VISVESVARAYA TECHNOLOGICAL UNIVERSITY

"JnanaSangama", Belgaum -590014, Karnataka.

LAB RECORD

Bio Inspired Systems (23CS5BSBIS)

Submitted by

Soham Hathi(1BM23CS335)

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
Aug-2025 to Jan-2026

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 **Soham Hathi (1BM23CS335),** who is bonafide student of **B.M.S. College of Engineering.** It is in partial fulfillment for the award of **Bachelor of Engineering in Computer Science and Engineering** of the Visvesvaraya Technological University, Belgaum. The Lab report has been approved as it satisfies the academic requirements of the above mentioned subject and the work prescribed for the said degree.

Rohith Vaidya K Assistant Professor Department of CSE, BMSCE	Dr. Kavitha Sooda Professor & HOD Department of CSE, BMSCE

Index

Github Link:

https://github.com/sohamhathi/BioLAb.git

INDEX

Name Soham Hathi

Standard Section 5'F' Roll No. 1BM23CS335

Subject Bio Inspired System.

SL No.	Date	Title	Page No.	Teacher Sign / Remarks
1	21/08/25	Application of all algorithum		
2	29/08/25	Lab1 - Genecatic Algorithum		
3	29/08/25	Lab2 - Gren Express Algorithum		
4	12/09/25	Lab 3 - Spratical Swarm optimization		
5	10/10/25	Lab-4 - Ant colony optimi-zation.		
6	17/10/25	Lab-5 Cuckoo Searching Algorithum.		
7.	18/10/25	Lab-6 - Gray Wolf optimizer Algorithum.		
8.	07/11/25	Lab-7 Parallel cellular algorithum.		

Program 1

Genetic Algorithm for Optimization Problems

We have a set of jobs that must be completed and a limited amount of resources available to perform them. The challenge is to determine how to assign each job to the available resources in a way that minimizes total completion time, reduces overall cost, or maximizes efficiency. The goal is to find an optimal scheduling strategy under these constraints.

Algorithum:

Bafna Gold
Date: Page:

Grenerotic Algorithum:

Steps:

1) Selecting enloding technique 0 to 31.

2) Select initial population =4'

S# N	Initial population	X value	fitness f(x)=x²	Prob f(x)/Σf(x)	% Prob	Expected count	Actual count from roulette
1	0110	12	144	0.1247	12.147	0.49	1
2	11001	25	625	0.5411	54.11	2.164	2
3	00101	5	25	0.0216	2.16	0.086	0
4	10011	19	361	0.3126	31.25	1.25	1
Sum			1155				
Avg			288.75				
max			625				

3) Select Mating point

S.N	Mating pool	Crossover point	Offspring after crossover	X value	fitness f(x)=x²
1	01100	4	01101	13	169
2	11001		11000	24	576
3	11001	2	11011	27	729
4	10011		10001	17	289
Sum					
Avg					
Max			729		

4) Cross over: Random 4 & 2

Max - 729

5) Mutation

S.N	Offspring after crossover	Mutation Chromosom for flipping	Offspring after mutation	X value	Fitness $f(x)=x^2$
1	01101	10000	11101	29	841
2	11000	00000	11000	24	576
3	11011	00000	11011	27	729
4	10001	00101	10100	20	400
Sum.					2546
Avg					636.5
max					841

Pseudocode for ~~generation~~ generatic Algorithum

Function

Function GeneticAlgorithm (population_size, generation,

population = Initizationpopulation (population_size)

