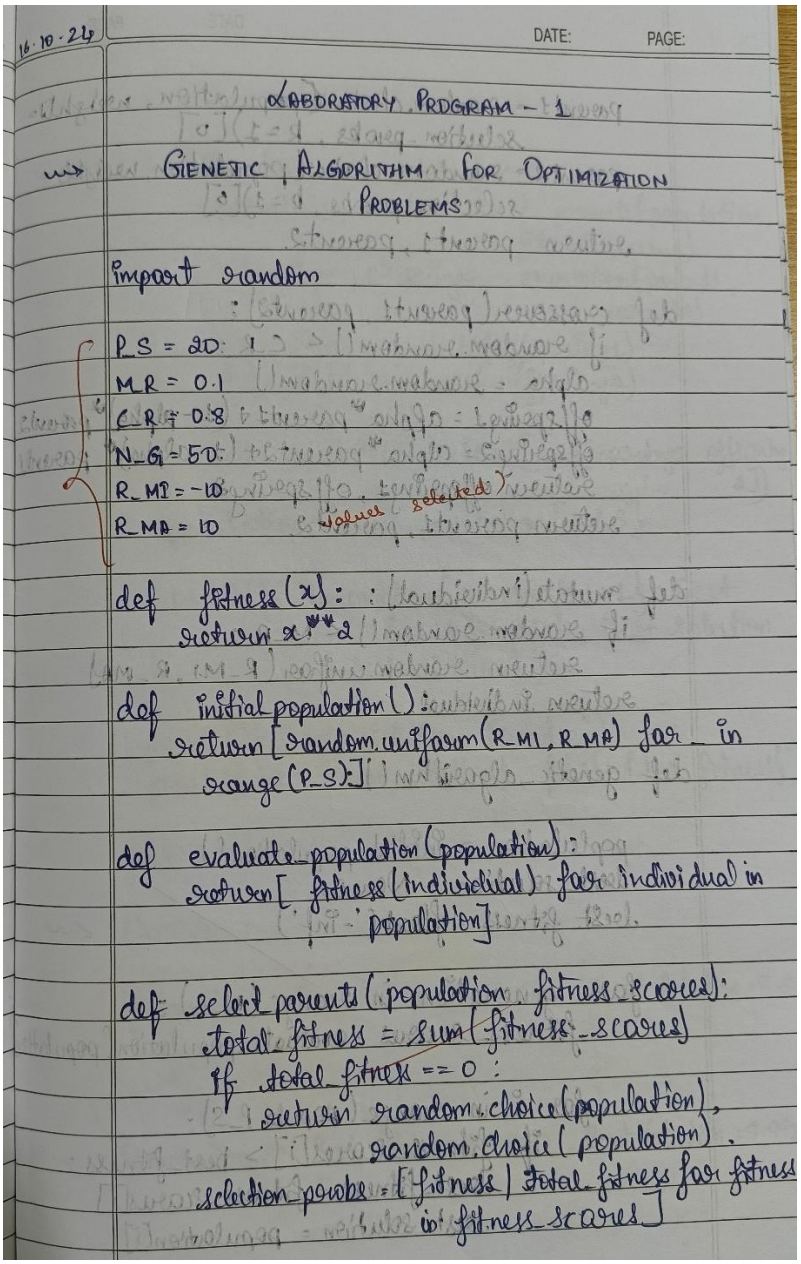
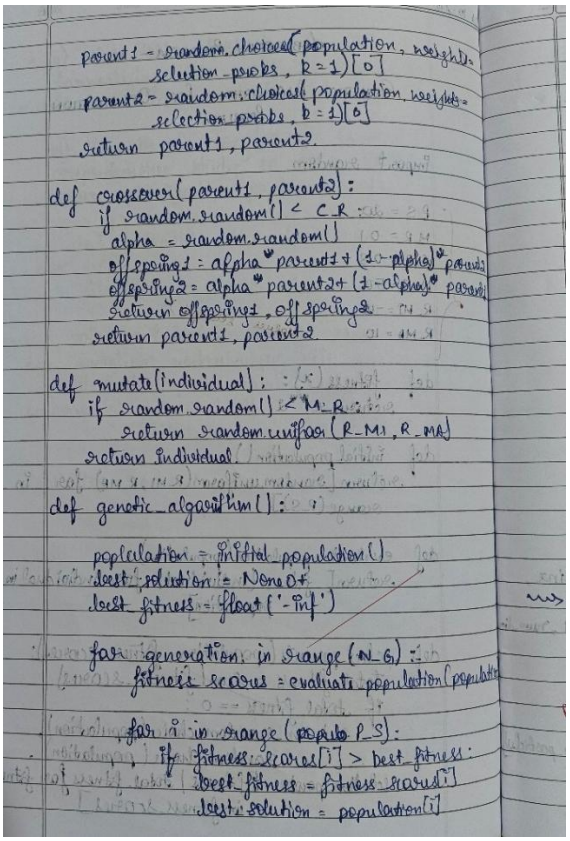
LABORATORY PROGRAM - 1

Genetic Algorithm for Optimization Problems

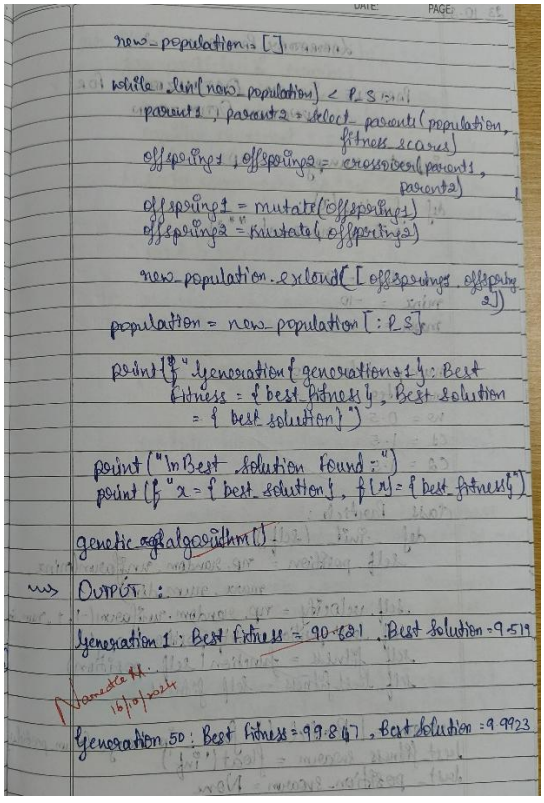
Genetic Algorithms (GA) are inspired by the process of natural selection and genetics, where the fittest individuals are selected for reproduction to produce the next generation. GAs are widely used for solving optimization and search problems. Implement a Genetic Algorithm using Python to solve a basic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

Algorithm





```
parent1 = random.choices(population, weights=
          selection_probs, k=1)[0]
parent2 = random.choices(population, weights=
          selection_probs, k=1)[0]
return parent1, parent2

def crossover(parent1, parent2):
    if random.random() < CR:
        alpha = random.random()
        offspring1 = alpha * parent1 + (1-alpha) * parent2
        offspring2 = alpha * parent2 + (1-alpha) * parent1
        return offspring1, offspring2
    return parent1, parent2

def mutate(individual):
    if random.random() < MUT_R:
        return random.uniform(R-MUT, R+MUT)
    return individual

def genetic_algorithm():

    population = initial_population()
    best_solution = None
    best_fitness = float('-inf')

    for generation in range(N_G):
        fitness_scores = evaluate_population(population)

        for i in range(POPULATION_S):
            if fitness_scores[i] > best_fitness:
                best_fitness = fitness_scores[i]
                best_solution = population[i]
```



```
new_population = []

while len(new_population) < POPULATION_S:
    parent1, parent2 = select_parents(population,
                       fitness_scores)
    offspring1, offspring2 = crossover(parent1,
                             parent2)
    offspring1 = mutate(offspring1)
    offspring2 = mutate(offspring2)

    new_population.extend([offspring1, offspring
                           2])
population = new_population[:POPULATION_S]

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

print("\nBest Solution Found:")
print(f"x = {best_solution}, f(x) = {best_fitness}")

genetic_algorithm()
```

Output :

```
Generation 1: Best Fitness = 90.681, Best Solution = 9.519
Generation 50: Best Fitness = 99.847, Best Solution = 9.9923
```

Code

```
import random

def fitness_function(x, y):
    return x ** 2

population_size = 100
mutation_rate = 0.1
crossover_rate = 0.8
num_generations = 50
variable_bounds = [-10, 10]

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

def evaluate_population(population):
    fitness_scores = []
    for individual in population:
        fitness_scores.append(fitness_function(individual[0], individual[1]))
    return fitness_scores

def selection(population, fitness_scores):
    total_fitness = sum(fitness_scores)
    selected_population = []
    for _ in range(len(population)):
        pick = random.uniform(0, total_fitness)
        current = 0
        for individual, score in zip(population, fitness_scores):
            current += score
            if current > pick:
                selected_population.append(individual)
                break
    return selected_population

def crossover(parent1, parent2, crossover_rate):
    if random.random() < crossover_rate:
        crossover_point = random.randint(1, len(parent1) - 1)
        child1 = parent1[:crossover_point] + parent2[crossover_point:]
        child2 = parent2[:crossover_point] + parent1[crossover_point:]
    else:
        child1, child2 = parent1, parent2
```

```
    return child1, child2

def mutate(individual, mutation_rate, bounds):
    if random.random() < mutation_rate:
        mutation_position = random.randint(0, len(individual) - 1)
        mutation_value = random.uniform(bounds[0], bounds[1])
        individual[mutation_position] = mutation_value
    return individual

def genetic_algorithm():
    # Display student information at the start
    print("Student Name: Likhith M")
    print("USN: 1BM22CS135")
    print("-" * 40)

    population = initialize_population(population_size, variable_bounds)
    overall_best_solution = None
    overall_best_fitness = float('-inf')

    for generation in range(num_generations):
        fitness_scores = evaluate_population(population)

        generation_best_fitness = max(fitness_scores)
        generation_best_solution = population[fitness_scores.index(generation_best_fitness)]

        if generation_best_fitness > overall_best_fitness:
            overall_best_fitness = generation_best_fitness
            overall_best_solution = generation_best_solution

        # Displaying the output for generations that are multiples of 10
        if (generation + 1) % 10 == 0:
            print(f"Generation {generation + 1}:")
            print(f"  Best fitness   = {generation_best_fitness}")
            print(f"  Best solution  = {generation_best_solution}")
            print("-" * 40)

        selected_population = selection(population, fitness_scores)
        next_population = []

        for i in range(0, population_size, 2):
            parent1 = selected_population[i]
            parent2 = selected_population[i + 1]
            child1, child2 = crossover(parent1, parent2, crossover_rate)
            child1 = mutate(child1, mutation_rate, variable_bounds)
            child2 = mutate(child2, mutation_rate, variable_bounds)
            next_population.append(child1)
            next_population.append(child2)
```

```
    population = next_population

print(f"Best solution found after {num_generations} generations: {overall_best_solution}")
print(f"Best fitness value: {overall_best_fitness}")

genetic_algorithm()
```

