VISVESVARAYA TECHNOLOGICAL UNIVERSITY

"JnanaSangama", Belgaum -590014, Karnataka.

LAB RECORD

Bio Inspired Systems (23CS5BSBIS)

Submitted by

Shreya Soni(1BM22CS268)

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 Soni (1BM22CS268),** 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.

Spoorthi D M Assistant Professor Department of CSE, BMSCE	Dr. Kavitha Soodha Professor & HOD Department of CSE, BMSCE

Index

Github Link:
https://github.com/SHREYASONI28/BIS-Lab

Program 1

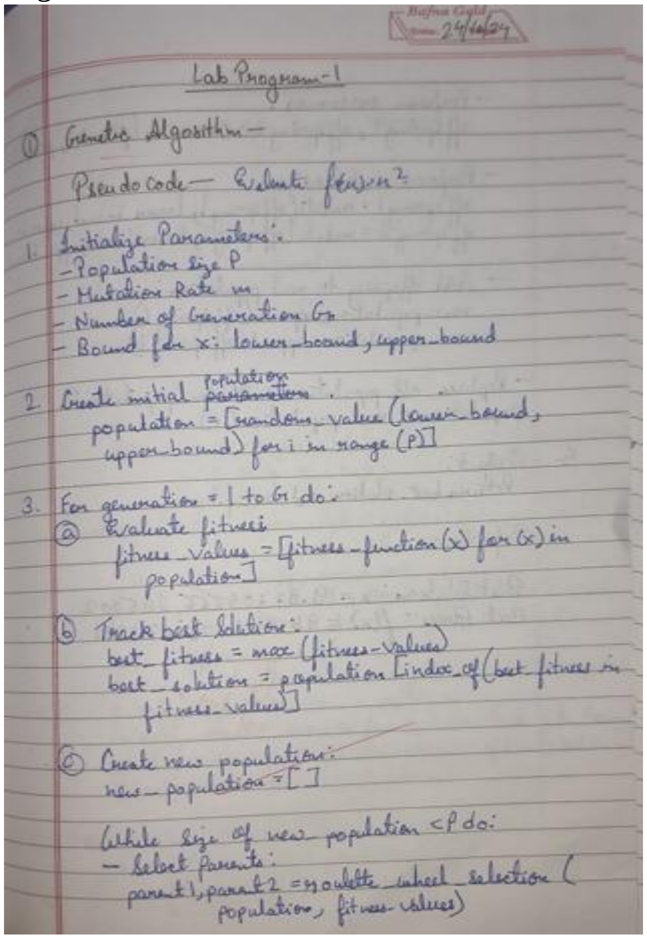
Genetic Algorithm for Optimization Problems

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:



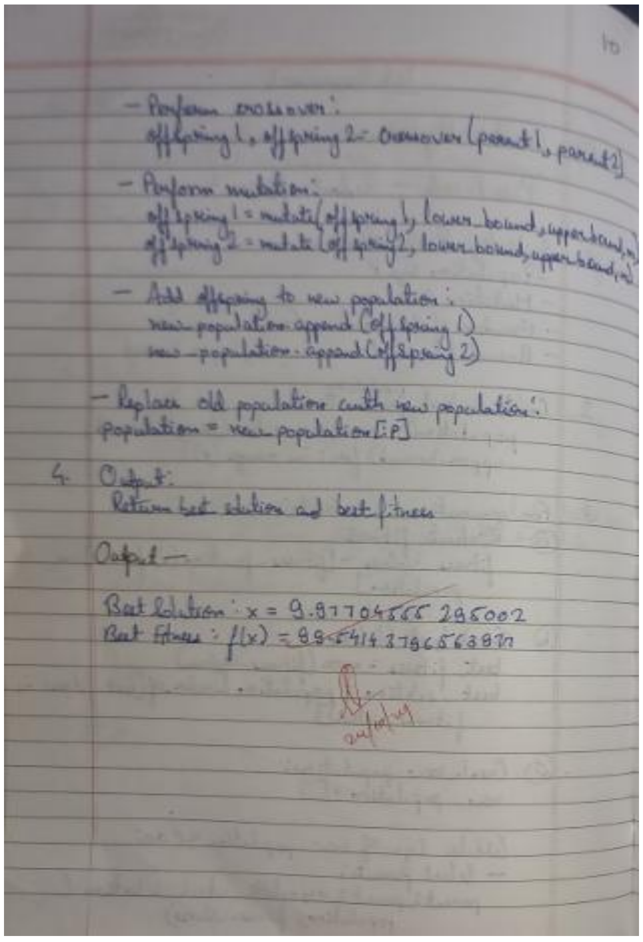
Date 24/10/24

Lab Program-1

① Genetic Algorithm—

Pseudo code— Evaluate $f(x)=x^2$

```
1. Initialize Parameters:
   - Population size P
   - Mutation Rate m
   - Number of Generation G
   - Bound for x: lower_bound, upper_bound

2. Create initial ~~parameters~~ population:
   population = [random_value(lower_bound,
       upper_bound) for i in range (P)]

3. For generation = 1 to G do:
   (a) Evaluate fitness
       fitness_values = [fitness_function(x) for (x) in
           population]

   (b) Track best Solution:
       best_fitness = max (fitness_values)
       best_solution = population [index_of(best_fitness in
           fitness_values)]

   (c) Create new population:
       new_population = [ ]

       While size of new population < P do:
       - Select Parents:
         parent1, parent2 = roulette_wheel_selection (
             population, fitness_values)
```



10

```
       - Perform crossover:
         offspring1, offspring2 = crossover (parent1, parent2)
       - Perform mutation:
         offspring1 = mutate(offspring1, lower_bound, upper_bound, m)
         offspring2 = mutate (offspring2, lower_bound, upper_bound, m)
       - Add offspring to new population:
         new_population.append (offspring1)
         new_population.append(offspring2)
     - Replace old population with new population:
       population = new_population[:P]

4. Output:
   Return best solution and best fitness
```

Output—

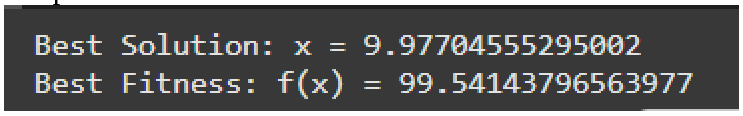
```
Best Solution: x = 9.97704565 295002
Best Fitness: f(x) = 99.54143796563897
```

