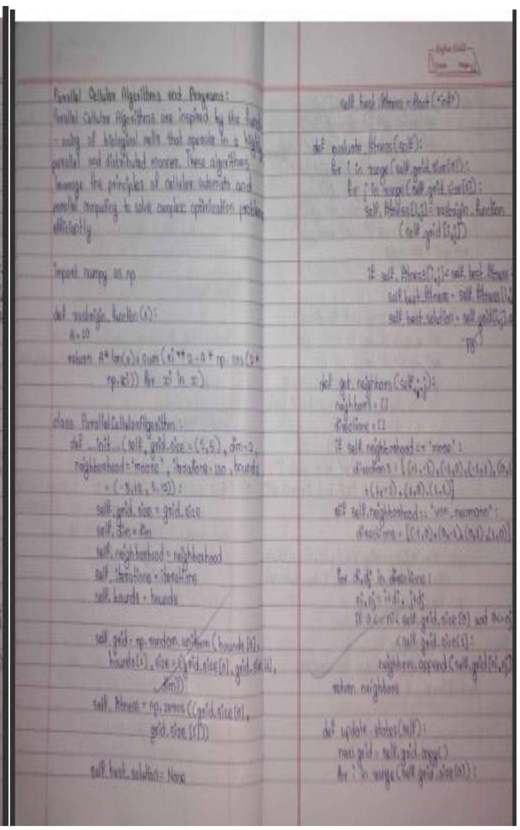
1. Define the Problem: Create a mathematical function to optimize.
2. Initialize Parameters: Set the number of cells, grid size, neighborhood structure, and number of iterations.
3. Initialize Population: Generate an initial population of cells with random positions in the solution space.
4. Evaluate 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:



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.

```
import numpy as np

def rastrigin_function(x):
    A = 10
    return A * len(x) + sum(xi**2 - A * np.cos(2 *
            np.pi * xi) for xi in x)

class ParallelCellularAlgorithm:
    def __init__(self, grid_size=(5,5), dim=2,
      neighborhood='moore', iterations=100, bounds
        =(-5.12, 5.12)):
        self.grid_size = grid_size
        self.dim = dim
        self.neighborhood = neighborhood
        self.iterations = iterations
        self.bounds = bounds

        self.grid = np.random.uniform(bounds[0],
          bounds[1], size=(grid_size[0], grid_size[1],
             dim))
        self.fitness = np.zeros((grid_size[0],
                grid_size[1]))

        self.best_solution = None
        self.best_fitness = float('inf')

    def evaluate_fitness(self):
        for i in range(self.grid_size[0]):
            for j in range(self.grid_size[1]):
                self.fitness[i,j] = rastrigin_function
                    (self.grid[i,j])

                if self.fitness[i,j] < self.best_fitness:
                    self.best_fitness = self.fitness[i,j]
                    self.best_solution = self.grid[i,j].co
                      py()

    def get_neighbors(self, i, j):
        neighbors = []
        directions = []
        if self.neighborhood == 'moore':
            directions = [(-1,-1),(-1,0),(-1,1),(0,
              ...(1,-1),(1,0),(1,1)]
        elif self.neighborhood == 'von_neumann':
            directions = [(-1,0),(0,-1),(0,1),(1,0)]

        for di, dj in directions:
            ni, nj = i+di, j+dj
            if 0 <= ni < self.grid_size[0] and 0 <= nj
                < self.grid_size[1]:
                neighbors.append(self.grid[ni, nj])
        return neighbors

    def update_states(self):
        new_grid = self.grid.copy()
        for i in range(self.grid_size[0]):
```