Output

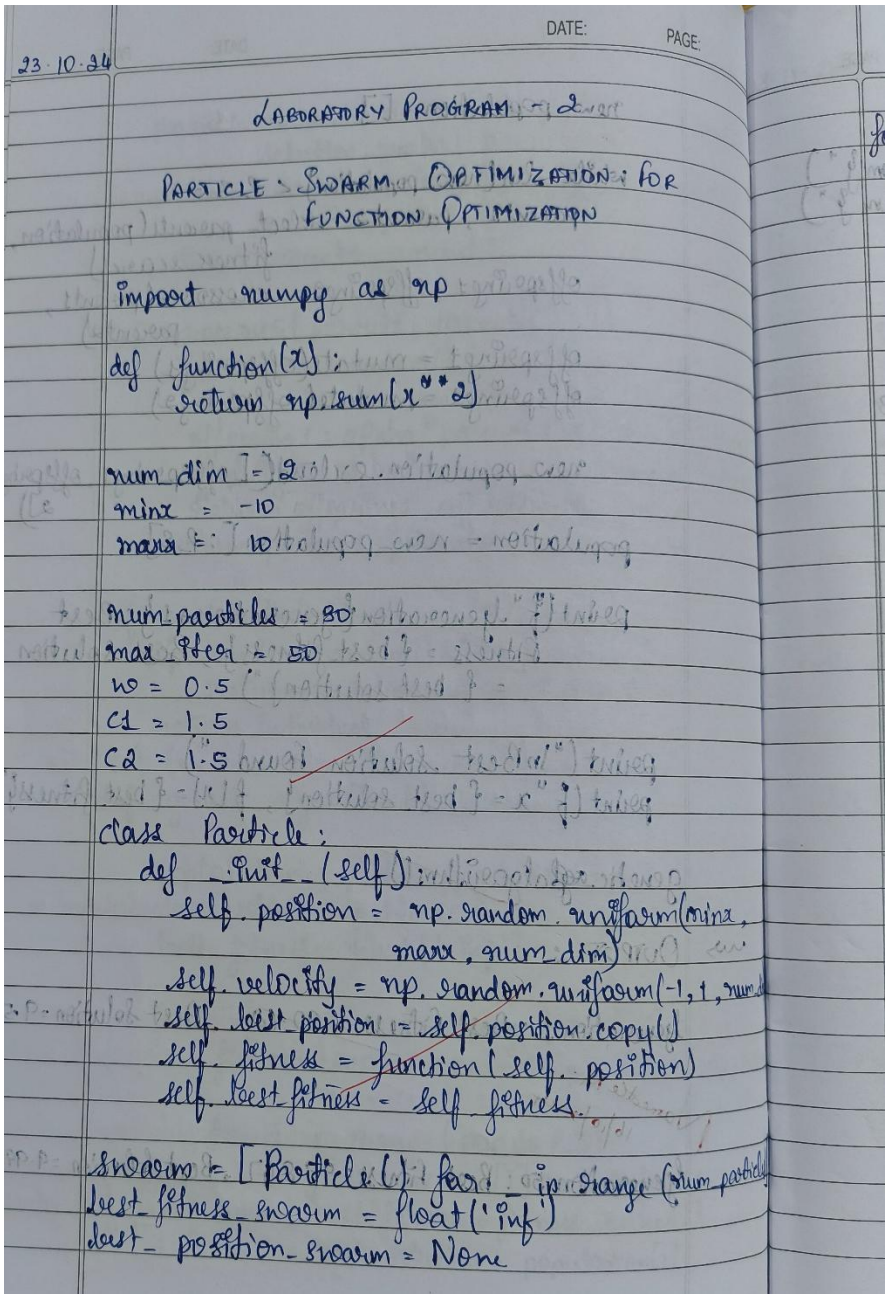
```
Student Name: Likhith M
USN: 1BM22CS135
----------------------------------------
Generation 10:
  Best fitness   = 99.44798466551265
  Best solution  = [-9.972361037663681, -8.984613074594794]
----------------------------------------
Generation 20:
  Best fitness   = 99.44798466551265
  Best solution  = [-9.972361037663681, -9.634560295132298]
----------------------------------------
Generation 30:
  Best fitness   = 99.44798466551265
  Best solution  = [-9.972361037663681, -7.617025418591046]
----------------------------------------
Generation 40:
  Best fitness   = 99.13270771662218
  Best solution  = [-9.956540951385787, -1.8134161191052698]
----------------------------------------
Generation 50:
  Best fitness   = 99.33177287264611
  Best solution  = [-9.966532640424457, -7.824368824053634]
----------------------------------------
Best solution found after 50 generations: [-9.972361037663681, -5.769196750607994]
Best fitness value: 99.44798466551265
```

LABORATORY PROGRAM - 2

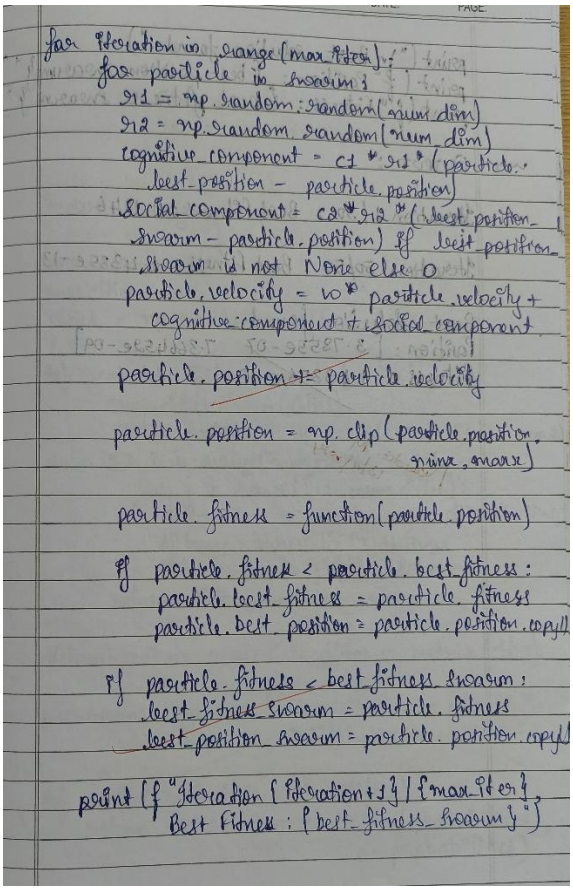
Particle Swarm Optimization for Function Optimization

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.

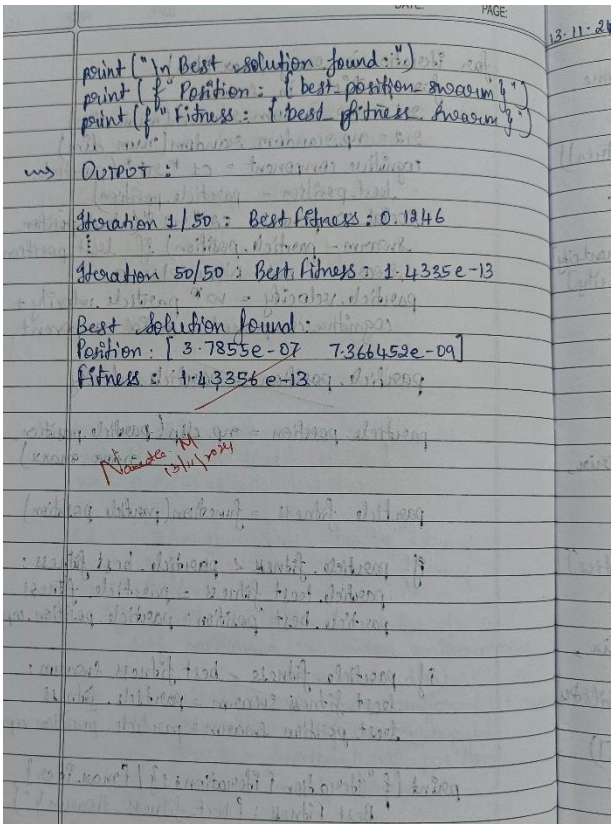
Algorithm



```
23.10.24

LABORATORY PROGRAM - 2

PARTICLE SWARM OPTIMIZATION FOR FUNCTION OPTIMIZATION

import numpy as np

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

num_dim = 2
minx = -10
maxx = 10

num_particles = 30
max_iter = 50
w = 0.5
c1 = 1.5
c2 = 1.5

class Particle:
    def __init__(self):
        self.position = np.random.uniform(minx,
                        maxx, num_dim)
        self.velocity = np.random.uniform(-1, 1, num_
        self.best_position = self.position.copy()
        self.fitness = function(self.position)
        self.best_fitness = self.fitness

swarm = [Particle() for _ in range(num_particl
best_fitness_swarm = float('inf')
best_position_swarm = None
```



```
for iteration in range(max_iter):
    for particle in swarm:
        r1 = np.random.random(num_dim)
        r2 = np.random.random(num_dim)
        cognitive_component = c1 * r1 * (particle.best_position - particle.position)
        social_component = c2 * r2 * (best_position_swarm - particle.position) if best_position_swarm is not None else 0
        particle.velocity = w * particle.velocity + cognitive_component + social_component
        particle.position += particle.velocity

        particle.position = np.clip(particle.position, min, max)

        particle.fitness = function(particle.position)

        if particle.fitness < particle.best_fitness:
            particle.best_fitness = particle.fitness
            particle.best_position = particle.position.copy()

        if particle.fitness < best_fitness_swarm:
            best_fitness_swarm = particle.fitness
            best_position_swarm = particle.position.copy()

    print(f"Iteration {iteration+1}/{max_iter}
          Best Fitness: {best_fitness_swarm}")
```



```
print("\n Best solution found:")
print(f" Position: {best_position_swarm}")
print(f" Fitness: {best_fitness_swarm}")
```

Output:

```
Iteration 1/50 : Best Fitness: 0.1346
⋮
Iteration 50/50 : Best Fitness: 1.4335e-13

Best solution found:
Position: [ 3.7855e-07  7.3664532e-09]
Fitness: 1.43356e-13
```

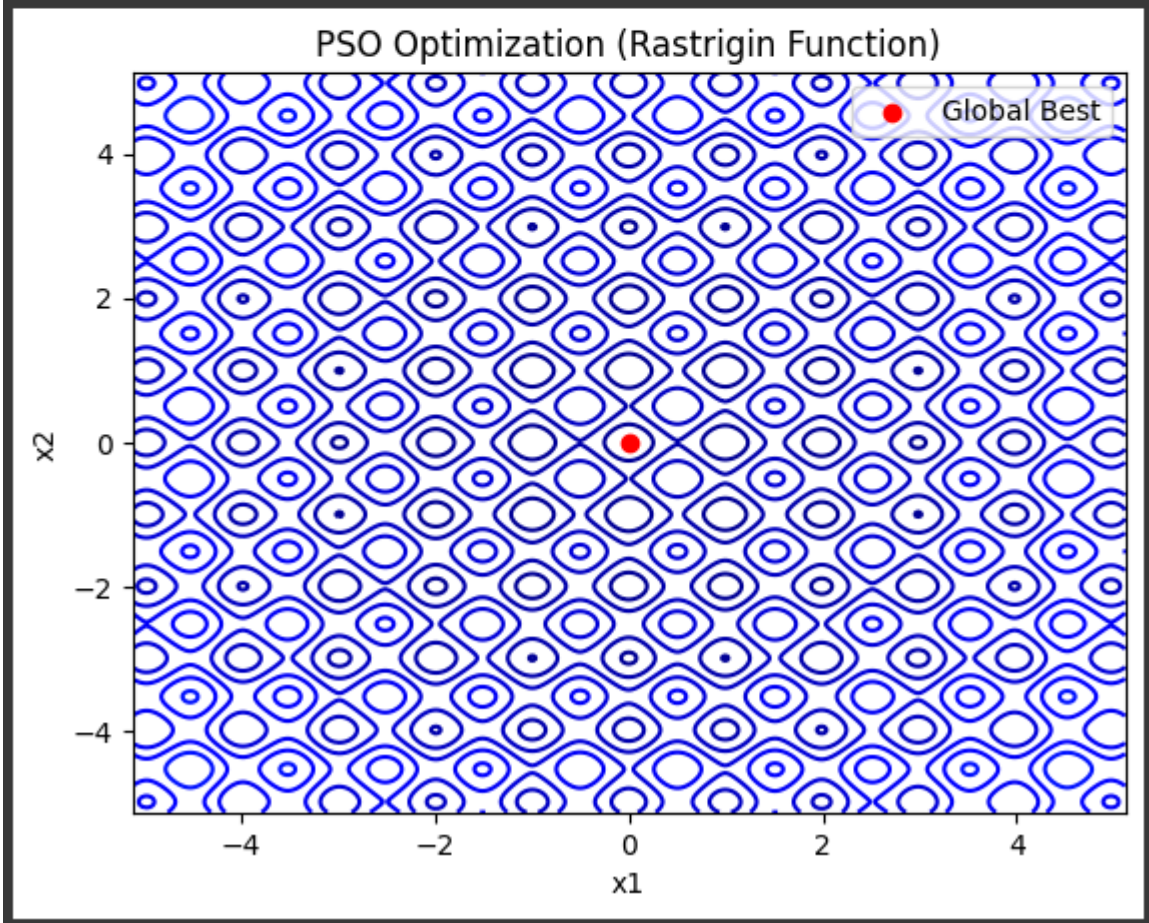
Code

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

# Student Details
print("Student Name: Likhith M")
print("USN: 1BM22CS135\n")

# Objective function (Rastrigin function as an example)
def rastrigin(x):
    A = 10
    return A * len(x) + sum(x_i**2 - A * np.cos(2 * np.pi * x_i) for x_i in x)

# PSO Parameters
n_particles = 30     # Number of particles
n_dimensions = 2     # Number of dimensions (parameters to optimize)
n_iterations = 100   # Number of iterations

w = 0.5              # Inertia weight
c1 = 1.5             # Cognitive coefficient (particle's own best position)
c2 = 1.5             # Social coefficient (global best position)
v_max = 2.0          # Maximum velocity

# Initialize particles' positions and velocities
positions = np.random.uniform(-5.12, 5.12, (n_particles, n_dimensions))
velocities = np.random.uniform(-1, 1, (n_particles, n_dimensions))

# Initialize personal best positions and global best position
pbest_positions = positions.copy()
pbest_values = np.apply_along_axis(rastrigin, 1, pbest_positions)

gbest_position = pbest_positions[np.argmin(pbest_values)]
gbest_value = np.min(pbest_values)

# PSO Main Loop
for t in range(n_iterations):
    # Evaluate fitness
    fitness_values = np.apply_along_axis(rastrigin, 1, positions)