[illegible] 24/10/24

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)
    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

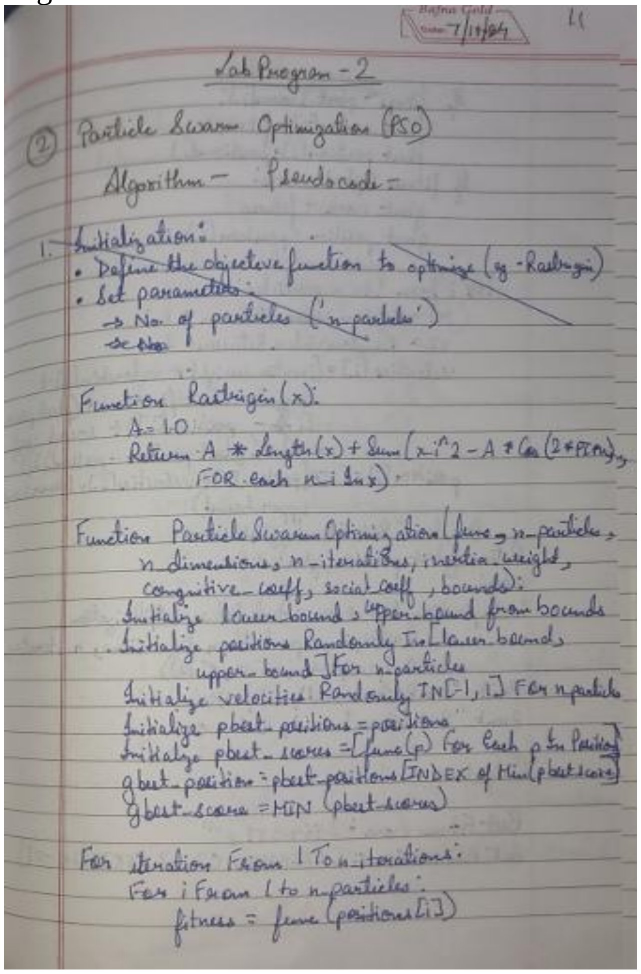
Particle Swarm Optimization for Function Optimization

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:



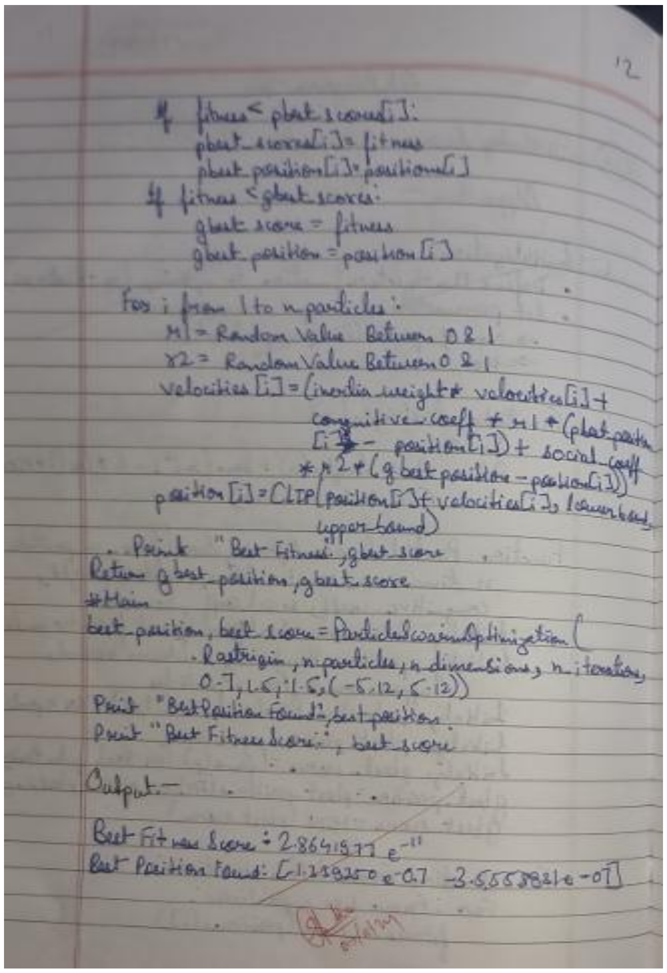
Bafna Gold
Date 7/11/24 11

Lab Program - 2

(2) Particle Swarm Optimization (PSO)

Algorithm - Pseudocode -

```
1. ~~Initialization:~~
   ~~• Define the objective function to optimize (eg - Rastrigin)~~
   ~~• Set parameters:~~
     ~~→ No. of particles ('n_particles')~~
     ~~Se [illegible]~~

Function Rastrigin(x):
    A = 10
    Return A * Length(x) + Sum(x_i^2 - A * Cos(2*PI*x_i),
           FOR each x_i In x)

Function ParticleSwarmOptimization(func, n_particles,
        n_dimensions, n_iterations, inertia_weight,
        congnitive_coeff, social_coeff, bounds):
    Initialize lower_bound, upper_bound from bounds
    Initialize positions Randomly In [lower_bound,
            upper_bound] For n_particles
    Initialize velocities Randomly IN [-1, 1] For n_particles
    Initialize pbest_positions = positions
    Initialize pbest_scores = [func(p) For Each p In Positions]
    gbest_position = pbest_positions [INDEX of Min(pbest score)]
    gbest_score = MIN (pbest_scores)

For iteration From 1 To n_iterations:
    For i From 1 to n_particles:
        fitness = func(positions[i])
```



12

```
        If fitness < pbest scores[i]:
            pbest_scores[i] = fitness
            pbest position[i] = position[i]
        If fitness < gbest scores:
            gbest score = fitness
            gbest_position = position[i]

    For i from 1 to n_particles:
        r1 = Random Value Between 0 & 1
        r2 = Random Value Between 0 & 1
        velocities [i] = (inertia_weight * velocities[i] +
                congnitive_coeff * r1 * (pbest_position
                [i] - position[i]) + social_coeff
                * r2 * (gbest position - position[i]))
        position [i] = CLIP(position[i] + velocities[i], lower bound,
                upper_bound)
    Print "Best Fitness", gbest_score
Return gbest_position, gbest score
#Main
best_position, best_score = ParticleSwarmOptimization(
        Rastrigin, n_particles, n_dimensions, n_iterations,
        0.7, 1.5, 1.5, (-5.12, 5.12))
Print "Best Position Found:", best_position
Print "Best Fitness Score:", best_score
```