Code:

```
#Parallel Cellular Algorithms and Programs
import numpy as np

# Define the optimization function
def fitness_function(x):
    return x**2

# Initialize parameters
num_cells = 10
grid_size = 1.0
iterations = 100
neighborhood_size = 1

# Initialize population
cells = np.random.uniform(-grid_size, grid_size, num_cells)

# Main loop
for _ in range(iterations):
    # Evaluate fitness
    fitness = np.array([fitness_function(cell) for cell in cells])

    # Update states
    new_cells = np.copy(cells)
    for i in range(num_cells):
        # Get neighbors
        neighbors = cells[max(0, i-neighborhood_size):min(num_cells, i+neighborhood_size+1)]
        # Update cell based on neighbors
        new_cells[i] = np.mean(neighbors) + np.random.uniform(-0.1, 0.1)  # Add some noise

    cells = new_cells

# Output the best solution
best_cell = cells[np.argmin(fitness)]
print(f"Best solution found: {best_cell}")
print(f"Fitness: {fitness_function(best_cell)}")
```

Output:

```
Best solution found: -0.11165744078455692
Fitness: 0.012467384082556834
```

Program 7

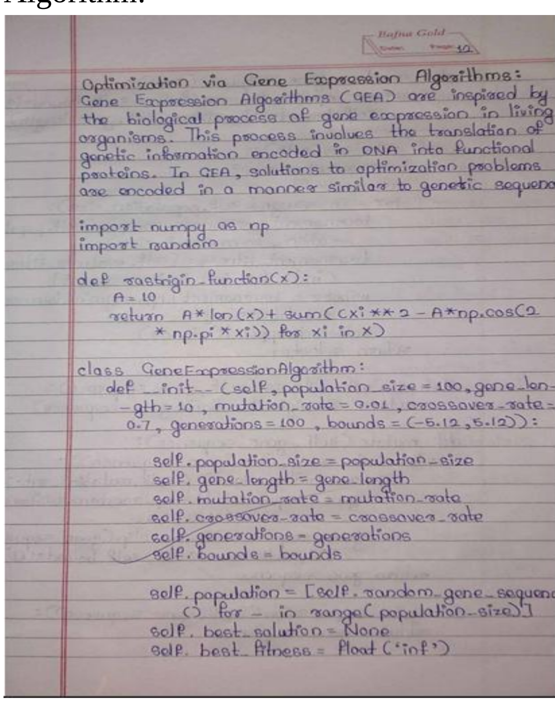
Optimization via Gene Expression Algorithms

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:

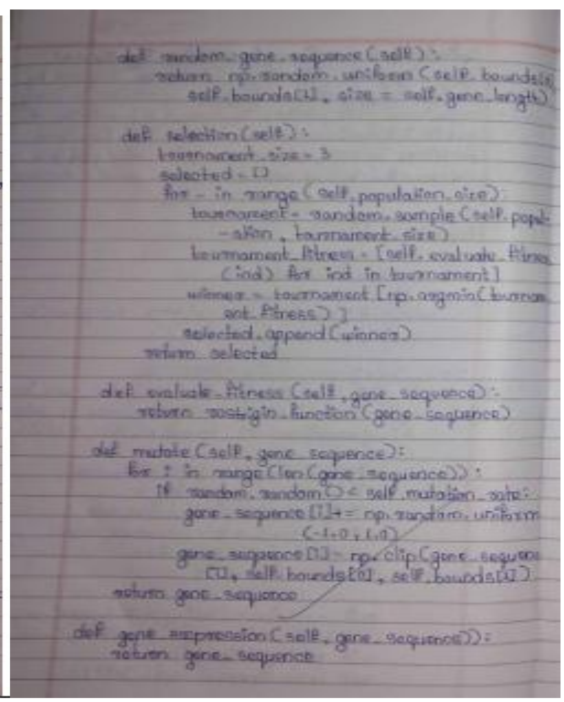
1. Define the Problem: Create a mathematical function to optimize.
2. Initialize Parameters: Set the population size, number of genes, mutation rate, crossover rate, and number of generations.
3. Initialize Population: Generate an initial population of random genetic sequences.
4. Evaluate Fitness: Evaluate 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:

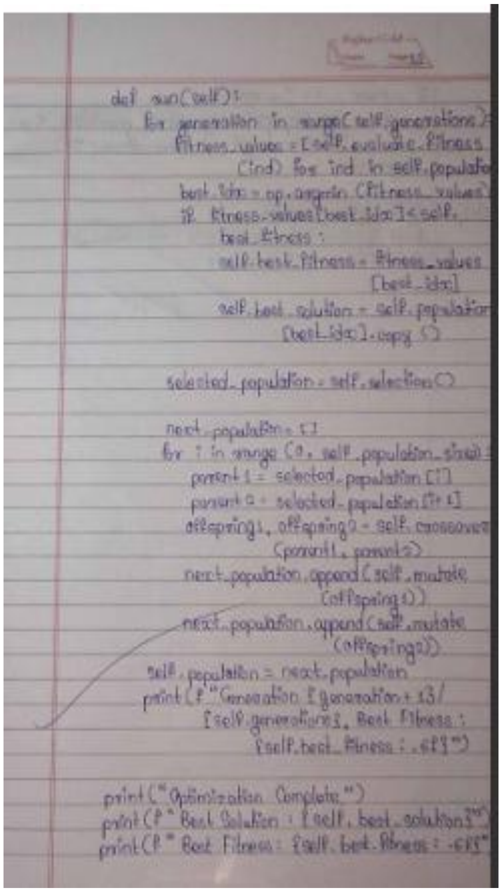


Optimization via Gene Expression Algorithms:
Gene Expression Algorithms (GEA) are inspired by the biological process of gene expression in living organisms. This process involves the translation of genetic information encoded in DNA into functional proteins. In GEA, solutions to optimization problems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

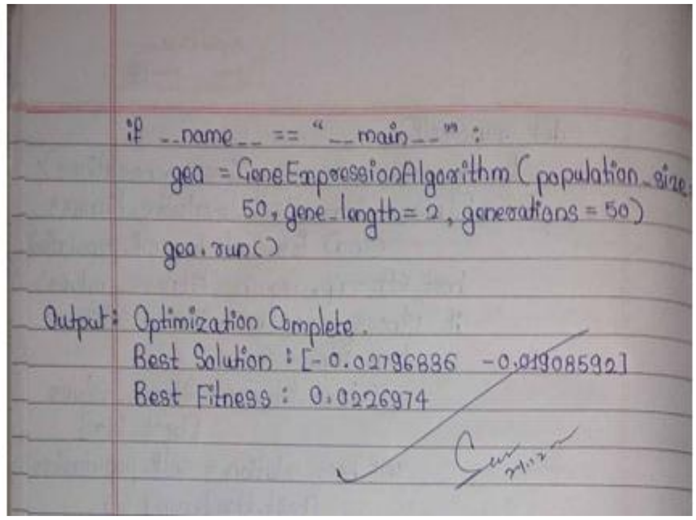
```
import numpy as np
import random

def rastrigin_function(x):
    A = 10
    return A * len(x) + sum((xi ** 2 - A * np.cos(2
        * np.pi * xi)) for xi in x)

class GeneExpressionAlgorithm:
    def __init__(self, population_size = 100, gene_len-
        -gth = 10, mutation_rate = 0.01, crossover_rate =
        0.7, generations = 100, bounds = (-5.12, 5.12)):

        self.population_size = population_size
        self.gene_length = gene_length
        self.mutation_rate = mutation_rate
        self.crossover_rate = crossover_rate
        self.generations = generations
        self.bounds = bounds

        self.population = [self.random_gene_sequenc
            () for _ in range(population_size)]
        self.best_solution = None
        self.best_fitness = float('inf')
```



```
    def random_gene_sequence(self):
        return np.random.uniform(self.bounds[0],
            self.bounds[1], size = self.gene_length)

    def selection(self):
        tournament_size = 3
        selected = []
        for _ in range(self.population_size):
            tournament = random.sample(self.popul
                -ation, tournament_size)
            tournament_fitness = [self.evaluate_fitnes
                (ind) for ind in tournament]
            winner = tournament[np.argmin(tournam
                ent_fitness)]
            selected.append(winner)
        return selected

    def evaluate_fitness(self, gene_sequence):
        return rastrigin_function(gene_sequence)

    def mutate(self, gene_sequence):
        for i in range(len(gene_sequence)):
            if random.random() < self.mutation_rate:
                gene_sequence[i] += np.random.uniform
                    (-1.0, 1.0)
                gene_sequence[i] = np.clip(gene_sequen
                    [i], self.bounds[0], self.bounds[1])
        return gene_sequence

    def gene_expression(self, gene_sequence)):
        return gene_sequence
```



```
def run(self):
    for generation in range(self.generations):
        fitness_values = [self.evaluate_fitness
                (ind) for ind in self.populatio
        best_idx = np.argmin(fitness_values)
        if fitness_values[best_idx] < self.
            best_fitness:
            self.best_fitness = fitness_values
                            [best_idx]
            self.best_solution = self.populatio
                        [best_idx].copy()

        selected_population = self.selection()

        next_population = []
        for i in range(0, self.population_size
            parent1 = selected_population[i]
            parent2 = selected_population[i+1]
            offspring1, offspring2 = self.crossove
                        (parent1, parent2)
            next_population.append(self.mutate
                            (offspring1))
            next_population.append(self.mutate
                            (offspring2))
        self.population = next_population
        print(f"Generation {generation+1}/
            {self.generations}, Best Fitness :
                {self.best_fitness:.6f}")

    print("Optimization Complete.")
    print(f" Best Solution : {self.best_solution}")
    print(f" Best Fitness: {self.best_fitness:.6f}"
```



```
if __name__ == "__main__":
    gea = GeneExpressionAlgorithm(population_size=
            50, gene_length=2, generations=50)
    gea.run()
```