    # Update personal bests
    for i in range(n_particles):
        if fitness_values[i] < pbest_values[i]:
            pbest_positions[i] = positions[i]
            pbest_values[i] = fitness_values[i]

    # Update global best
```

```
    min_fitness_idx = np.argmin(pbest_values)
    if pbest_values[min_fitness_idx] < gbest_value:
        gbest_position = pbest_positions[min_fitness_idx]
        gbest_value = pbest_values[min_fitness_idx]

    # Update velocity and position for each particle
    r1 = np.random.rand(n_particles, n_dimensions)
    r2 = np.random.rand(n_particles, n_dimensions)

    velocities = (w * velocities +
                  c1 * r1 * (pbest_positions - positions) +
                  c2 * r2 * (gbest_position - positions))

    # Apply velocity limits (optional)
    velocities = np.clip(velocities, -v_max, v_max)

    # Update positions
    positions = positions + velocities

    # Print the current best solution only for multiples of 10
    if t % 10 == 0:
        print(f"Iteration {t}, Global Best Value: {gbest_value:.5f}")

# Final output
print(f"\nFinal Global Best Position: {gbest_position}")
print(f"Final Global Best Value: {gbest_value:.5f}")

# Plotting the optimization process (visualization for 2D case)
x_vals = np.linspace(-5.12, 5.12, 400)
y_vals = np.linspace(-5.12, 5.12, 400)
X, Y = np.meshgrid(x_vals, y_vals)
Z = rastrigin([X, Y])

plt.contour(X, Y, Z, levels=np.linspace(0, 500, 50), cmap='jet')
plt.scatter(gbest_position[0], gbest_position[1], color='red', label='Global Best')
plt.title("PSO Optimization (Rastrigin Function)")
plt.xlabel('x1')
plt.ylabel('x2')
plt.legend()
plt.show()
```

Output

```
Student Name: Likhith M
USN: 1BM22CS135

Iteration 0, Global Best Value: 7.29865
Iteration 10, Global Best Value: 0.01789
Iteration 20, Global Best Value: 0.00018
Iteration 30, Global Best Value: 0.00000
Iteration 40, Global Best Value: 0.00000
Iteration 50, Global Best Value: 0.00000
Iteration 60, Global Best Value: 0.00000
Iteration 70, Global Best Value: 0.00000
Iteration 80, Global Best Value: 0.00000
Iteration 90, Global Best Value: 0.00000

Final Global Best Position: [2.10172483e-09 6.30154843e-10]
Final Global Best Value: 0.00000
```

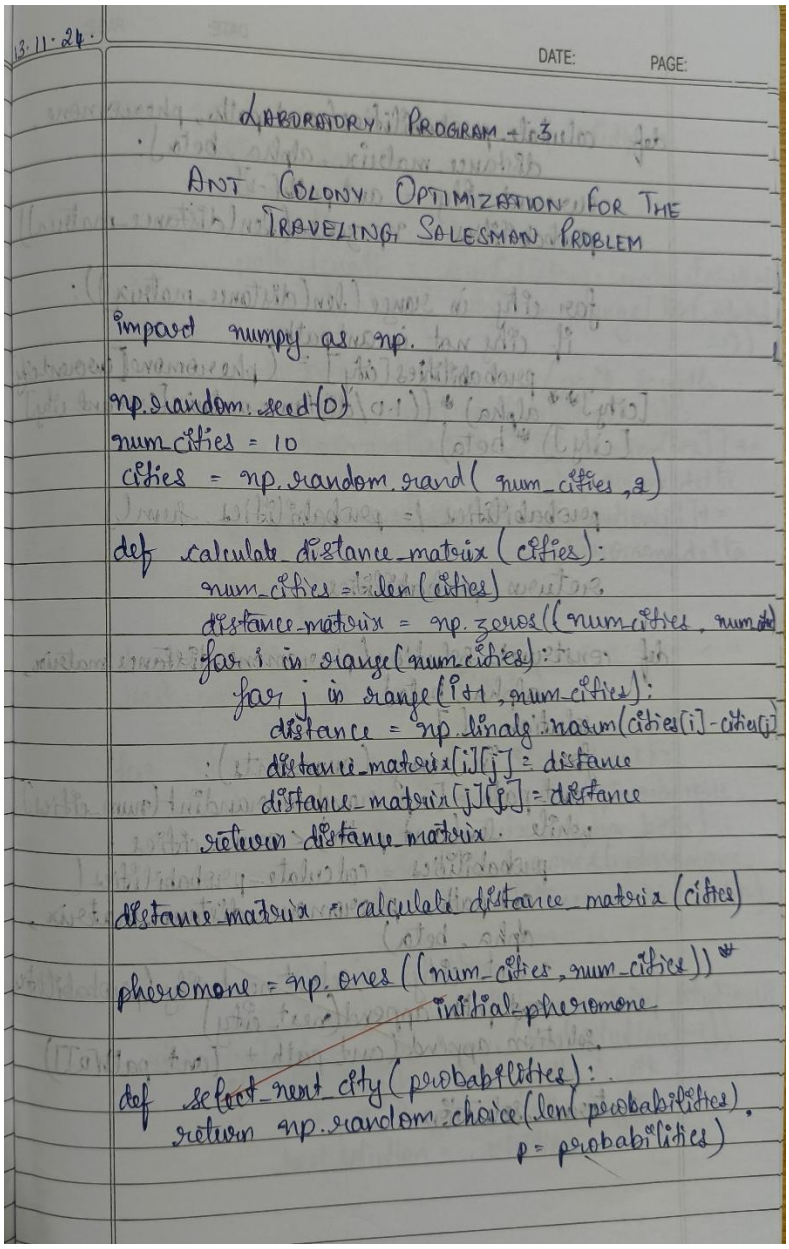


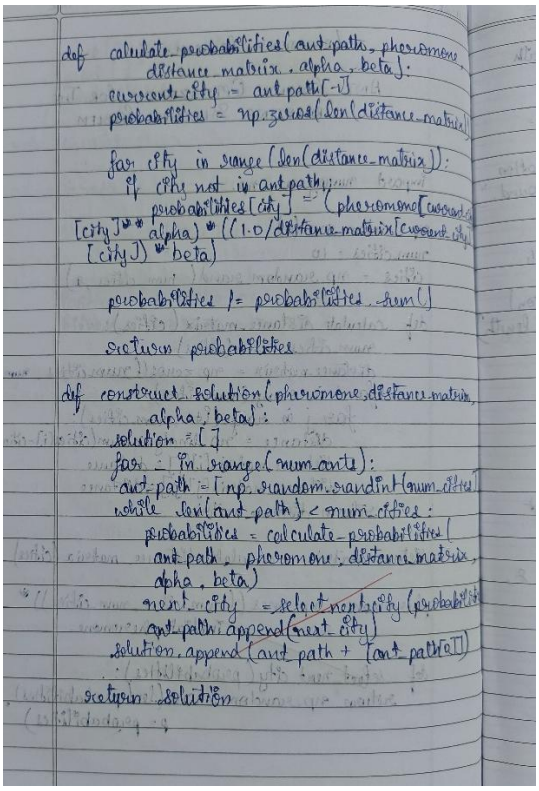
LABORATORY PROGRAM - 3

Ant Colony Optimization for the Traveling Salesman Problem

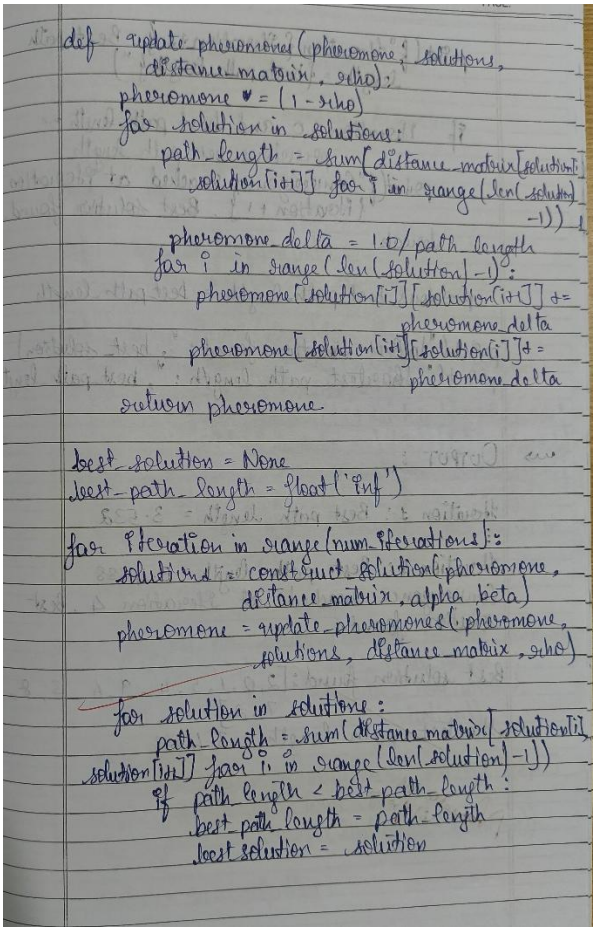
The foraging behavior of ants has inspired the development of optimization algorithms that can solve complex problems such as the Traveling Salesman Problem (TSP). Ant Colony Optimization (ACO) simulates the way ants find the shortest path between food sources and their nest. Implement the ACO algorithm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

Algorithm

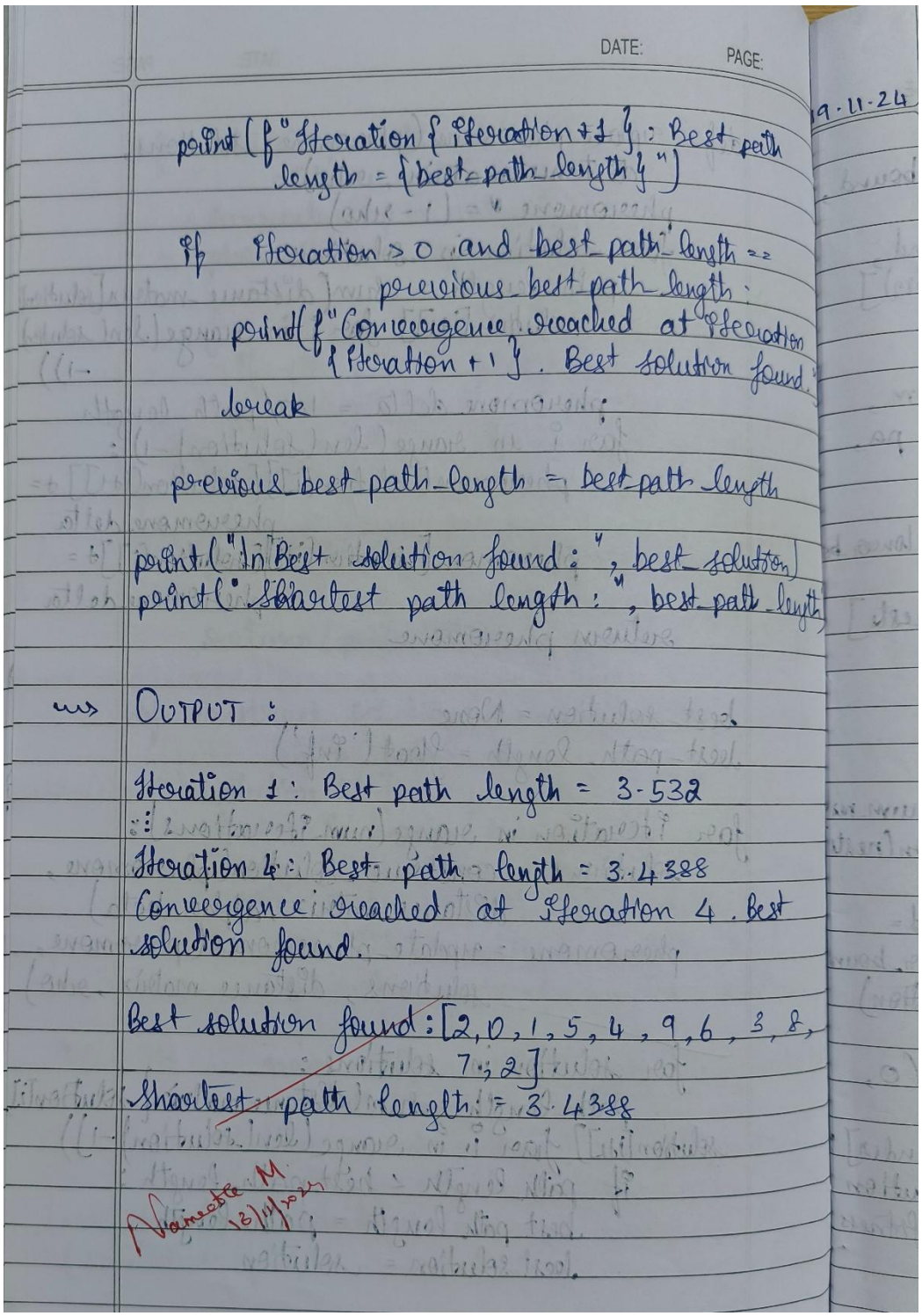