Output:-

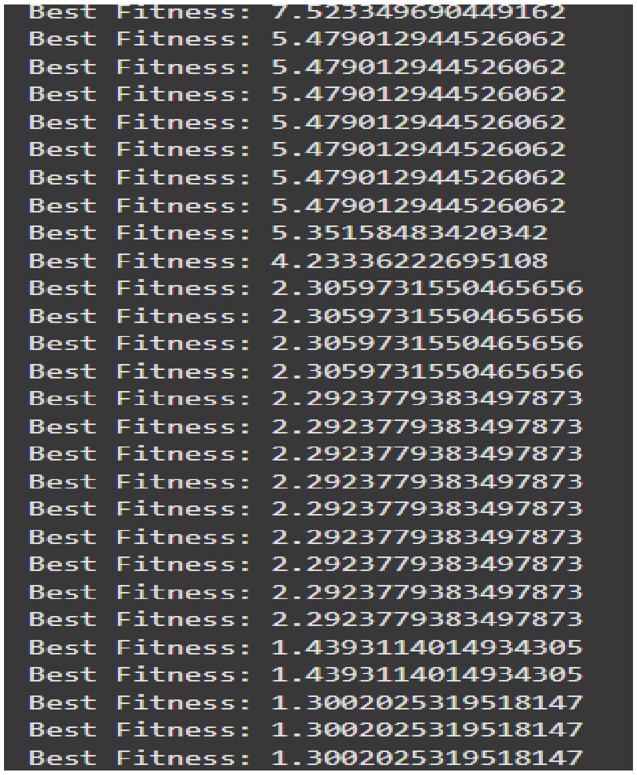
```
Best Fitness Score : 2.86419777 e^-11
Best Position Found: [-1.239250 e-07 -3.5553831e-07]
```

[illegible]

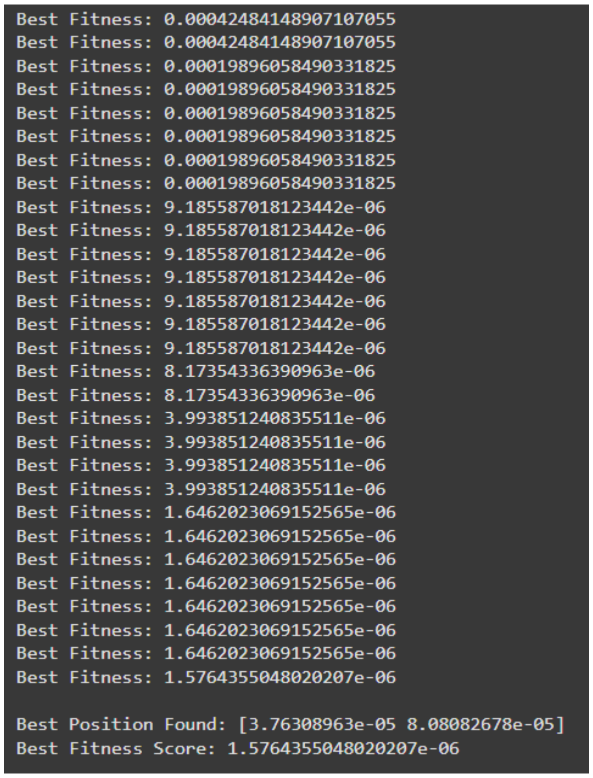
Code:

```
#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:



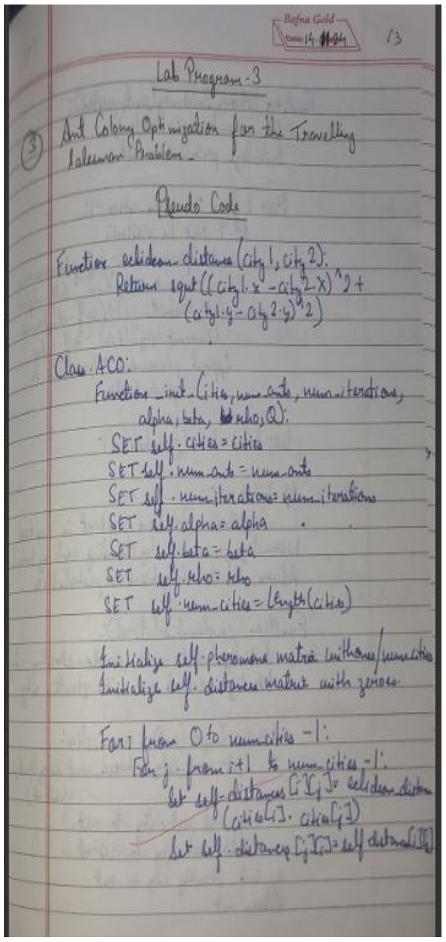
```
Best Fitness: 7.523349690449162
Best Fitness: 5.479012944526062
Best Fitness: 5.479012944526062
Best Fitness: 5.479012944526062
Best Fitness: 5.479012944526062
Best Fitness: 5.479012944526062
Best Fitness: 5.479012944526062
Best Fitness: 5.479012944526062
Best Fitness: 5.35158483420342
Best Fitness: 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
```

Program 3
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:



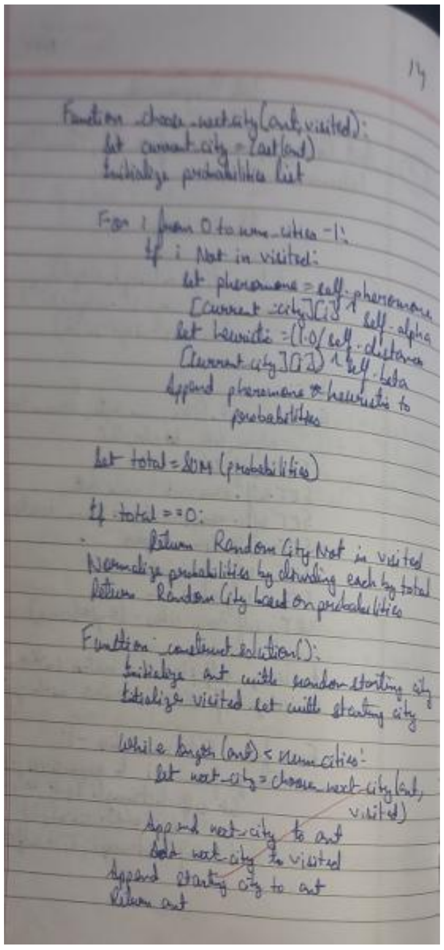
Bafna Gold
Date: 14-11-24 13

Lab Program-3

(3) Ant Colony Optimization for the Travelling Salesman Problem.

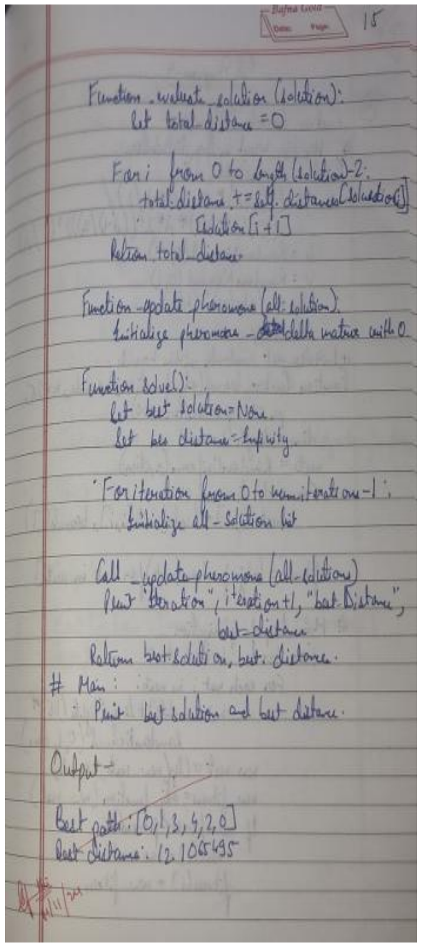
Pseudo Code