Output: Optimization Complete.
Best Solution : [-0.02796836 -0.01908592]
Best Fitness : 0.0226974

Code:
#Optimization via Gene Expression Algorithms

```
import numpy as np

# Define the optimization function
def fitness_function(x):
    return x**2

# Convert binary string to decimal
def binary_to_decimal(binary_str):
    return int(binary_str, 2) / (2**len(binary_str) - 1) * 10 - 5  # Scale to [-5, 5]

# Initialize parameters
population_size = 20
num_genes = 10
mutation_rate = 0.1
crossover_rate = 0.7
generations = 100

# Initialize population
population = [''.join(np.random.choice(['0', '1'], num_genes)) for _ in range(population_size)]
```

```
# Main loop
for _ in range(generations):
    # Evaluate fitness
    fitness = [fitness_function(binary_to_decimal(ind)) for ind in population]

    # Selection (roulette wheel)
    total_fitness = sum(fitness)
    probabilities = [f / total_fitness for f in fitness]
    selected = np.random.choice(population, size=population_size, p=probabilities)

    # Crossover
    offspring = []
    for i in range(0, population_size, 2):
        if np.random.rand() < crossover_rate:
            point = np.random.randint(1, num_genes)
            offspring.append(selected[i][:point] + selected[i+1][point:])
            offspring.append(selected[i+1][:point] + selected[i][point:])
        else:
            offspring.append(selected[i])
            offspring.append(selected[i+1])

    # Mutation
    for i in range(population_size):
        if np.random.rand() < mutation_rate:
            point = np.random.randint(num_genes)
            offspring[i] = offspring[i][:point] + ('1' if offspring[i][point] == '0' else '0') +
offspring[i][point+1:]

    population = offspring

# Output the best solution
best_individual = min(population, key=lambda ind: fitness_function(binary_to_decimal(ind)))
best_fitness = fitness_function(binary_to_decimal(best_individual))
print(f"Best solution found: {binary_to_decimal(best_individual)}")
print(f" Fitness: {best_fitness}")
```

Output:

```
Best solution found: -4.872922776148583
 Fitness: 23.74537638230761
```