```
def calculate_probabilities(ant_path, pheromone,
        distance_matrix, alpha, beta):
    current_city = ant_path[-1]
    probabilities = np.zeros(len(distance_matrix))

    for city in range(len(distance_matrix)):
        if city not in ant_path:
            probabilities[city] = (pheromone[current_
[city]] ** alpha) * ((1.0/distance_matrix[current_city
[city]]) ** beta)

    probabilities /= probabilities.sum()

    return probabilities

def construct_solution(pheromone, distance_matrix,
        alpha, beta):
    solution = []
    for _ in range(num_ants):
        ant_path = [np.random.randint(num_cities)]
        while len(ant_path) < num_cities:
            probabilities = calculate_probabilities(
            ant_path, pheromone, distance_matrix,
            alpha, beta)
            next_city = select_next_city(probabilities,
            ant_path.append(next_city)
        solution.append(ant_path + [ant_path[0]])
    return solution
```



```
def update_pheromones(pheromone, solutions,
        distance_matrix, rho):
    pheromone *= (1 - rho)
    for solution in solutions:
        path_length = sum[distance_matrix[solution[i],
        solution[i+1]] for i in range(len(solution
                                        -1))]
        pheromone_delta = 1.0/path_length
        for i in range(len(solution)-1):
            pheromone[solution[i]][solution[i+1]] +=
                                        pheromone_delta
            pheromone[solution[i+1]][solution[i]] +=
                                        pheromone_delta
    return pheromone

best_solution = None
best_path_length = float('inf')

for iteration in range(num_iterations):
    solutions = construct_solution(pheromone,
            distance_matrix, alpha, beta)
    pheromone = update_pheromones(pheromone,
            solutions, distance_matrix, rho)

    for solution in solutions:
        path_length = sum(distance_matrix[solution[i],
        solution[i+1]] for i in range(len(solution)-1))
        if path_length < best_path_length:
            best_path_length = path_length
            best_solution = solution
```



```
        print(f"Iteration {iteration+1}: Best path
              length = {best_path_length}")

        if iteration > 0 and best_path_length ==
                        previous_best_path_length:
            print(f"Convergence reached at iteration
                  {iteration+1}. Best solution found.")
            break

        previous_best_path_length = best_path_length

    print("\nBest solution found:", best_solution)
    print("Shortest path length:", best_path_length)
```

19-11-24

⇝ OUTPUT :

```
Iteration 1: Best path length = 3.532
.
.
.
Iteration 4: Best path length = 3.4388
Convergence reached at iteration 4. Best
solution found.

Best solution found: [2, 0, 1, 5, 4, 9, 6, 3, 8,
                      7, 2]
Shortest path length: 3.4388
```

Namrata M
18/11/2024

Code

```
import numpy as np

# Student Details
print("Student Name: Likhith M")
print("USN: 1BM22CS135\n")

np.random.seed(0)

# Problem Parameters
num_cities = 10
num_ants = 20
alpha = 1.0
beta = 5.0
rho = 0.5
initial_pheromone = 0.1
num_iterations = 100

# Generate Random Cities
cities = np.random.rand(num_cities, 2)

# Distance Matrix Calculation
def calculate_distance_matrix(cities):
    num_cities = len(cities)
    distance_matrix = np.zeros((num_cities, num_cities))
    for i in range(num_cities):
        for j in range(i + 1, num_cities):
            distance = np.linalg.norm(cities[i] - cities[j])
            distance_matrix[i][j] = distance
            distance_matrix[j][i] = distance
    return distance_matrix

distance_matrix = calculate_distance_matrix(cities)

# Initialize Pheromone Matrix
pheromone = np.ones((num_cities, num_cities)) * initial_pheromone

# City Selection
def select_next_city(probabilities):
    return np.random.choice(len(probabilities), p=probabilities)

# Probability Calculation
def calculate_probabilities(ant_path, pheromone, distance_matrix, alpha, beta):
    current_city = ant_path[-1]
    probabilities = np.zeros(len(distance_matrix))

    for city in range(len(distance_matrix)):
```

```
        if city not in ant_path:
            probabilities[city] = (pheromone[current_city][city] ** alpha) * ((1.0 /
distance_matrix[current_city][city]) ** beta)
    probabilities /= probabilities.sum()
    return probabilities

# Solution Construction
def construct_solution(pheromone, distance_matrix, alpha, beta):
    solution = []
    for _ in range(num_ants):
        ant_path = [np.random.randint(num_cities)]
        while len(ant_path) < num_cities:
            probabilities = calculate_probabilities(ant_path, pheromone, distance_matrix, alpha, beta)
            next_city = select_next_city(probabilities)
            ant_path.append(next_city)
        solution.append(ant_path + [ant_path[0]])
    return solution

# Pheromone Update
def update_pheromones(pheromone, solutions, distance_matrix, rho):
    pheromone *= (1 - rho)
    for solution in solutions:
        path_length = sum(distance_matrix[solution[i], solution[i+1]] for i in range(len(solution) - 1))
        pheromone_delta = 1.0 / path_length
        for i in range(len(solution) - 1):
            pheromone[solution[i]][solution[i + 1]] += pheromone_delta
            pheromone[solution[i + 1]][solution[i]] += pheromone_delta
    return pheromone

# Optimization Process
best_solution = None
best_path_length = float('inf')

for iteration in range(num_iterations):
    solutions = construct_solution(pheromone, distance_matrix, alpha, beta)
    pheromone = update_pheromones(pheromone, solutions, distance_matrix, rho)

    for solution in solutions:
        path_length = sum(distance_matrix[solution[i], solution[i + 1]] for i in range(len(solution) - 1))
        if path_length < best_path_length:
            best_path_length = path_length
            best_solution = solution

    # Display output only for multiples of 10
    if (iteration + 1) % 10 == 0:
        print(f"Iteration {iteration + 1}: Path length = {best_path_length:.5f}")
```

```
    # Check for convergence
    if iteration > 0 and best_path_length == previous_best_path_length:
        print(f"Convergence reached at iteration {iteration + 1}. Best solution found.")
        break

    previous_best_path_length = best_path_length

# Final Output
print("\nBest solution found:", best_solution)
print("Shortest path length:", best_path_length)
```

Output

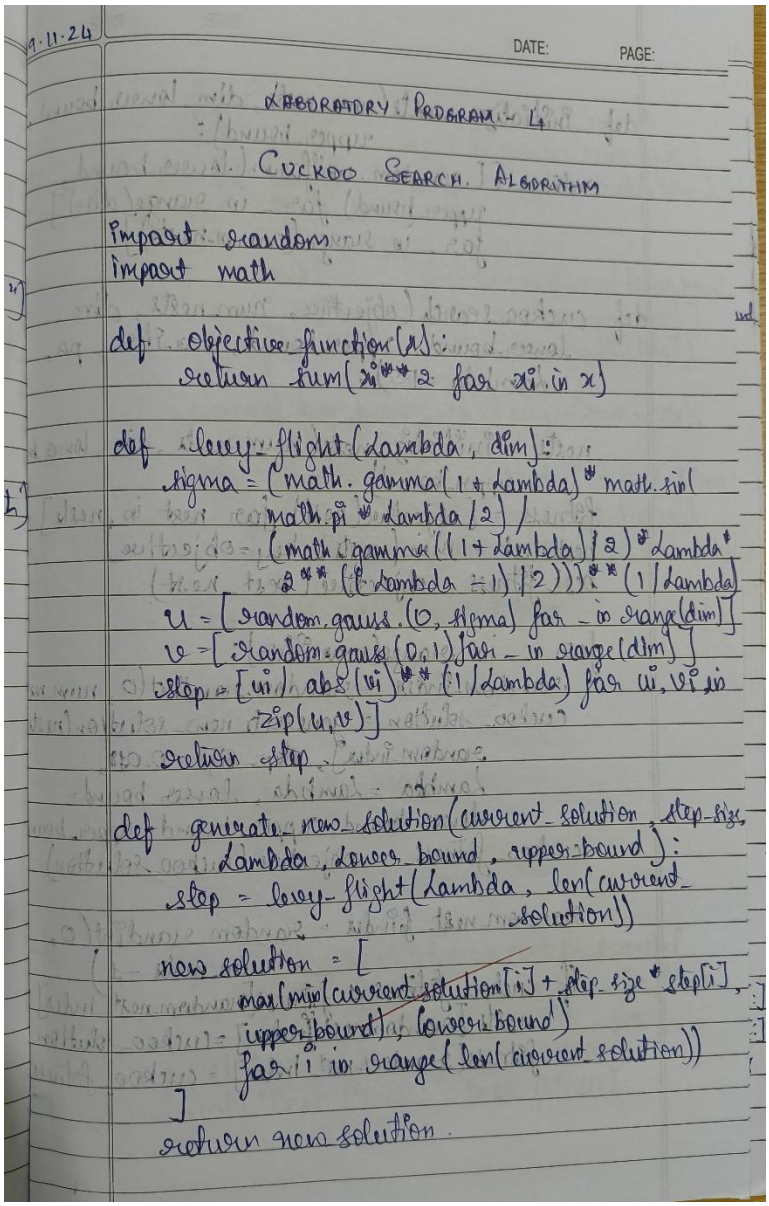
```
Student Name: Likhith M
USN: 1BM22CS135

Convergence reached at iteration 4. Best solution found.