```
Function eclidean_distance(city1, city2):
        Return sqrt((city1.x - city2.x)^2 +
               (city1.y - city2.y)^2)

Class ACO:
    Function _init_(cities, num_ants, num_iterations,
            alpha, beta, rho, Q):
        SET self.cities = cities
        SET self.num_ants = num_ants
        SET self.num_iterations = num_iterations
        SET self.alpha = alpha
        SET self.beta = beta
        SET self.rho = rho
        SET self.num_cities = Length(cities)

        Initialize self.pheromone matrix with ones/num_cities
        Initialize self.distances matrix with zeroes

        For i from 0 to num_cities - 1:
            For j from i+1 to num_cities - 1:
                let self.distances[i][j] = eclidean_distance
                      (cities[i], cities[j])
                Set self.distances[j][i] = self.distances[i][j]
```



14

```
Function _choose_next_city(ant, visited):
    let current_city = last(ant)
    Initialize probabilities list

    For i from 0 to num_cities - 1:
        If i Not in visited:
            let pheromone = self.pheromone
              [current_city][i] ^ self.alpha
            let heuristic = (1.0 / self.distances
              [current_city][i]) ^ self.beta
            Append pheromone * heuristic to
                  probabilities

    let total = SUM (probabilities)

    If total == 0:
          Return Random City Not in visited
    Normalize probabilities by dividing each by total
    Return Random City based on probabilities

Function construct_solution():
    Initialize ant with random starting city
    Initialize visited set with starting city

    While length(ant) < num_cities:
        let next_city = choose_next_city(ant,
                                   visited)

        Append next_city to ant
        Add next_city to visited
    Append starting city to ant
    Return ant
```



Bafna Gold
Date: Page: 15

```
Function _evaluate_solution(solution):
    let total_distance = 0

    For i from 0 to length(solution)-2:
        total_distance += self.distances[solution[i]]
               [solution[i+1]]
    Return total_distance

Function _update_pheromone(all_solution):
    Initialize pheromone_delta matrix with 0

Function solve():
    let best solution = None
    let best distance = Infinity

    For iteration from 0 to num_iterations - 1:
        Initialize all_solution list

    Call _update_pheromone(all_solutions)
    Print "Iteration", iteration+1, "best Distance",
                best_distance
    Return best_solution, best_distance
# Main:
    Print best solution and best distance.
```

Output -

Best path: [0, 1, 3, 4, 2, 0]
Best Distance: 12.1065495

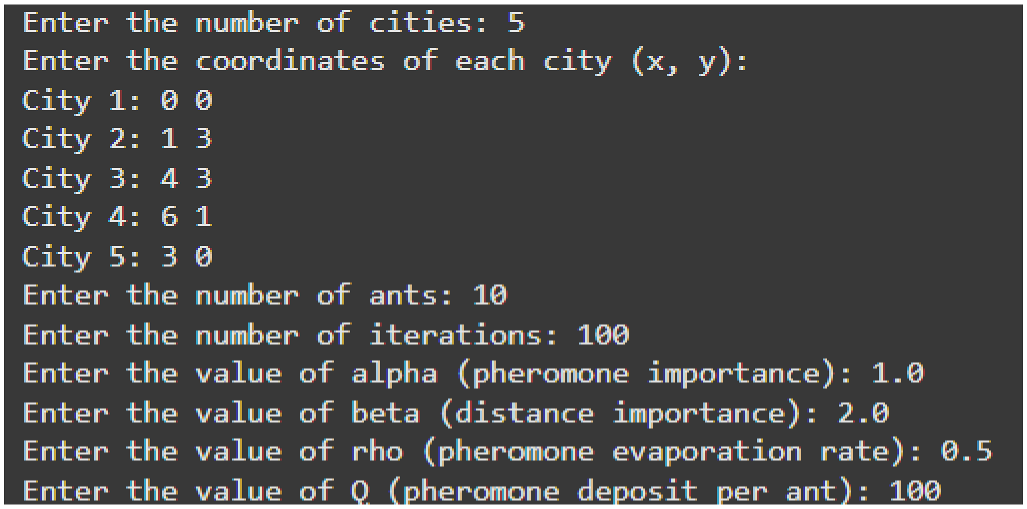
[illegible] 14/11/24

Code:

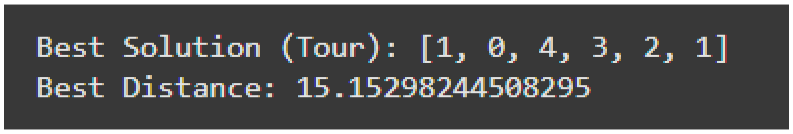
```
#ANT COLONY OPTIMIZATION