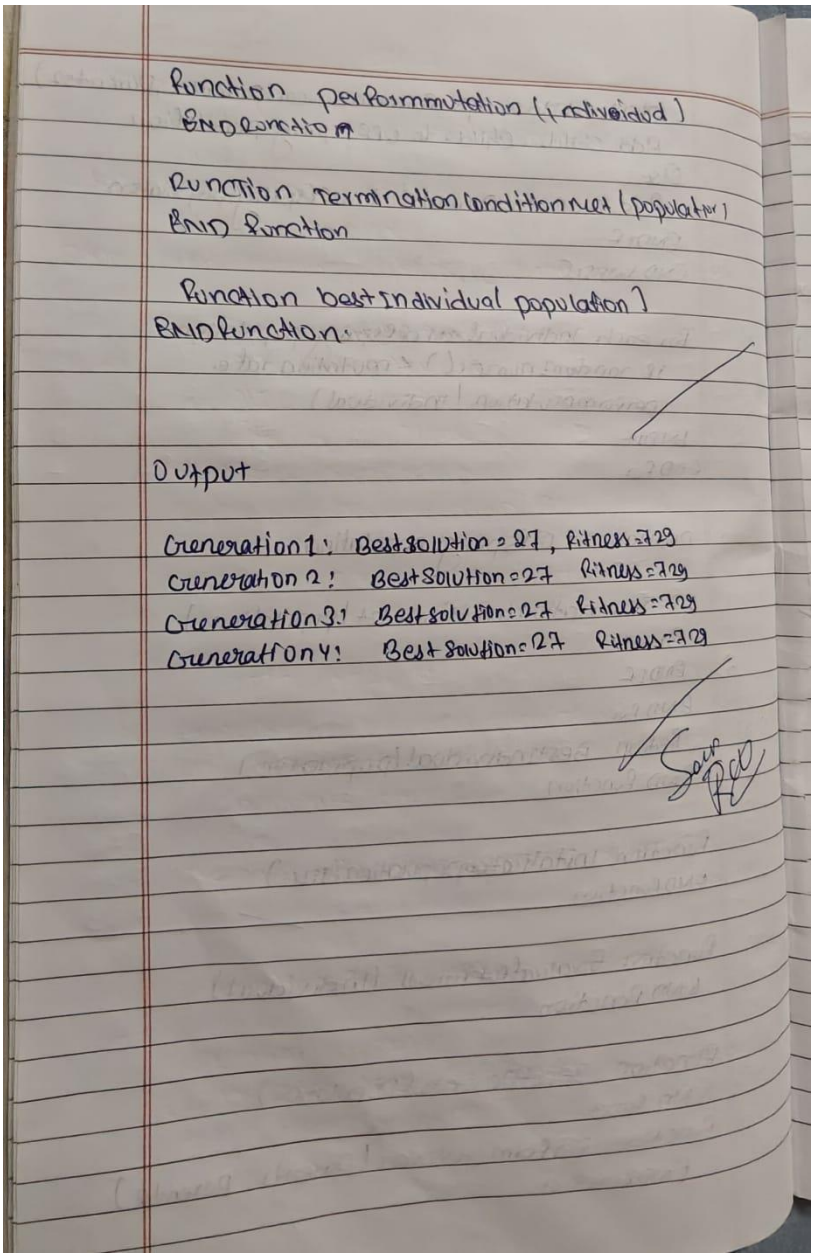
For each generation from 1 TO generation
For each individual in population
individual.fitness = Evaluate fitness (individual)
END for

parents = selectparents(population);

Off_spring_population = EMPTY_LIST
WHILE size (Off_spring_population) < population_size
parents1, parents2 = Chooserootparents (parents)
IF random_number() < crossover_rate

Bafna Gold
Date: Page:

```
    Child1, child2 = performcrossover(parent1, parent2)
    Add child1, child2 to OFF_spring_population
  Else
    Add parent1, parent2 to offspring_population
  ENDIF
END WHILE

For each individual in offspring_population
  If random_number() < mutation rate.
    performMutation (individual)
ENDIF
END For

population = offspring_population

If Termination conditionMet (population)
  Break
  ENDIF
END For
  Return BestIndividual(population)
END Function

Function Initialpopulation(size)
END Function

Function Evaluatefitness (individual)
  END Function

Function SelectParents (population)
END Funct.
Function performcrossover (parent1, parent2)
  END Function.
```



Code:

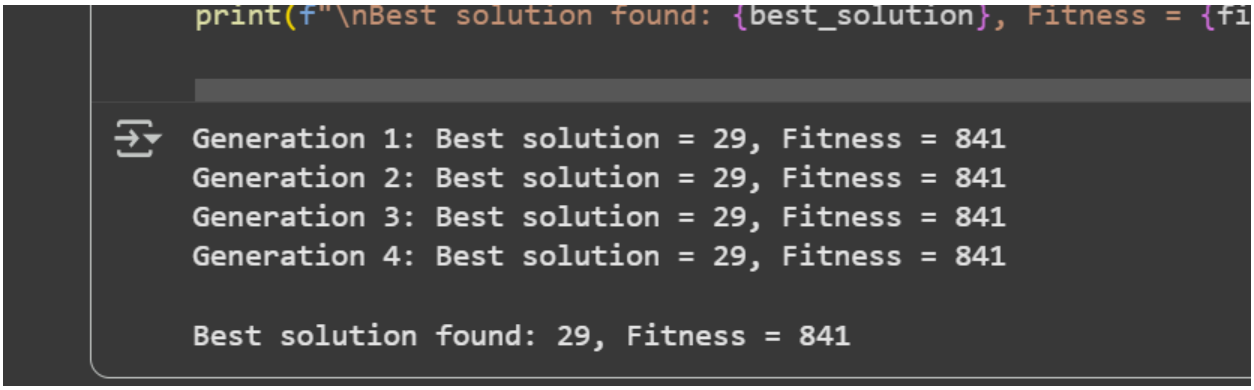
```
import random

def fitness(x): return x**2

def create_population(pop_size, lower_bound, upper_bound):
```

```
 population = [random.randint(lower_bound, upper_bound) for _ in range(pop_size)]

 return population

def selection(population):

 tournament_size = 3

 selected = random.sample(population, tournament_size)

 selected = sorted(selected, key=fitness, reverse=True)

 return selected[0]

def to_binary_string(number, bits=32):

 """Converts an integer to its binary string representation, handling negative numbers."""

if number < 0: return '-' + bin(abs(number))[2:].zfill(bits)

else:

 return bin(number)[2:].zfill(bits)

def from_binary_string(binary_string):

"""Converts a binary string representation back to an integer, handling negative numbers."""

 if binary_string.startswith('-'): return -int(binary_string[1:], 2) else: return int(binary_string, 2)

def crossover(parent1, parent2):

binary_parent1 = to_binary_string(parent1) binary_parent2 = to_binary_string(parent2)

# Ensure crossover point is at least 1 and not beyond the length of the binary string
crossover_point = random.randint(1, max(1, len(binary_parent1.lstrip('-')) - 1))


child1_binary = binary_parent1[:crossover_point] + binary_parent2[crossover_point:]
child2_binary = binary_parent2[:crossover_point] + binary_parent1[crossover_point:]

child1 = from_binary_string(child1_binary)
child2 = from_binary_string(child2_binary)

return child1, child2


def mutation(child, mutation_rate, lower_bound, upper_bound):
```

```
if random.random() < mutation_rate:

  binary_child = to_binary_string(child) # Avoid mutating the sign bit

   mutation_point = random.randint(1, len(binary_child) - 1)

if binary_child.startswith('-')

Else

 random.randint(0, len(binary_child) - 1)

mutated_child_list = list(binary_child) mutated_child_list[mutation_point] = '1'

 if mutated_child_list[mutation_point] == '0'

else '0' mutated_child = ''.join(mutated_child_list)

child = from_binary_string(mutated_child)

return max(lower_bound, min(child, upper_bound))


def genetic_algorithm(pop_size, generations, mutation_rate, lower_bound, upper_bound):

population = create_population(pop_size, lower_bound, upper_bound)

for generation in range(generations):
    new_population = []

    for _ in range(pop_size // 2):
        parent1 = selection(population)
        parent2 = selection(population)

        child1, child2 = crossover(parent1, parent2)
        child1 = mutation(child1, mutation_rate, lower_bound, upper_bound)
        child2 = mutation(child2, mutation_rate, lower_bound, upper_bound)

        new_population.extend([child1, child2])

    population = new_population

    best_solution = max(population, key=fitness)
    print(f"Generation {generation + 1}:

 Best solution = {best_solution}, Fitness = {fitness(best_solution)}")
```

```
return max(population, key=fitness)


pop_size = 5 generations = 4 mutation_rate = 0.01 lower_bound = 0 upper_bound = 31

best_solution = genetic_algorithm(pop_size, generations, mutation_rate, lower_bound, upper_bound)
print(f"\nBest solution found: {best_solution}, Fitness = {fitness(best_solution)}")
```