Best solution found: [2, 0, 1, 5, 4, 9, 6, 3, 8, 7, 2]
Shortest path length: 3.4388850126686448
```

LABORATORY PROGRAM - 4

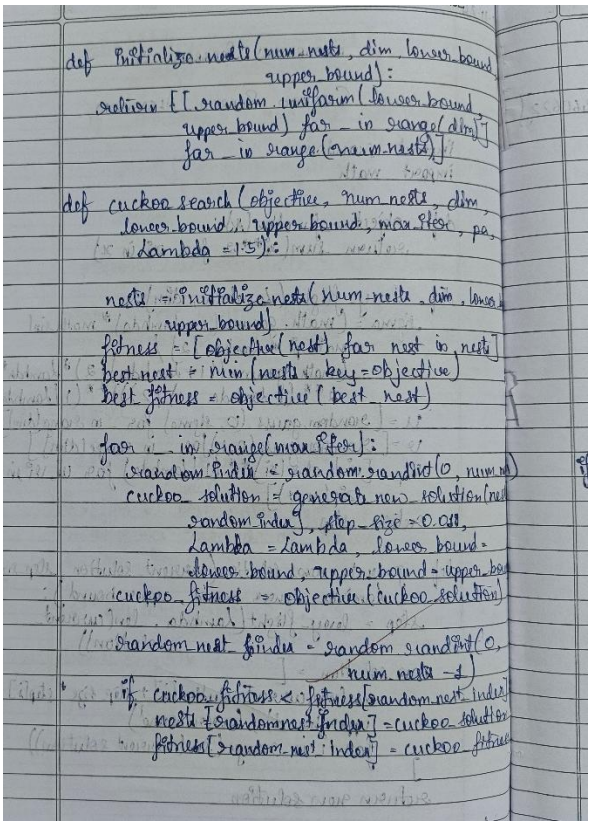
Cuckoo Search (CS)

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.

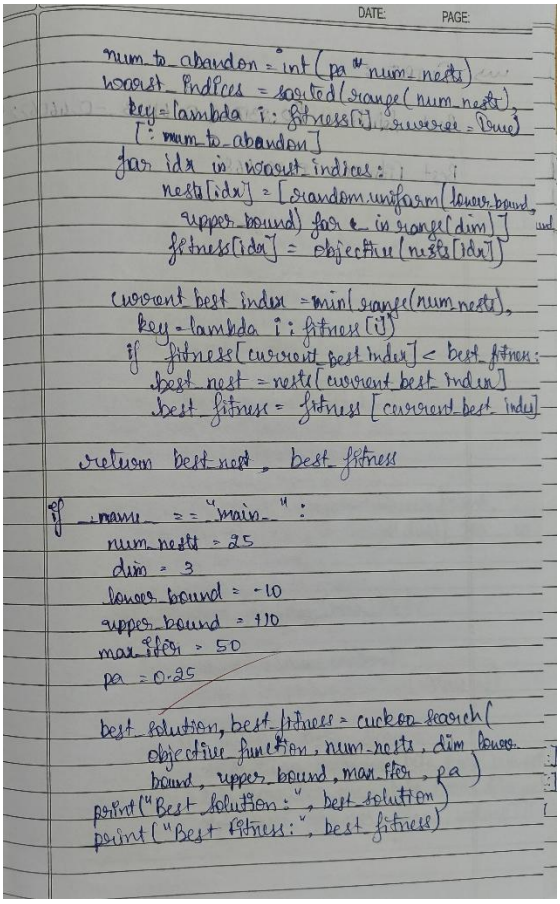
Algorithm



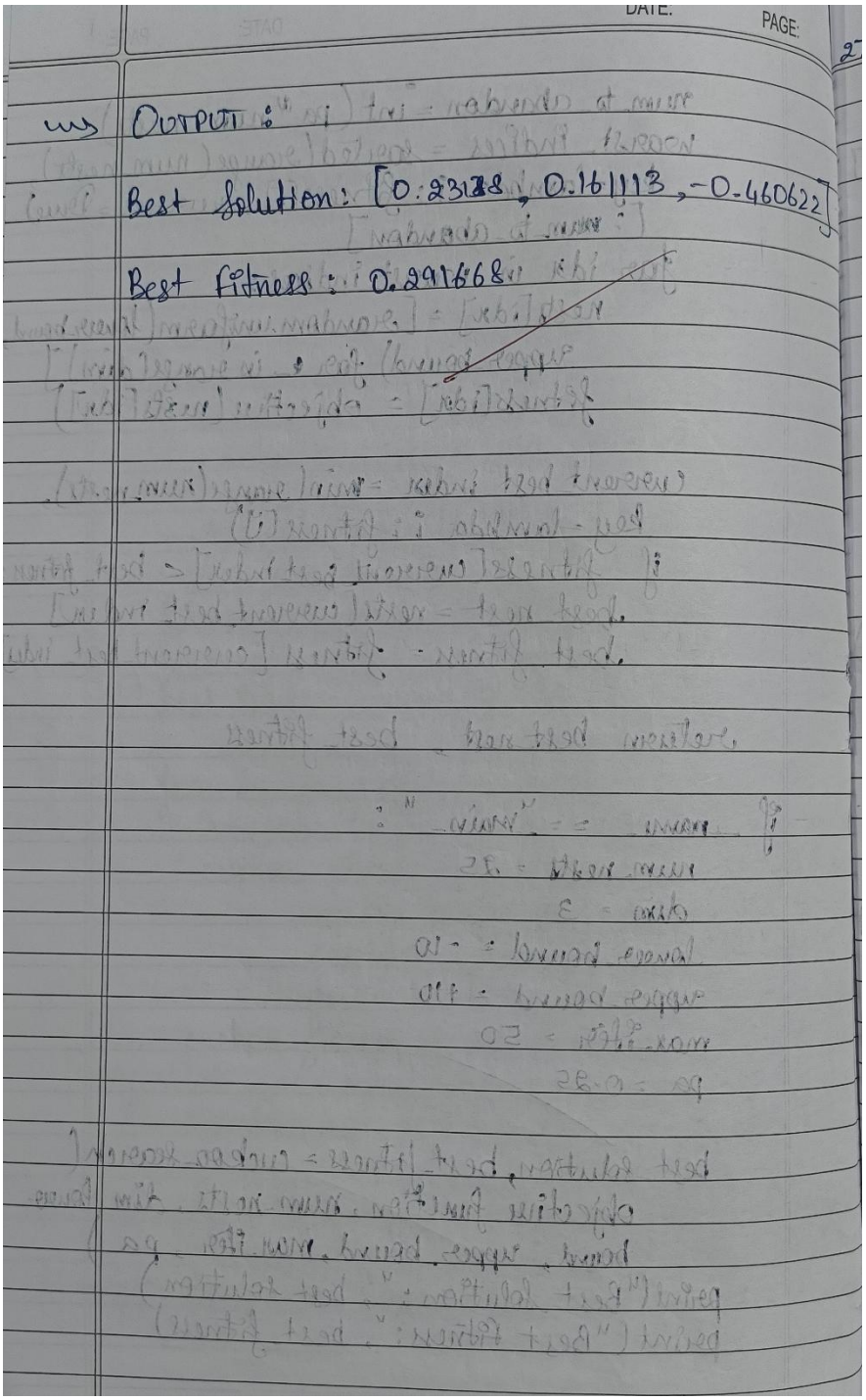
```
9.11.24                                   DATE:        PAGE:

LABORATORY PROGRAM - 4

CUCKOO SEARCH ALGORITHM

import random
import math

def objective_function(x):
    return sum(xi**2 for xi in x)

def levy_flight(lambda, dim):
    sigma = (math.gamma(1+lambda) * math.sin(
        math.pi * lambda/2) /
        (math.gamma((1+lambda)/2) * lambda *
        2**((lambda-1)/2)))**(1/lambda)
    u = [random.gauss(0, sigma) for _ in range(dim)]
    v = [random.gauss(0,1) for _ in range(dim)]
    step = [ui / abs(vi)**(1/lambda) for ui, vi in
        zip(u,v)]
    return step

def generate_new_solution(current_solution, step_size,
    lambda, lower_bound, upper_bound):
    step = levy_flight(lambda, len(current_
        solution))
    new_solution = [
        max(min(current_solution[i] + step_size * step[i],
        upper_bound), lower_bound)
        for i in range(len(current_solution))
    ]
    return new_solution
```



```
def initialize_nests(num_nests, dim, lower_bound,
                     upper_bound):
    return [[random.uniform(lower_bound,
             upper_bound) for _ in range(dim)]
            for _ in range(num_nests)]

def cuckoo_search(objective, num_nests, dim,
    lower_bound, upper_bound, max_iter, pa,
    lambda_=1.5):

    nests = initialize_nests(num_nests, dim, lower_
        upper_bound)
    fitness = [objective(nest) for nest in nests]
    best_nest = min(nests, key=objective)
    best_fitness = objective(best_nest)

    for _ in range(max_iter):
        random_index = random.randint(0, num_n
        cuckoo_solution = generate_new_solution(nes
            random_index], step_size=0.01,
            lambda_=lambda_, lower_bound=
            lower_bound, upper_bound=upper_bo
        cuckoo_fitness = objective(cuckoo_solution)

        random_nest_index = random.randint(0,
            num_nests-1)
        if cuckoo_fitness < fitness[random_nest_index]:
            nests[random_nest_index] = cuckoo_solution
            fitness[random_nest_index] = cuckoo_fitness
```



```
    num_to_abandon = int(pa * num_nests)
    worst_indices = sorted(range(num_nests),
        key=lambda i: fitness[i], reverse=True)
        [:num_to_abandon]
    for idx in worst_indices:
        nests[idx] = [random.uniform(lower_bound,
            upper_bound) for _ in range(dim)]
        fitness[idx] = objective(nests[idx])

    current_best_index = min(range(num_nests),
        key=lambda i: fitness[i])
    if fitness[current_best_index] < best_fitness:
        best_nest = nests[current_best_index]
        best_fitness = fitness[current_best_index]

return best_nest, best_fitness

if __name__ == "__main__":
    num_nests = 25
    dim = 3
    lower_bound = -10
    upper_bound = 10
    max_iter = 50
    pa = 0.25

    best_solution, best_fitness = cuckoo_search(
        objective_function, num_nests, dim, lower_
        bound, upper_bound, max_iter, pa)
    print("Best solution:", best_solution)
    print("Best fitness:", best_fitness)
```


DATE:
PAGE:
OUTPUT :
Best Solution: [0.23188, 0.161113, -0.460622]
Best Fitness: 0.291668

Code

```
import random
import math

# Student Details
print("Student Name: Likhith M")
print("USN: 1BM22CS135\n")

# Objective Function
def objective_function(x):
    return sum(xi ** 2 for xi in x)

# Levy Flight Step
def levy_flight(Lambda, dim):
    sigma = (math.gamma(1 + Lambda) * math.sin(math.pi * Lambda / 2) /
             (math.gamma((1 + Lambda) / 2) * Lambda * 2 ** ((Lambda - 1) / 2))) ** (1 / Lambda)
    u = [random.gauss(0, sigma) for _ in range(dim)]
    v = [random.gauss(0, 1) for _ in range(dim)]
    step = [ui / abs(vi) ** (1 / Lambda) for ui, vi in zip(u, v)]
    return step

# Generate New Solution
def generate_new_solution(current_solution, alpha, Lambda, lower_bound, upper_bound):
    step = levy_flight(Lambda, len(current_solution))
    new_solution = [
        max(min(current_solution[i] + alpha * step[i], upper_bound), lower_bound)
        for i in range(len(current_solution))
    ]
    return new_solution

# Initialize Nests
def initialize_nests(n_nests, dim, lower_bound, upper_bound):
    return [[random.uniform(lower_bound, upper_bound) for _ in range(dim)] for _ in range(n_nests)]

# Cuckoo Search Algorithm
def cuckoo_search(objective, n_nests, max_iter, alpha, pa, lower_bound=-10, upper_bound=10,
Lambda=1.5):
    nests = initialize_nests(n_nests, 2, lower_bound, upper_bound)
    fitness = [objective(nest) for nest in nests]
    best_nest = min(nests, key=objective)
```

```
    best_fitness = objective(best_nest)

    for iteration in range(max_iter):
        random_index = random.randint(0, n_nests - 1)
        cuckoo_solution = generate_new_solution(nests[random_index], alpha, Lambda, lower_bound,
upper_bound)
        cuckoo_fitness = objective(cuckoo_solution)

        random_nest_index = random.randint(0, n_nests - 1)
        if cuckoo_fitness < fitness[random_nest_index]:
            nests[random_nest_index] = cuckoo_solution
            fitness[random_nest_index] = cuckoo_fitness

        num_to_abandon = int(pa * n_nests)
        worst_indices = sorted(range(n_nests), key=lambda i: fitness[i],
reverse=True)[:num_to_abandon]
        for idx in worst_indices:
            nests[idx] = [random.uniform(lower_bound, upper_bound) for _ in range(2)]
            fitness[idx] = objective(nests[idx])

        current_best_index = min(range(n_nests), key=lambda i: fitness[i])
        if fitness[current_best_index] < best_fitness:
            best_nest = nests[current_best_index]
            best_fitness = fitness[current_best_index]

        # Output every 100 iterations
        if iteration % 100 == 0 or iteration == max_iter - 1:
            print(f"Iteration {iteration}, Best Fitness: {best_fitness:.5f}, Best Solution: {best_nest}")

    return best_nest, best_fitness

# Main Function
if __name__ == "__main__":
    n_nests = 25    # Number of nests
    max_iter = 1000 # Maximum iterations
    alpha = 0.1     # Step size
    pa = 0.25       # Probability of abandoning worse nests

    best_solution, best_value = cuckoo_search(
        objective_function, n_nests=n_nests, max_iter=max_iter, alpha=alpha, pa=pa
    )
```

```
print(f"\nBest solution found: x = {best_solution[0]:.5f}, y = {best_solution[1]:.5f}")
print(f"Best objective function value: {best_value:.5f}")
```

Output

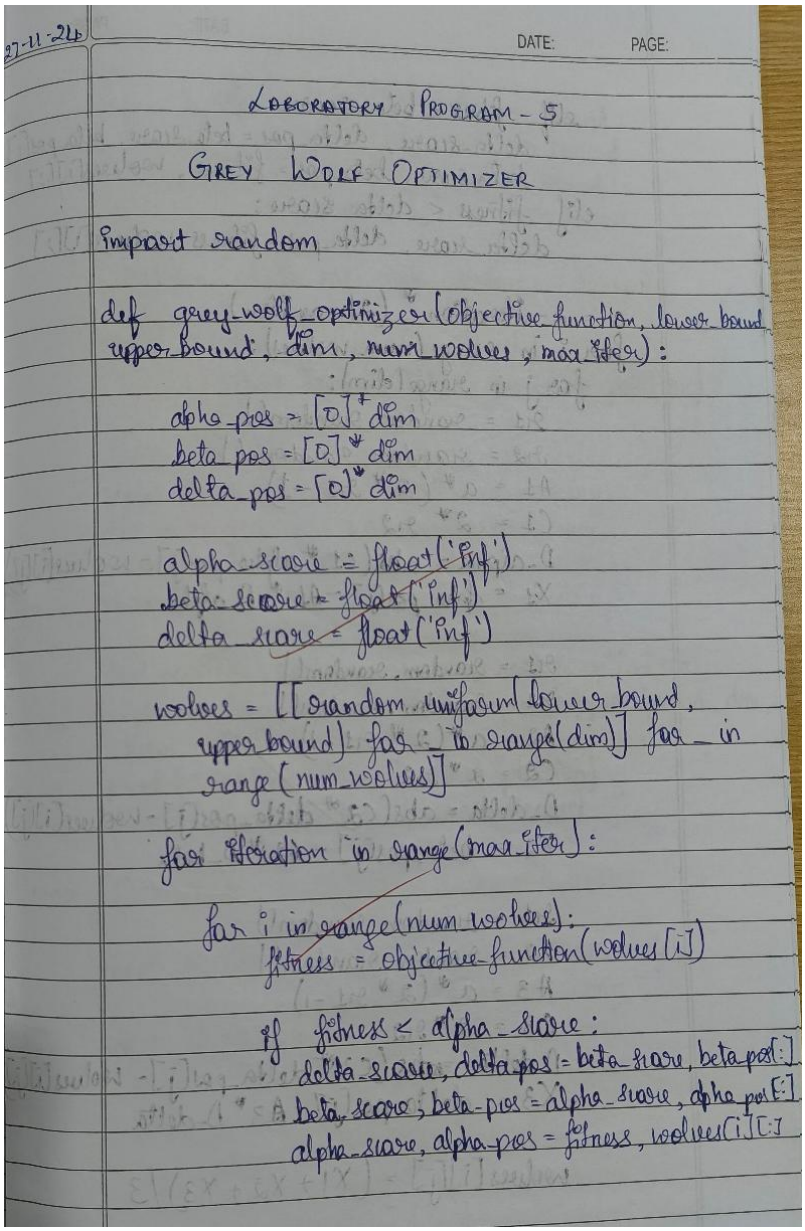
```
Student Name: Likhith M
USN: 1BM22CS135