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

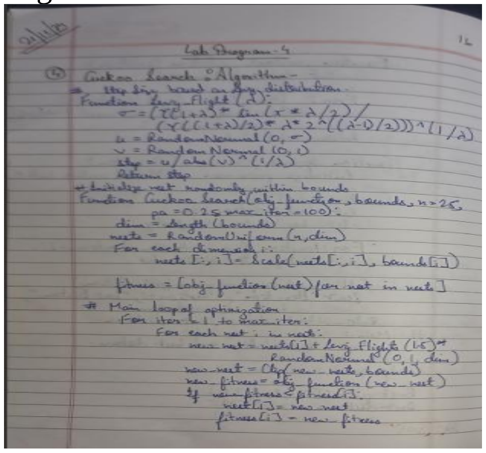
Cuckoo Search (CS)

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.

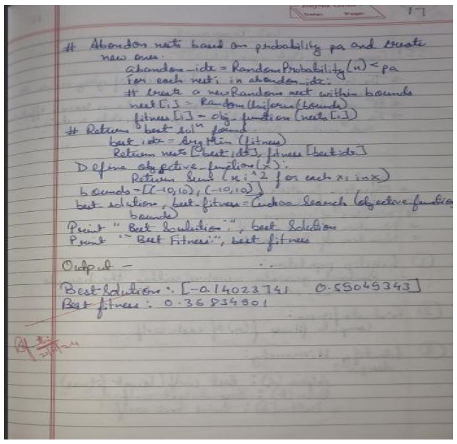
Implementation Steps:

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:



```
21/11/24                                                    12
                         Lab Program - 4
(4) Cuckoo Search : Algorithm -
# Step Size based on Levy distribution
Function Levy_Flight (λ):
      σ = (Γ(1+λ) * sin(π * λ/2) /
           (Γ((1+λ)/2) * λ * 2^((λ-1)/2)))^(1/λ)
      u = RandomNormal (0, σ)
      v = Random Normal (0, 1)
      step = u / abs(v)^(1/λ)
      Return step
# Initialize nest randomly within bounds
Function Cuckoo Search(obj_function, bounds, n=25,
           pa = 0.25 max_iter = 100):
      dim = Length (bounds)
      nests = RandomUniform (n, dim)
      For each dimension i:
           nests [:, i] = Scale( nests[:, i], bounds[i])

      fitness = [obj_function (nest) for nest in nests]

 # Main loop of optimization
      For iter = 1 to max_iter:
           For each nest i in nests:
                new_nest = nests[i] + Levy_Flight (1.5) *
                               RandomNormal (0, 1, dim)
                new_nest = Clip( new_nests, bounds)
                new_fitness = obj_function (new_nest)
                If new_fitness < fitness[i]:
                     nest[i] = new_nest
                     fitness[i] = new_fitness
```



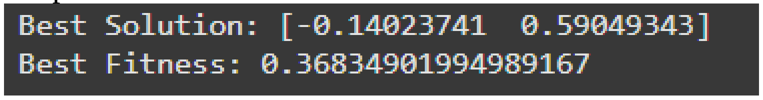
```
                                                            17
# Abandon nests based on probability pa and create
  new ones.
      abandon_idx = RandomProbability(n) < pa
      For each nest i in abandon_idx:
      # Create a new Random nest within bounds
      nest[i] = RandomUniform(bounds)
      fitness[i] = obj_function (nests[i])
# Return "best sol^n found.
      best_idx = ArgMin (fitness)
      Return nests [best_idx], fitness [best_idx]
 Define objective_function(x):
      Return Sum (xi^2 for each xi in x)
 bounds = [(-10,10), (-10,10)]
 best_solution, best_fitness = Cuckoo Search (objective_function,
      bounds)
Print " Best Solution :", best Solution
Print " Best Fitness :", best fitness

Output -

Best Solution : [-0.14023741    0.55049343]
Best fitness : 0.36834901
```

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

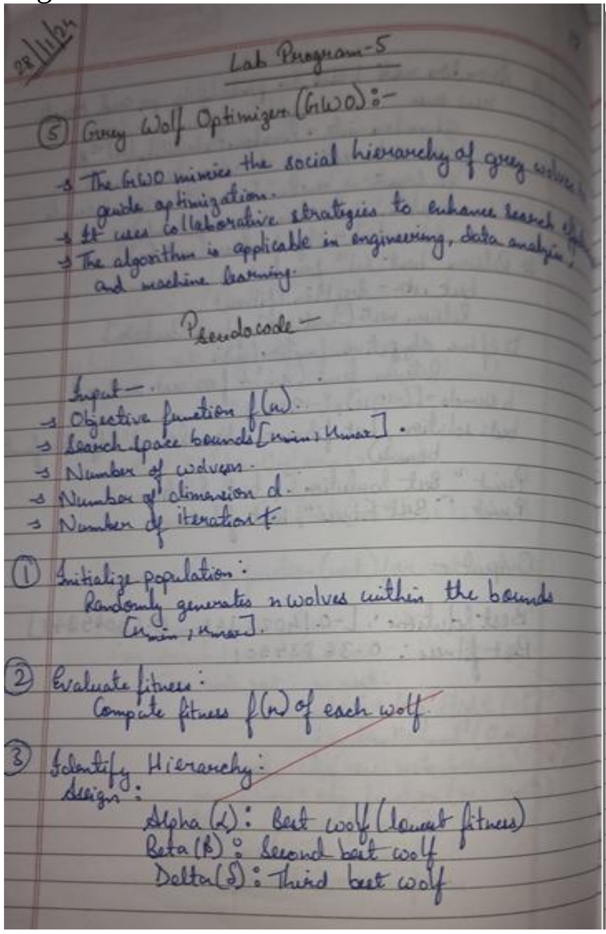
Grey Wolf Optimizer (GWO)

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:



28/11/24

Lab Program-5

⑤ Grey Wolf Optimizer (GWO):-

→ The GWO mimics the social hierarchy of grey wolves to guide optimization.
→ It uses collaborative strategies to enhance search eff
→ The algorithm is applicable in engineering, data analysis, and machine learning.