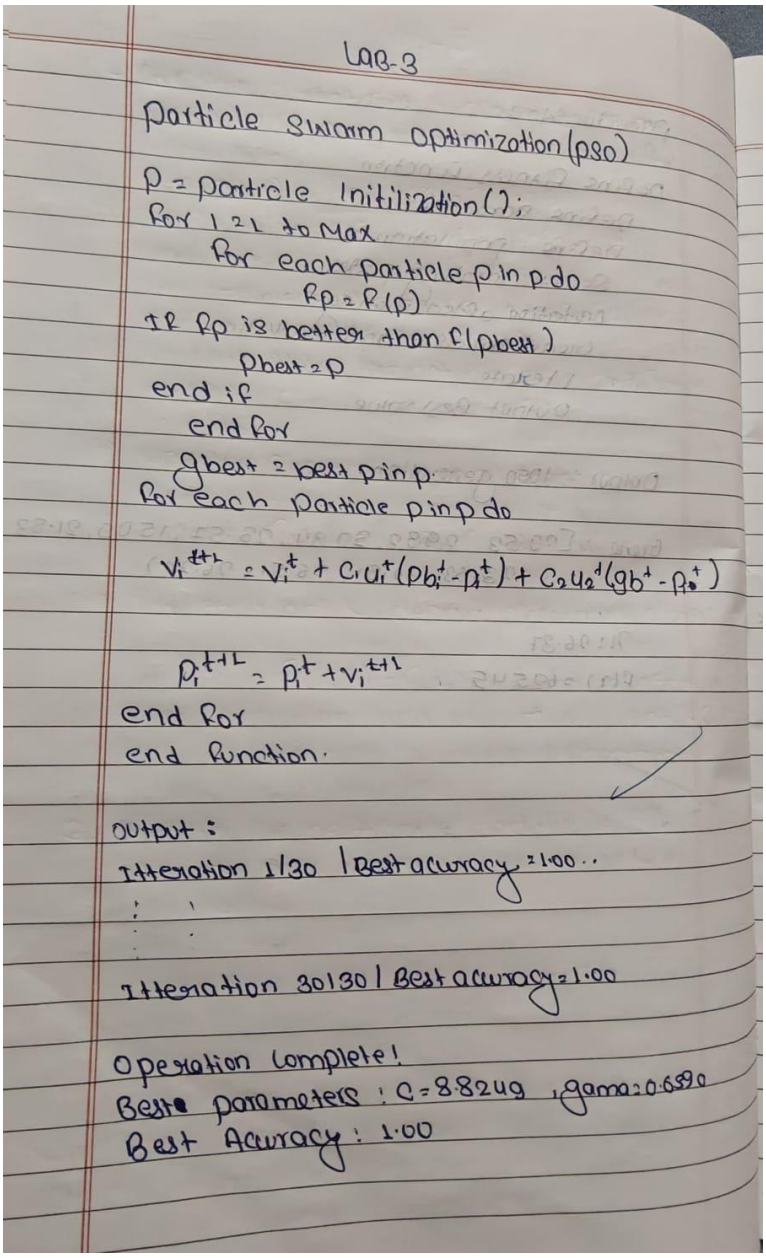
Output:



Program 2

Particle Swarm Optimization for Function Optimization Portfolio Optimization (Selecting assets) using Particle Swarm Optimization is about choosing how much money to allocate to different assets (stocks, bonds, etc.) to maximize expected return while minimizing risk (variance).

Algorithum:



Code:

```
import random


import numpy as np

from sklearn import datasets from sklearn.model_selection
```

import train_test_split from sklearn.svm

import SVC from sklearn.metrics

import accuracy_score

iris = datasets.load_iris() X_train, X_test, y_train, y_test = train_test_split(iris.data, iris.target, test_size=0.3, random_state=42)

def fitness_function(params):

C, gamma = params model = SVC(C=C, gamma=gamma)

model.fit(X_train, y_train) y_pred = model.predict(X_test) accuracy = accuracy_score(y_test, y_pred)

return 1 - accuracy # PSO minimizes this

class Particle:

def **init**(self, bounds):