Iteration 0, Best Fitness: 0.22830, Best Solution: [-0.4664999684868931, 0.1033251221074103]
Iteration 100, Best Fitness: 0.05437, Best Solution: [-0.23267040694054952, 0.015338160420397762]
Iteration 200, Best Fitness: 0.00443, Best Solution: [0.06251808820158372, 0.022832902631498175]
Iteration 300, Best Fitness: 0.00010, Best Solution: [-0.009763824782933361, -0.0005021375169151215]
Iteration 400, Best Fitness: 0.00004, Best Solution: [-0.005905180653441653, -0.0016099886133596197]
Iteration 500, Best Fitness: 0.00004, Best Solution: [-0.005905180653441653, -0.0016099886133596197]
Iteration 600, Best Fitness: 0.00004, Best Solution: [-0.005905180653441653, -0.0016099886133596197]
Iteration 700, Best Fitness: 0.00004, Best Solution: [-0.005905180653441653, -0.0016099886133596197]
Iteration 800, Best Fitness: 0.00004, Best Solution: [-0.005905180653441653, -0.0016099886133596197]
Iteration 900, Best Fitness: 0.00004, Best Solution: [-0.005905180653441653, -0.0016099886133596197]
Iteration 999, Best Fitness: 0.00001, Best Solution: [0.0033637344832600207, -0.0008645107976493899]

Best solution found: x = 0.00336, y = -0.00086
Best objective function value: 0.00001
```

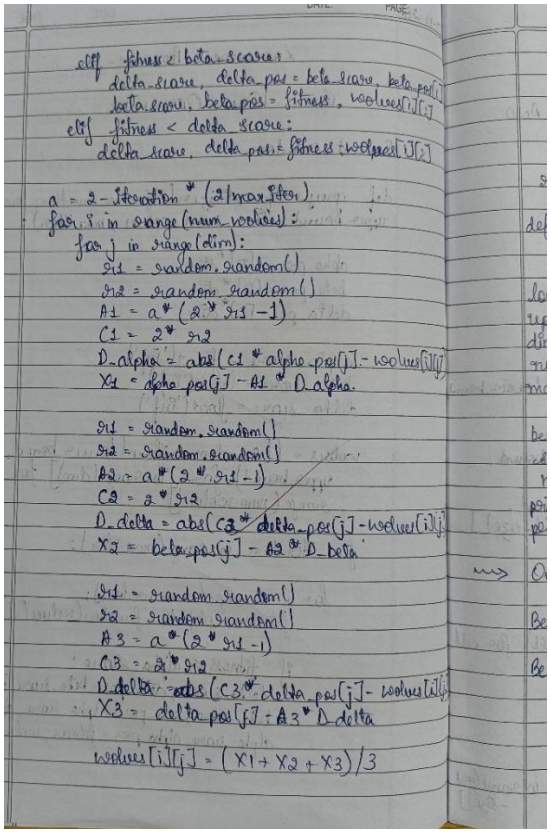
LABORATORY PROGRAM - 5

Grey Wolf Optimizer (GWO)

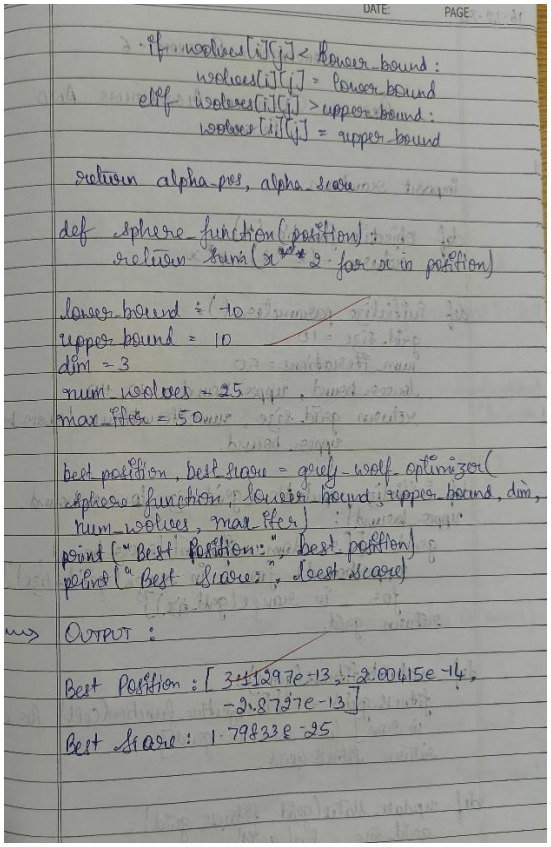
The Grey Wolf Optimizer (GWO) algorithm is a swarm intelligence algorithm inspired by the social hierarchy and hunting behavior of grey wolves. It mimics the leadership structure of alpha, beta, delta, and omega wolves and their collaborative hunting strategies. The GWO algorithm uses these social hierarchies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

Algorithm





```
elif fitness < beta_score:
    delta_score, delta_pos = beta_score, beta_pos[:]
    beta_score, beta_pos = fitness, wolves[i][:]
elif fitness < delta_score:
    delta_score, delta_pos = fitness, wolves[i][:]

a = 2 - iteration * (2/max_iter)
for i in range(num_wolves):
    for j in range(dim):
        r1 = random.random()
        r2 = random.random()
        A1 = a * (2 * r1 - 1)
        C1 = 2 * r2
        D_alpha = abs(C1 * alpha_pos[j] - wolves[i][j])
        X1 = alpha_pos[j] - A1 * D_alpha

        r1 = random.random()
        r2 = random.random()
        A2 = a * (2 * r1 - 1)
        C2 = 2 * r2
        D_beta = abs(C2 * beta_pos[j] - wolves[i][j])
        X2 = beta_pos[j] - A2 * D_beta

        r1 = random.random()
        r2 = random.random()
        A3 = a * (2 * r1 - 1)
        C3 = 2 * r2
        D_delta = abs(C3 * delta_pos[j] - wolves[i][j])
        X3 = delta_pos[j] - A3 * D_delta

        wolves[i][j] = (X1 + X2 + X3) / 3
```



```
        if wolves[i][j] < lower_bound:
            wolves[i][j] = lower_bound
        elif wolves[i][j] > upper_bound:
            wolves[i][j] = upper_bound

    return alpha_pos, alpha_score

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

lower_bound = -10
upper_bound = 10
dim = 3
num_wolves = 25
max_iter = 150

best_position, best_score = grey_wolf_optimizer(
    sphere_function, lower_bound, upper_bound, dim,
    num_wolves, max_iter)
print("Best position:", best_position)
print("Best score:", best_score)
```

OUTPUT :

```
Best Position : [ 3.51297e-13, -2.00415e-14,
-2.87377e-13 ]
Best Score : 1.79833e-25
```

Code

```
import random

# Student Details
print("Student Name: Likhith M")
print("USN: 1BM22CS135\n")

# Grey Wolf Optimizer Function
def grey_wolf_optimizer(objective_function, lower_bound, upper_bound, dim, num_wolves, max_iter):

    alpha_pos = [0] * dim
    beta_pos = [0] * dim
    delta_pos = [0] * dim

    alpha_score = float('inf')
    beta_score = float('inf')
    delta_score = float('inf')

    wolves = [[random.uniform(lower_bound, upper_bound) for _ in range(dim)] for _ in range(num_wolves)]

    for iteration in range(max_iter):
        for i in range(num_wolves):
            fitness = objective_function(wolves[i])

            if fitness < alpha_score:
                delta_score, delta_pos = beta_score, beta_pos[:]
                beta_score, beta_pos = alpha_score, alpha_pos[:]
                alpha_score, alpha_pos = fitness, wolves[i][:]
            elif fitness < beta_score:
                delta_score, delta_pos = beta_score, beta_pos[:]
                beta_score, beta_pos = fitness, wolves[i][:]
            elif fitness < delta_score:
                delta_score, delta_pos = fitness, wolves[i][:]

        a = 2 - iteration * (2 / max_iter)
        for i in range(num_wolves):
            for j in range(dim):
                r1 = random.random()
                r2 = random.random()
                A1 = a * (2 * r1 - 1)
                C1 = 2 * r2
                D_alpha = abs(C1 * alpha_pos[j] - wolves[i][j])
                X1 = alpha_pos[j] - A1 * D_alpha

                r1 = random.random()
```

```
            r2 = random.random()
            A2 = a * (2 * r1 - 1)
            C2 = 2 * r2
            D_beta = abs(C2 * beta_pos[j] - wolves[i][j])
            X2 = beta_pos[j] - A2 * D_beta

            r1 = random.random()
            r2 = random.random()
            A3 = a * (2 * r1 - 1)
            C3 = 2 * r2
            D_delta = abs(C3 * delta_pos[j] - wolves[i][j])
            X3 = delta_pos[j] - A3 * D_delta

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

            if wolves[i][j] < lower_bound:
                wolves[i][j] = lower_bound
            elif wolves[i][j] > upper_bound:
                wolves[i][j] = upper_bound

        # Output every 10 iterations
        if iteration % 10 == 0 or iteration == max_iter - 1:
            print(f"Iteration {iteration}: Best Score = {alpha_score:.5f}, Best Position = {alpha_pos}")

    return alpha_pos, alpha_score

# Sphere Function (Objective Function)
def sphere_function(position):
    return sum(x ** 2 for x in position)

# Problem Parameters
lower_bound = -10
upper_bound = 10
dim = 3
num_wolves = 25
max_iter = 50

# Run Grey Wolf Optimizer
best_position, best_score = grey_wolf_optimizer(sphere_function, lower_bound, upper_bound, dim,
num_wolves, max_iter)

# Final Output
print("\nFinal Best Position:", best_position)
print("Final Best Score:", best_score)
```

Output

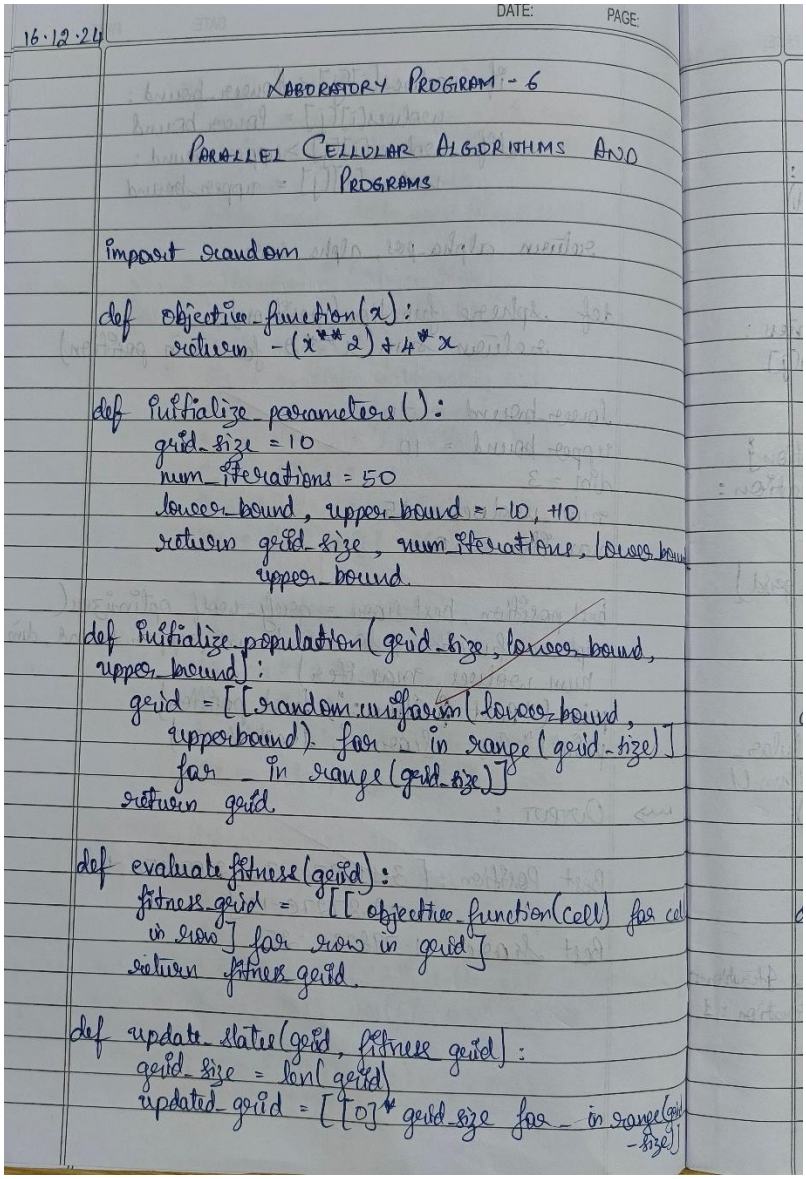
```
Student Name: Likhith M
USN: 1BM22CS135