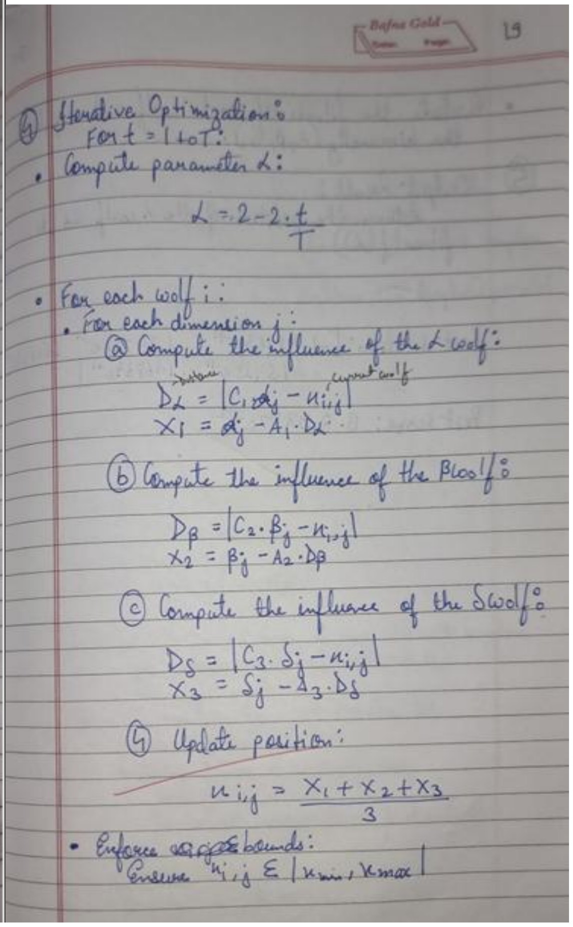
Pseudocode—

Input—
→ Objective function $f(x)$.
→ Search space bounds $[x_{min}, x_{max}]$.
→ Number of wolves.
→ Number of dimension d.
→ Number of iteration T.

① Initialize population:
Randomly generates n wolves within the bounds $[x_{min}, x_{max}]$.

② Evaluate fitness:
Compute fitness $f(x)$ of each wolf.

③ Identify Hierarchy:
Assign:
Alpha (α): Best wolf (lowest fitness)
Beta (β): Second best wolf
Delta (δ): Third best wolf



19

④ Iterative Optimization:
For $t = 1$ to T:
• Compute parameter α:

$$\alpha = 2 - 2 \cdot \frac{t}{T}$$

• For each wolf i:
• For each dimension j:
ⓐ Compute the influence of the α wolf:

$$\overset{\text{distance}}{D_\alpha} = |C_1 \cdot \alpha_j - \overset{\text{current wolf}}{x_{i,j}}|$$
$$X_1 = \alpha_j - A_1 \cdot D_\alpha$$

ⓑ Compute the influence of the β wolf:

$$D_\beta = |C_2 \cdot \beta_j - x_{i,j}|$$
$$X_2 = \beta_j - A_2 \cdot D_\beta$$

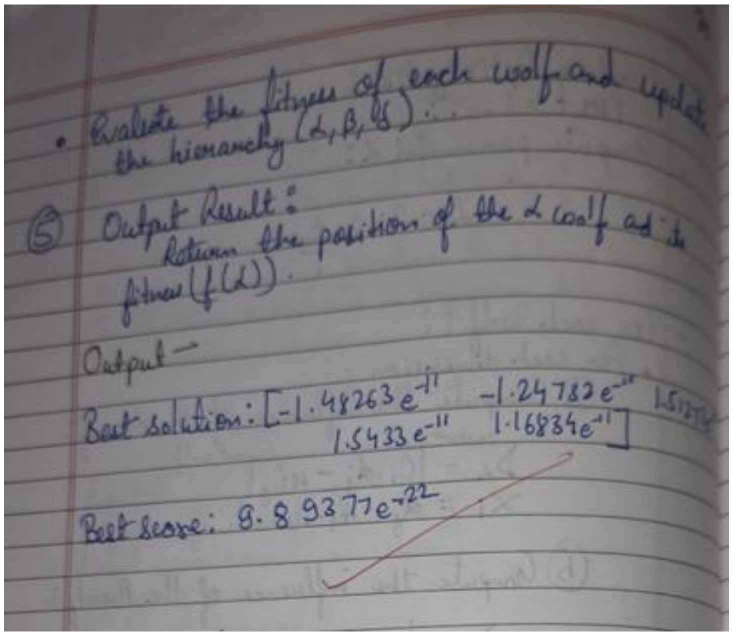
ⓒ Compute the influence of the δ wolf:

$$D_\delta = |C_3 \cdot \delta_j - x_{i,j}|$$
$$X_3 = \delta_j - A_3 \cdot D_\delta$$

④ Update position:

$$x_{i,j} = \frac{X_1 + X_2 + X_3}{3}$$

• Enforce [illegible] bounds:
Ensure $x_{i,j} \in |x_{min}, x_{max}|$



- Evaluate the fitness of each wolf and update the hierarchy (α, β, γ).

⑤ Output Result:
Return the position of the α wolf and its fitness $(f(\alpha))$.