C, gamma ranges self.position = [random.uniform(bounds[0][0], bounds[0][1]), random.uniform(bounds[1][0], bounds[1][1])]

self.velocity = [random.uniform(-1, 1),

random.uniform(-1, 1)]

self.best_pos = list(self.position) self.best_val = fitness_function(self.position)

```
def update_velocity(self, global_best_pos, w=0.5, c1=1.5, c2=1.5):
    for i in range(len(self.velocity)):
        r1, r2 = random.random(), random.random()
        cognitive = c1 * r1 * (self.best_pos[i] - self.position[i])
        social = c2 * r2 * (global_best_pos[i] - self.position[i])
        self.velocity[i] = w * self.velocity[i] + cognitive + social

def update_position(self, bounds):
    for i in range(len(self.position)):
        self.position[i] += self.velocity[i]
        # Keep position in bounds
```

```
        self.position[i] = max(bounds[i][0], min(bounds[i][1],
self.position[i]))
```

def particle_swarm_optimization(num_particles=20, max_iter=30): bounds = [(0.1, 100), (0.0001, 1)] # (C range), (gamma range) swarm = [Particle(bounds) for _ in range(num_particles)]

```
global_best = min(swarm, key=lambda p: p.best_val)
global_best_pos = list(global_best.best_pos)
global_best_val = global_best.best_val

for iteration in range(max_iter):
    for particle in swarm:
        fitness = fitness_function(particle.position)
        if fitness < particle.best_val:
            particle.best_val = fitness
            particle.best_pos = list(particle.position)

        if fitness < global_best_val:
            global_best_val = fitness
            global_best_pos = list(particle.position)

    for particle in swarm:
        particle.update_velocity(global_best_pos)
        particle.update_position(bounds)

    print(f"Iteration {iteration+1}/{max_iter} | Best Accuracy = {(1 -
global_best_val):.4f}")

return global_best_pos, global_best_val
```

if **name** == "**main**":

best_pos, best_val = particle_swarm_optimization()

print("\n ✅ Optimization complete!")

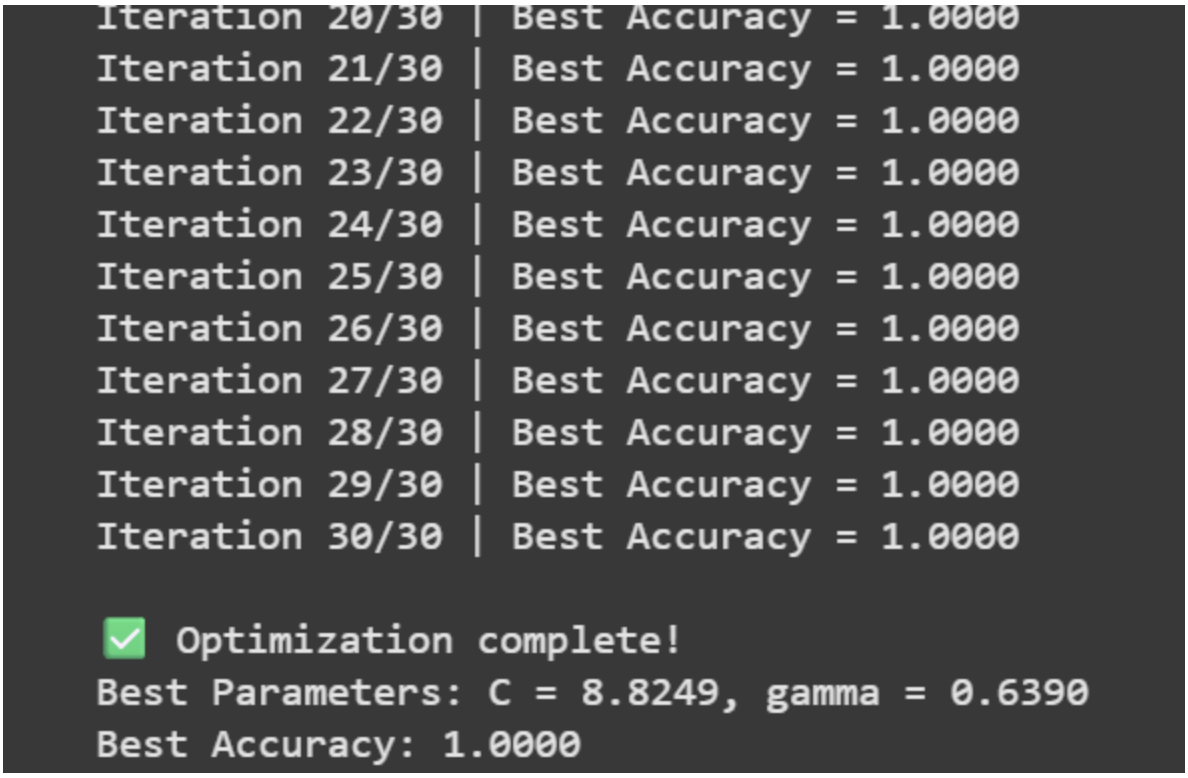
print(f"Best Parameters:

C = {best_pos[0]:.4f}, gamma = {best_pos[1]:.4f}")

print(f"Best Accuracy: {(1 - best_val):.4f}")

Output:

```
Iteration 1/30 | Best Accuracy = 1.0000
Iteration 2/30 | Best Accuracy = 1.0000
Iteration 3/30 | Best Accuracy = 1.0000
Iteration 4/30 | Best Accuracy = 1.0000
Iteration 5/30 | Best Accuracy = 1.0000
Iteration 6/30 | Best Accuracy = 1.0000
Iteration 7/30 | Best Accuracy = 1.0000
Iteration 8/30 | Best Accuracy = 1.0000
Iteration 9/30 | Best Accuracy = 1.0000
Iteration 10/30 | Best Accuracy = 1.0000
Iteration 11/30 | Best Accuracy = 1.0000
Iteration 12/30 | Best Accuracy = 1.0000
Iteration 13/30 | Best Accuracy = 1.0000
Iteration 14/30 | Best Accuracy = 1.0000
Iteration 15/30 | Best Accuracy = 1.0000
Iteration 16/30 | Best Accuracy = 1.0000
Iteration 17/30 | Best Accuracy = 1.0000
Iteration 18/30 | Best Accuracy = 1.0000
Iteration 19/30 | Best Accuracy = 1.0000
Iteration 20/30 | Best Accuracy = 1.0000
```



Program 3

Ant Colony Optimization for the Traveling Salesman Problem Ant Colony Optimization (ACO) for the Vehicle Routing Problem (VRP): It involves finding optimal routes for multiple vehicles to deliver goods to a set of customers from a central depot.

Algorithum:

Lab-4

Bafna Gold
Date: Page:

Ant Colony Optimization (ACO)

Input :
- A set of cities with know condition
- Parameters number of ants, alpha, beta, rho, no of itteration, intial ~~pro~~ pheromone value.

Output :
- The best ~~to~~ (shortest) route visiting all cities exactly once and returning to the start

Initialize :
- Calculate the Distance matrix D between each pair of cities.
- Initilize pheromone level $\tau(i,j) = c_0$ for all edges
- Calculate the heuristic cost $\eta(i,j) = 1/D(i,j)$ For all edge (expect where distance is zero)

2. For itteration = 1 to N:

a. For each ant k = 1 to m :
   i. place ant k on a randomly selected start city.
   ii) Initialize the ant's tabu list (visited city) with start city.

(or When the ant has not visited

- For the current city i, calculate the probability $p(i,j)$ of moving to an unvisited city j as:

$$p(i,j) = \frac{[\tau(i,j)]^{\alpha} \cdot [\eta(i,j)]^{\beta}}{\sum_{l \in \text{allowed}} [\tau(i,l)]^{\alpha} \cdot [\eta(i,l)]^{\beta}}$$

- Select the next city probabilistically according to $p(i,j)$.
- Move the ant to city j and add j to the tour.