Iteration 0: Best Score = 8.87469, Best Position = [-1.3937875919991694, 2.6192794260861127, 0.2672459070460409]
Iteration 10: Best Score = 0.00006, Best Position = [-0.0022888418251507912, 0.005312156238702422, 0.0047323367161349284]
Iteration 20: Best Score = 0.00000, Best Position = [-5.509811050364098e-06, -4.388573147797282e-06, 3.946947280531768e-06]
Iteration 30: Best Score = 0.00000, Best Position = [-2.4489778136877548e-08, -8.658917449203874e-09, -1.4297757082884138e-08]
Iteration 40: Best Score = 0.00000, Best Position = [-4.301575426213075e-09, -4.106251702412561e-09, -4.374118169436466e-09]
Iteration 49: Best Score = 0.00000, Best Position = [-3.2476095209067505e-09, -2.7864689468237177e-09, -2.931715786065307e-09]

Final Best Position: [-3.2476095209067505e-09, -2.7864689468237177e-09, -2.931715786065307e-09]
Final Best Score: 2.690633424216157e-17
```

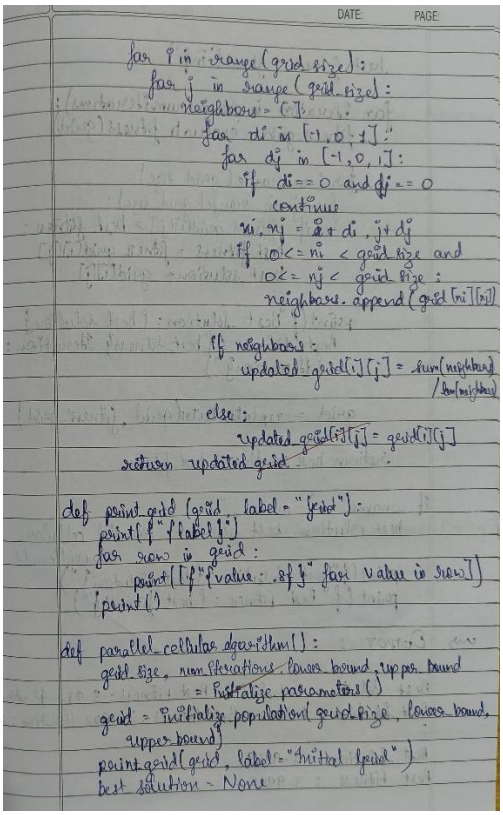
LABORATORY PROGRAM - 6

Parallel Cellular Algorithms and Programs

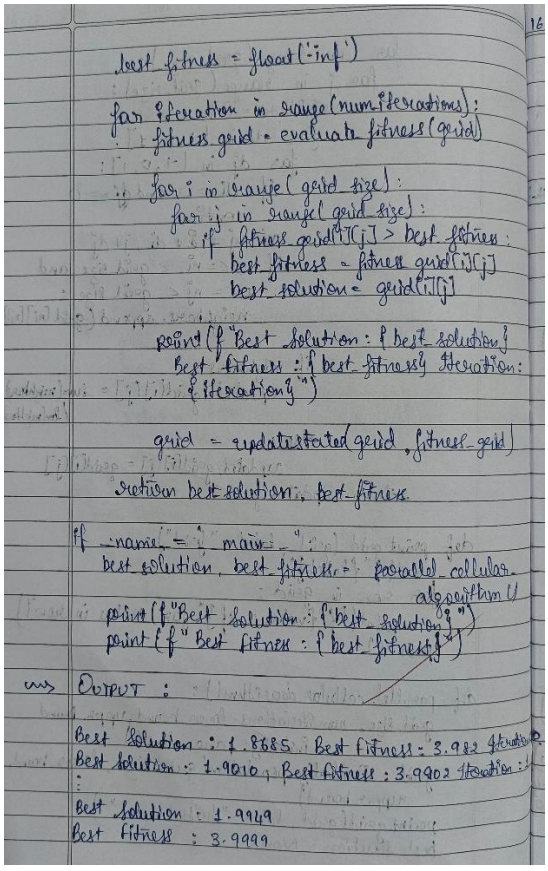
Parallel Cellular Algorithms are inspired by the functioning of biological cells that operate in a highly parallel and distributed manner. These algorithms leverage the principles of cellular automata and parallel computing to solve complex optimization problems efficiently. Each cell represents a potential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

Algorithm





```
for i in range(grid_size):
    for j in range(grid_size):
        neighbors = []
        for di in [-1, 0, 1]:
            for dj in [-1, 0, 1]:
                if di == 0 and dj == 0
                    continue
                ni, nj = i + di, j + dj
                if 0 <= ni < grid_size and
                    0 <= nj < grid_size:
                    neighbors.append(grid[ni][nj])
        if neighbors:
            updated_grid[i][j] = sum(neighbors)
                                 / len(neighbors)
        else:
            updated_grid[i][j] = grid[i][j]
return updated_grid

def print_grid(grid, label="Grid"):
    print(f"{label}:")
    for row in grid:
        print([f"{value:.3f}" for value in row])
    print()

def parallel_cellular_algorithm():
    grid_size, num_iterations, lower_bound, upper_bound
        = Initialize_parameters()
    grid = Initialize_population(grid_size, lower_bound,
        upper_bound)
    print_grid(grid, label="Initial Grid")
    best_solution = None
```



```
    best_fitness = float('-inf')

    for iteration in range(num_iterations):
        fitness_grid = evaluate_fitness(grid)

        for i in range(grid_size):
            for j in range(grid_size):
                if fitness_grid[i][j] > best_fitness:
                    best_fitness = fitness_grid[i][j]
                    best_solution = grid[i][j]

        print(f"Best Solution: {best_solution}
            Best Fitness: {best_fitness} Iteration:
            {iteration}")

        grid = update_state(grid, fitness_grid)
    return best_solution, best_fitness

if __name__ == "__main__":
    best_solution, best_fitness = parallel_cellular_
                                  algorithm()
    print(f"Best Solution: {best_solution}")
    print(f"Best Fitness: {best_fitness}")
```

OUTPUT :

```
Best Solution : 1.8685 Best Fitness : 3.982 Iteration:
Best Solution : 1.9010 , Best Fitness : 3.9902 Iteration:
:
Best Solution : 1.9949
Best Fitness : 3.9999
```

Code

```
import random

# Student Details
print("Student Name: Likhith M")
print("USN: 1BM22CS135\n")

# Objective Function
def objective_function(x):
    return -(x ** 2) + 4 * x

# Initialize Parameters
def initialize_parameters():
    grid_size = 10  # Grid size
    num_iterations = 50  # Number of iterations
    lower_bound, upper_bound = -10, 10  # Bounds for the grid values
    return grid_size, num_iterations, lower_bound, upper_bound

# Initialize Population Grid
def initialize_population(grid_size, lower_bound, upper_bound):
    grid = [[random.uniform(lower_bound, upper_bound) for _ in range(grid_size)] for _ in
range(grid_size)]
    return grid

# Evaluate Fitness Grid
def evaluate_fitness(grid):
    fitness_grid = [[objective_function(cell) for cell in row] for row in grid]
    return fitness_grid

# Update Grid States Based on Neighbor Averages
def update_states(grid, fitness_grid):
    grid_size = len(grid)
    updated_grid = [[0] * grid_size for _ in range(grid_size)]

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

            if neighbors:
                updated_grid[i][j] = sum(neighbors) / len(neighbors)
```

```
            else:
                updated_grid[i][j] = grid[i][j]

    return updated_grid

# Print Grid State
def print_grid(grid, label="Grid"):
    print(f"{label}")
    for row in grid:
        print([f"{value: .4f}" for value in row])
    print()

# Parallel Cellular Algorithm
def parallel_cellular_algorithm():
    grid_size, num_iterations, lower_bound, upper_bound = initialize_parameters()
    grid = initialize_population(grid_size, lower_bound, upper_bound)

    print_grid(grid, label="Initial Grid")

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

    for iteration in range(num_iterations):
        fitness_grid = evaluate_fitness(grid)

        for i in range(grid_size):
            for j in range(grid_size):
                if fitness_grid[i][j] > best_fitness:
                    best_fitness = fitness_grid[i][j]
                    best_solution = grid[i][j]

        # Output progress at multiples of 10 iterations
        if iteration % 10 == 0 or iteration == num_iterations - 1:
            print(f"Iteration {iteration}: Best Solution = {best_solution:.4f}, Best Fitness =
{best_fitness:.4f}")

        grid = update_states(grid, fitness_grid)

    return best_solution, best_fitness

# Main Function
if __name__ == "__main__":
    best_solution, best_fitness = parallel_cellular_algorithm()
    print("\nFinal Results:")
    print(f"Best Solution: {best_solution:.4f}")
    print(f"Best Fitness: {best_fitness:.4f}")
```

Output

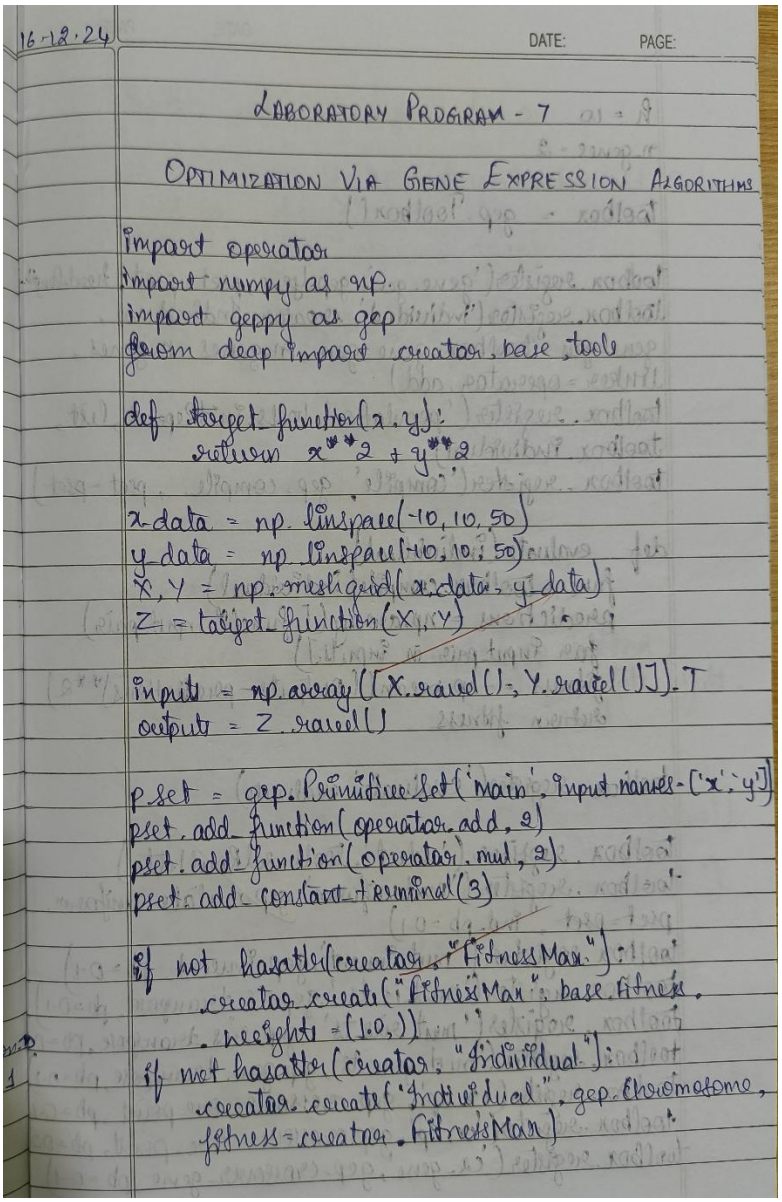
```
Student Name: Likhith M
USN: 1BM22CS135