Output →

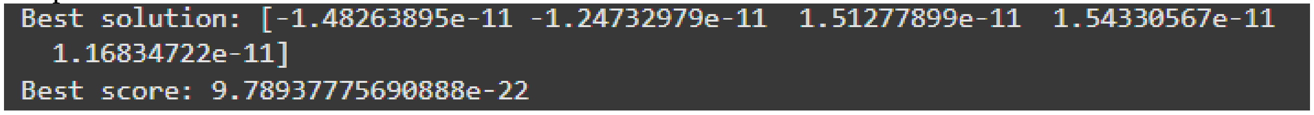
Best solution: [-1.48263e^{-11} -1.24782e^{-11} 1.512
1.5433e^{-11} 1.16834e^{-11}]

Best score: 8.89377e^{-22}

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):
                # 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}")
```

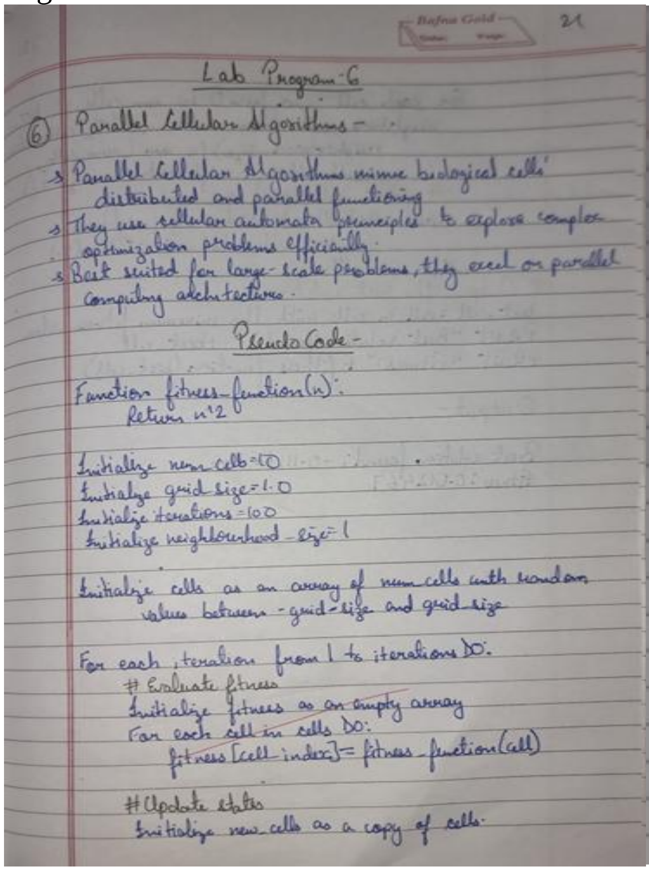
Output:



```
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:



Bafna Gold 21

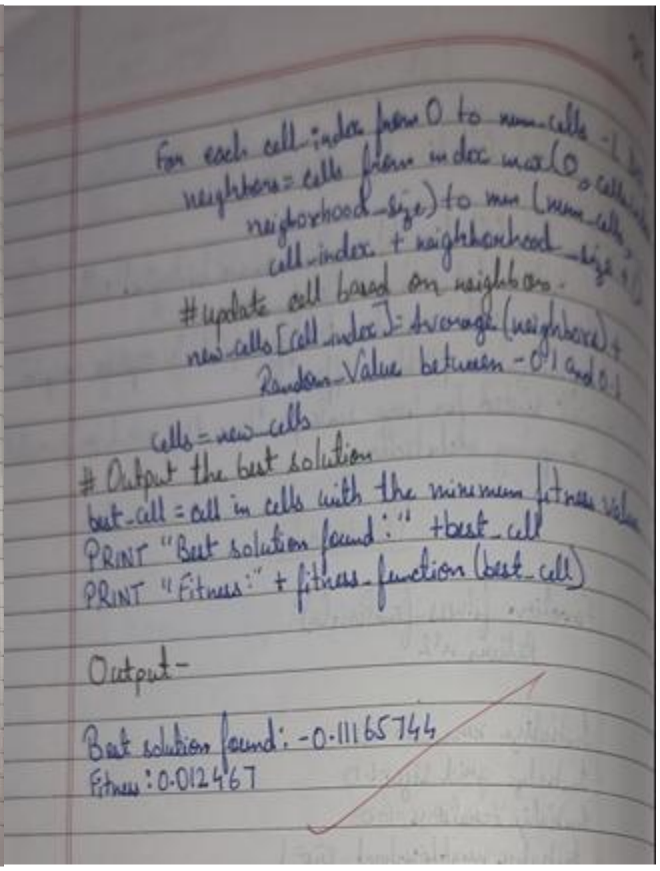
Lab Program-6

(6) Parallel Cellular Algorithms -

- Parallel Cellular Algorithms mimic biological cells' distributed and parallel functioning
- They use cellular automata principles to explore complex optimization problems efficiently.
- Best suited for large-scale problems, they excel on parallel computing architectures.

Pseudo Code -

```
Function fitness_function(x):
    Return x^2

Initialize num_cells=10
Initialize grid_size=1.0
Initialize iterations=100
Initialize neighbourhood_size=1

Initialize cells as an array of num_cells with random
        values between -grid_size and grid_size

For each iteration from 1 to iterations DO:
    # Evaluate fitness
    Initialize fitness as an empty array
    For each cell in cells DO:
        fitness[cell_index] = fitness_function(cell)

    # Update states
    Initialize new_cells as a copy of cells.
```



```
        For each cell index from 0 to num_cells -1
            neighbors = cells from index max(0, cell
                neighborhood_size) to min(num_cell
                cell_index + neighborhood_size + 1)
            # update cell based on neighbors.
            new_cells[cell_index] = Average(neighbors) +
                Random_Value between -0.1 and 0.1
    cells = new_cells
# Output the best solution
best_cell = cell in cells with the minimum fitness value
PRINT "Best solution found:" +best_cell
PRINT "Fitness:" + fitness_function(best_cell)
```

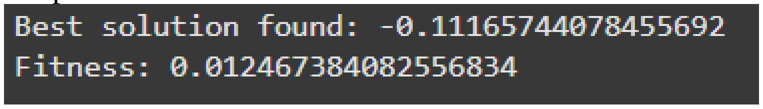
Output-

```
Best solution found: -0.1165744
Fitness: 0.012467
```

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

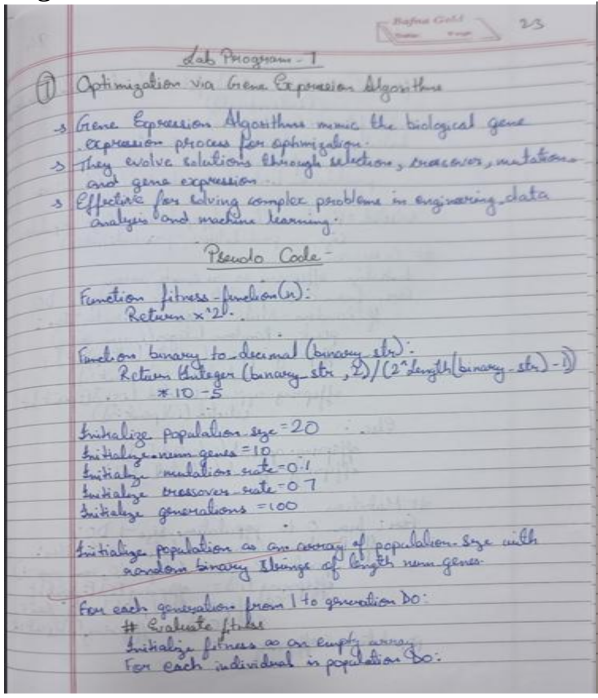
Optimization via Gene Expression Algorithms