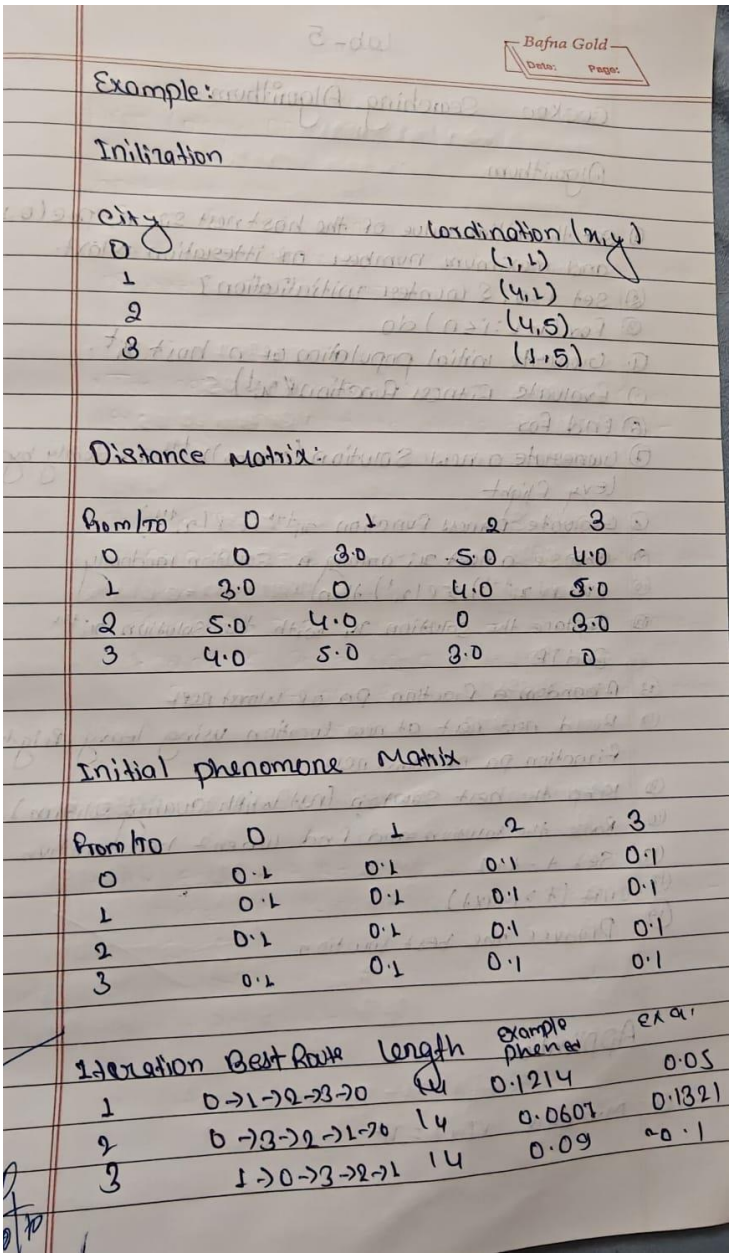
(b) Update pheromones on all edges:

$$\tau(i,j) = (1-\rho) \cdot \tau(i,j)$$

(c) For each ant k, deposit pheromone along its path:

$\tau(i,j) = \tau(i,j) + \frac{1}{L_k}$ for each edge (i,j) in ant k's tour.

3. Return the best route and its length found during all iterations.



Example:

Initialization

city	cordination (x,y)
0	(1,1)
1	(4,1)
2	(4,5)
3	(1,5)

Distance Matrix:

From/To	0	1	2	3
0	0	3.0	5.0	4.0
1	3.0	0	4.0	5.0
2	5.0	4.0	0	3.0
3	4.0	5.0	3.0	0

Initial phenomone Matrix

From/To	0	1	2	3
0	0.1	0.1	0.1	0.1
1	0.1	0.1	0.1	0.1
2	0.1	0.1	0.1	0.1
3	0.1	0.1	0.1	0.1

Iteration	Best Route	length	example phenomone	exa.
1	0→1→2→3→0	14	0.1214	0.05
2	0→3→2→1→0	14	0.0607	0.1321
3	1→0→3→2→1	14	0.09	~0.1