Initial Grid
[' 2.1002', '-8.5424', ' 0.5768', ' 1.0575', ' 8.1478', ' 5.6918', ' 0.0955', '-7.0900', ' 8.5333', '-5.1055']
['-8.9354', '-9.6884', '-5.2842', '-3.4578', '-4.9637', '-0.9836', '-3.5976', ' 8.3437', ' 6.9574', '-6.4682']
['-1.9309', '-7.2601', '-4.5101', ' 7.2017', '-8.3728', '-7.4899', '-8.2406', '-6.4359', '-0.4393', ' 1.8302']
[' 0.2574', ' 8.0121', '-4.9665', '-4.6416', ' 7.0306', ' 0.2532', ' 8.5815', ' 3.3289', ' 8.8667', '-5.1253']
[' 3.9054', '-6.9647', '-7.1257', '-5.9857', ' 8.9998', '-3.9828', ' 6.6955', '-7.3007', ' 9.8741', '-4.1903']
[' 5.8162', '-8.5069', ' 6.6198', ' 8.0581', '-7.6086', ' 7.3042', ' 8.2752', ' 1.8829', ' 3.0308', ' 3.5125']
['-1.9685', ' 7.2189', ' 7.0566', ' 5.4051', ' 0.6901', '-5.4429', '-6.6738', ' 9.6035', '-8.9977', ' 7.5862']
['-9.0879', '-0.7581', '-4.5743', ' 6.4059', '-6.0020', '-3.6441', ' 3.8070', '-7.6950', '-3.8934', '-3.1362']
['-3.2436', ' 4.7337', '-1.0044', '-9.3255', ' 1.6884', '-0.4363', ' 1.3384', ' 9.5544', ' 6.7410', '-2.7891']
[' 2.3002', '-6.1585', ' 3.3928', '-0.9514', '-3.0380', ' 2.2574', '-0.1634', ' 7.6562', '-8.3370', ' 9.5738']

Iteration 0: Best Solution = 2.1002, Best Fitness = 3.9900
Iteration 10: Best Solution = 1.9938, Best Fitness = 4.0000
Iteration 20: Best Solution = 1.9938, Best Fitness = 4.0000
Iteration 30: Best Solution = 1.9938, Best Fitness = 4.0000
Iteration 40: Best Solution = 1.9938, Best Fitness = 4.0000
Iteration 49: Best Solution = 1.9938, Best Fitness = 4.0000

Final Results:
Best Solution: 1.9938
Best Fitness: 4.0000
```

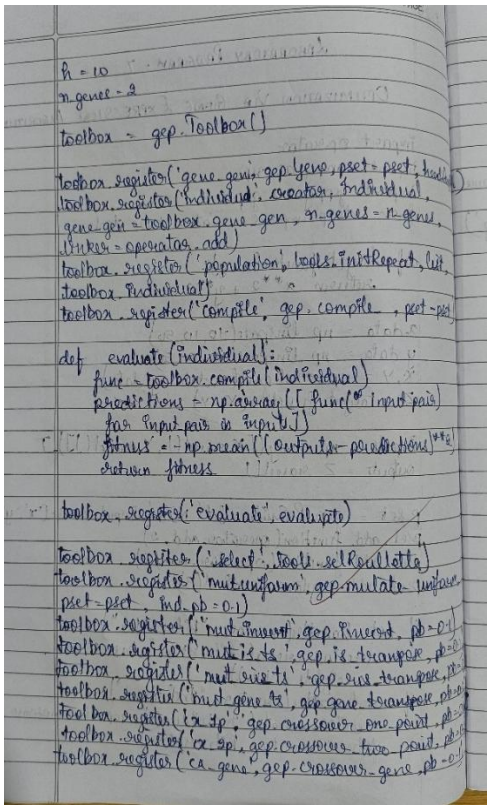
LABORATORY PROGRAM - 7

Optimization via Gene Expression Algorithms

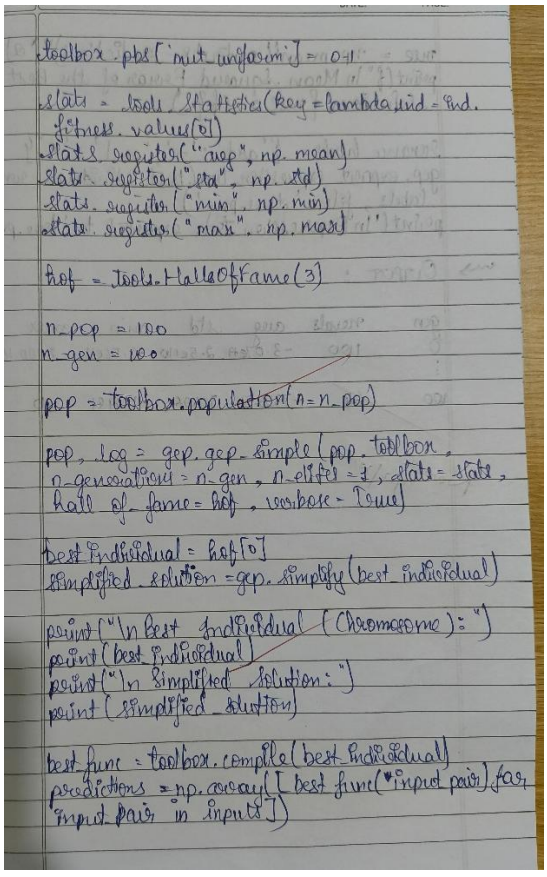
Gene Expression Algorithms (GEA) are inspired by the biological process of gene expression in living organisms. This process involves the translation of genetic information encoded in DNA into functional proteins. In GEA, solutions to optimizatsion problems are encoded in a manner similar to genetic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

Algorithm





```
h = 10
n_genes = 2

toolbox = gep.Toolbox()

toolbox.register('gene_gen', gep.Gene, pset=pset, head=h)
toolbox.register('individual', creator.Individual,
gene_gen=toolbox.gene_gen, n_genes=n_genes,
linker=operator.add)
toolbox.register('population', tools.initRepeat, list,
toolbox.individual)
toolbox.register('compile', gep.compile_, pset=pset)

def evaluate(individual):
    func = toolbox.compile(individual)
    predictions = np.array([func(*input_pair)
    for input_pair in inputs])
    fitness = -np.mean((outputs-predictions)**2)
    return fitness

toolbox.register('evaluate', evaluate)

toolbox.register('select', tools.selRoulette)
toolbox.register('mut_uniform', gep.mutate_uniform,
pset=pset, ind_pb=0.1)
toolbox.register('mut_invert', gep.invert, pb=0.1)
toolbox.register('mut_is_ts', gep.is_transpose, pb=0.1)
toolbox.register('mut_ris_ts', gep.ris_transpose, pb=0.1)
toolbox.register('mut_gene_ts', gep.gene_transpose, pb=0.1)
toolbox.register('cx_1p', gep.crossover_one_point, pb=0.4)
toolbox.register('cx_2p', gep.crossover_two_point, pb=0.2)
toolbox.register('cx_gene', gep.crossover_gene, pb=0.1)
```



```
toolbox.pbs['mut_uniform'] = 0.1

stats = tools.Statistics(key=lambda ind: ind.
fitness.values[0])
stats.register("avg", np.mean)
stats.register("std", np.std)
stats.register("min", np.min)
stats.register("max", np.max)

hof = tools.HallOfFame(3)

n_pop = 100
n_gen = 100

pop = toolbox.population(n=n_pop)

pop, log = gep.gep_simple(pop, toolbox,
n_generations=n_gen, n_elites=1, stats=stats,
hall_of_fame=hof, verbose=True)

best_individual = hof[0]
simplified_solution = gep.simplify(best_individual)

print("\nBest individual (Chromosome):")
print(best_individual)
print("\nSimplified solution:")
print(simplified_solution)

best_func = toolbox.compile(best_individual)
predictions = np.array([best_func(*input_pair) for
input_pair in inputs])
```

DATE: PAGE:

```
mse = np.mean((outputs - predictions)**2)
print(f"\nMean Squared Error of the Best Solution: {mse:.6f}")

rename_labels = {'add': '+', 'mul': '*'}
gep.export_expression_tree(best_individual, rename_labels, file='tree.png')
print("\nExpression tree exported to 'tree.png'")
```

⇒ OUTPUT :

gen	nevals	avg	std	min	max
0	100	-3.07e+9	2.5e+10	-2.56e11	-2200.14
:					
100	64	+7355.51	304064	-162981	-0

Namrata. M
17/12/2022

Code

```
import operator
import numpy as np
import geppy as gep
from deap import creator, base, tools

# Student Information
print("Student Name: Likhith M")
print("USN: 1BM22CS135\n")

# Step 1: Define the target function
def target_function(x, y):
    return x**2 + y**2

# Step 2: Define the dataset
x_data = np.linspace(-10, 10, 50)
y_data = np.linspace(-10, 10, 50)
X, Y = np.meshgrid(x_data, y_data)
Z = target_function(X, Y)  # Target outputs

# Flatten the data for evaluation
inputs = np.array([X.ravel(), Y.ravel()]).T
outputs = Z.ravel()

# Step 3: Define the GEP primitive set
pset = gep.PrimitiveSet('main', input_names=['x', 'y'])
pset.add_function(operator.add, 2)
pset.add_function(operator.mul, 2)
pset.add_constant_terminal(3)

# Step 4: Define the fitness and individual
if not hasattr(creator, "FitnessMax"):
    creator.create("FitnessMax", base.Fitness, weights=(1.0,))
if not hasattr(creator, "Individual"):
    creator.create('Individual', gep.Chromosome, fitness=creator.FitnessMax)

# Define head length and number of genes
h = 10  # Set head length to a suitable value
n_genes = 2  # Adjusted to ensure compatibility with the linker

# Step 5: Define the toolbox
toolbox = gep.Toolbox()

# Register chromosome, population, and compile function
toolbox.register('gene_gen', gep.Gene, pset=pset, head_length=h)
toolbox.register('individual', creator.Individual, gene_gen=toolbox.gene_gen, n_genes=n_genes,
```

```
linker=operator.add)
toolbox.register('population', tools.initRepeat, list, toolbox.individual)
toolbox.register('compile', gep.compile_, pset=pset)

# Define the fitness evaluation function
def evaluate(individual):
    func = toolbox.compile(individual)
    predictions = np.array([func(*input_pair) for input_pair in inputs])
    fitness = -np.mean((outputs - predictions)**2)  # Negative MSE
    return fitness,

toolbox.register('evaluate', evaluate)

# Register selection, mutation, and crossover operators
toolbox.register('select', tools.selRoulette)
toolbox.register('mut_uniform', gep.mutate_uniform, pset=pset, ind_pb=0.1)
toolbox.register('mut_invert', gep.invert, pb=0.1)
toolbox.register('mut_is_ts', gep.is_transpose, pb=0.1)
toolbox.register('mut_ris_ts', gep.ris_transpose, pb=0.1)
toolbox.register('mut_gene_ts', gep.gene_transpose, pb=0.1)
toolbox.register('cx_1p', gep.crossover_one_point, pb=0.4)
toolbox.register('cx_2p', gep.crossover_two_point, pb=0.2)
toolbox.register('cx_gene', gep.crossover_gene, pb=0.1)

# Explicitly set probabilities for the operators in Toolbox.pbs
toolbox.pbs['mut_uniform'] = 0.1  # Set the probability for mut_uniform

# Step 6: Define statistics and Hall of Fame
stats = tools.Statistics(key=lambda ind: ind.fitness.values[0])
stats.register("avg", np.mean)
stats.register("std", np.std)
stats.register("min", np.min)
stats.register("max", np.max)

hof = tools.HallOfFame(3)

# Step 7: Set population size and generations
n_pop = 100
n_gen = 5

pop = toolbox.population(n=n_pop)

print("Starting Genetic Programming Evolution...\n")

# Start evolution
pop, log = gep.gep_simple(pop, toolbox, n_generations=n_gen, n_elites=1,
                          stats=stats, hall_of_fame=hof, verbose=True)
```

```
# Step 8: Output the best individual
best_individual = hof[0]
simplified_solution = gep.simplify(best_individual)

print("\nBest Individual (Chromosome):")
print(best_individual)
print("\nSimplified Solution:")
print(simplified_solution)

# Evaluate the error of the solution
best_func = toolbox.compile(best_individual)
predictions = np.array([best_func(*input_pair) for input_pair in inputs])
mse = np.mean((outputs - predictions)**2)
print(f"\nMean Squared Error of the Best Solution: {mse:.6f}")

# Export the expression tree
rename_labels = {'add': '+', 'mul': '*'}
gep.export_expression_tree(best_individual, rename_labels, file='tree.png')
print("\nExpression tree exported to 'tree.png'.")
```

Output

```
Student Name: Likhith M
USN: 1BM22CS135

Starting Genetic Programming Evolution...

gen     nevals  avg             std             min             max
0       100     -9.38893e+08    7.05021e+09     -6.96414e+10    -2512.38
1       75      -8326.55        22768.3         -153292         -1934.2
2       69      -7262.29        24797.9         -217036         -1621.95
3       64      -256400         1.53122e+06     -1.26277e+07    -0
4       78      -153100         1.3645e+06      -1.36496e+07    -0
5       76      -211820         1.56857e+06     -1.44685e+07    -0

Best Individual (Chromosome):
add(
        mul(x, x),
        mul(y, y)
)

Simplified Solution:
x**2 + y**2

Mean Squared Error of the Best Solution: 0.000000

Expression tree exported to 'tree.png'.
```