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:



23

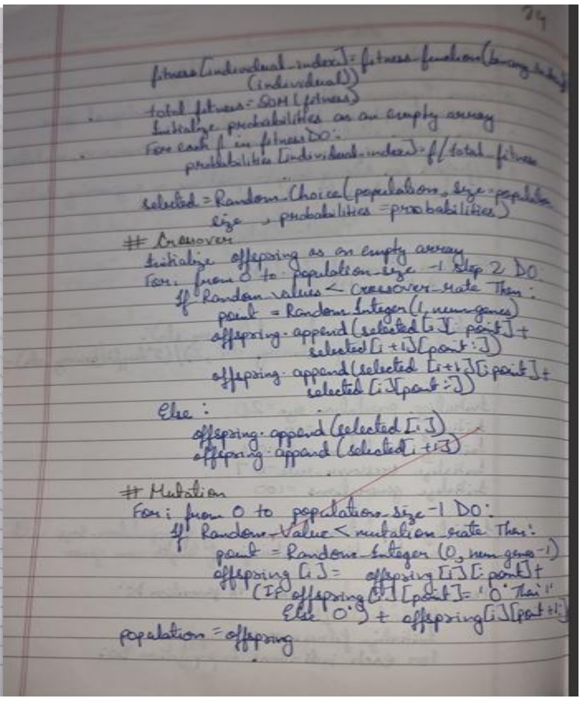
Lab Program - 7

(1) Optimization via Gene Expression Algorithms

-> Gene Expression Algorithms mimic the biological gene expression process for optimization.
-> They evolve solutions through selection, crossover, mutations and gene expression.
-> Effective for solving complex problems in engineering, data analysis and machine learning.

Pseudo Code -

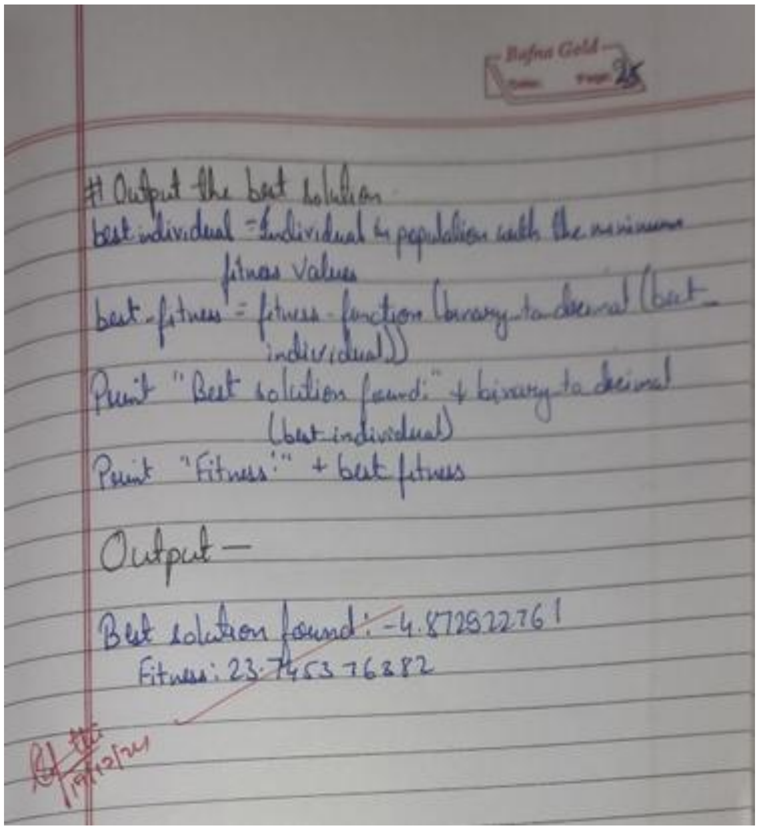
```
Function fitness_function(x):
    Return x^2.

Function binary_to_decimal (binary_str):
    Return Integer (binary_str, 2)/(2^length(binary_str)-1)
        *10 -5

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

Initialize population as an array of population_size with
    random binary strings of length num_genes.

For each generation from 1 to generation DO:
    # Evaluate fitness
    Initialize fitness as an empty array
    For each individual in population DO:
```



24

```
        fitness[individual_index] = fitness_function(binary_to_decimal
                (individual))
    total_fitness = SUM (fitness)
    Initialize probabilities as an empty array
    For each f in fitness DO:
        probabilities [individual_index] = f/total_fitness

    selected = Random_Choice(population, size=population
            size , probabilities = probabilities)
  # Crossover
    Initialize offspring as an empty array
    For i from 0 to population_size -1 step 2 DO:
        If Random values < crossover_rate Then:
            point = Random Integer(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 from 0 to population_size -1 DO:
      If Random_Value < mutation_rate Then:
          point = Random_Integer (0, num_genes -1)
          offspring [i] = offspring[i][:point] +
              (If offspring[i][point] = '0' Then '1'
                  Else '0') + offspring[i][point+1:]
  population = offspring
```

19



Rafna Gold
Date: Page: 25

```
# Output the best solution
best_individual = Individual in population with the minimum
              fitness values
best_fitness = fitness_function (binary_to_decimal (best_
                 individual))
Print "Best solution found:" + binary_to_decimal
                 (best_individual)
Print "Fitness:" + best_fitness
```

Output —

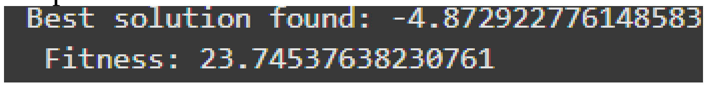
```
Best solution found: -4.872922761
   Fitness: 23.7453 76382
```

19/12/24

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
```