Code:

```
import numpy as np

cities = np.array([ [1, 1], [4, 1], [4, 5], [1, 5] ])

num_cities = len(cities)

num_ants = 10 alpha = 1 # pheromone importance beta = 5 # heuristic importance rho = 0.5 # evaporation rate pheromone_init = 0.1 iterations = 20 # Reduced for readability Q = 1 # pheromone deposit factor
```

```
def distance_matrix(cities): dist = np.zeros((num_cities, num_cities)) for i in
range(num_cities): for j in range(num_cities): dist[i][j] = np.linalg.norm(cities[i] - cities[j])
return dist

dist_matrix = distance_matrix(cities)

heuristic = 1 / (dist_matrix + 1e-10)

pheromone = np.ones((num_cities, num_cities)) * pheromone_init

def select_next_city(current_city, visited):

 probabilities = [] for city in range(num_cities):

 if city not in visited:

 tau = pheromone[current_city][city] ** alpha

eta = heuristic[current_city][city] ** beta probabilities.append(tau * eta)

else:

 probabilities.append(0)

probabilities = np.array(probabilities)

probabilities_sum = probabilities.sum()

if probabilities_sum == 0:

 # If no options,

choose randomly from unvisited unvisited = [c for c in range(num_cities)

if c not in visited] return np.random.choice(unvisited)

probabilities /= probabilities_sum

return np.random.choice(range(num_cities), p=probabilities)

def construct_solution():

solutions = []
```

lengths = [] for _ in range(num_ants):

visited = []

start_city = np.random.randint(num_cities) visited.append(start_city)

```
   while len(visited) < num_cities:
        current_city = visited[-1]
        next_city = select_next_city(current_city, visited)
        visited.append(next_city)

    visited.append(visited[0])  # return to start
    solutions.append(visited)

    # Calculate route length
    length = 0
    for i in range(len(visited) - 1):
        length += dist_matrix[visited[i]][visited[i + 1]]
    lengths.append(length)

return solutions, lengths
```

def update_pheromone(solutions, lengths): global pheromone # Evaporate pheromone pheromone = (1 - rho) * pheromone

```
# Deposit pheromone
for i, route in enumerate(solutions):
    length = lengths[i]
    for j in range(len(route) - 1):
        city_i = route[j]
        city_j = route[j + 1]
        pheromone[city_i][city_j] += Q / length
        pheromone[city_j][city_i] += Q / length  # symmetric TSP
```

best_route = None best_length = float('inf')

for iteration in range(iterations): solutions, lengths = construct_solution() update_pheromone(solutions, lengths)

```
min_length = min(lengths)
min_index = lengths.index(min_length)

if min_length < best_length:
    best_length = min_length
    best_route = solutions[min_index]

# Detailed iteration output
print(f"Iteration {iteration + 1}:")
print(f"  Best route this iteration: {solutions[min_index]}")
print(f"  Length of best route this iteration: {min_length:.4f}")
print(f"  Global best route so far: {best_route}")
print(f"  Length of global best route so far: {best_length:.4f}\n")
```

print("Final Best Route:", best_route) print("Final Best Route Length:", best_length)

Output:

```
Iteration 18:
  Best route this iteration: [3, np.int64(0), np.int64(1), np.int64(2), 3]
  Length of best route this iteration: 14.0000
  Global best route so far: [2, np.int64(1), np.int64(0), np.int64(3), 2]
  Length of global best route so far: 14.0000

Iteration 19:
  Best route this iteration: [1, np.int64(0), np.int64(3), np.int64(2), 1]
  Length of best route this iteration: 14.0000
  Global best route so far: [2, np.int64(1), np.int64(0), np.int64(3), 2]
  Length of global best route so far: 14.0000

Iteration 20:
  Best route this iteration: [0, np.int64(1), np.int64(2), np.int64(3), 0]
  Length of best route this iteration: 14.0000
  Global best route so far: [2, np.int64(1), np.int64(0), np.int64(3), 2]
  Length of global best route so far: 14.0000

Final Best Route: [2, np.int64(1), np.int64(0), np.int64(3), 2]
Final Best Route Length: 14.0
```

```
Iteration 12:
  Best route this iteration: [2, np.int64(1), np.int64(0), np.int64(3), 2]
  Length of best route this iteration: 14.0000
  Global best route so far: [2, np.int64(1), np.int64(0), np.int64(3), 2]
  Length of global best route so far: 14.0000

Iteration 13:
  Best route this iteration: [1, np.int64(0), np.int64(3), np.int64(2), 1]
  Length of best route this iteration: 14.0000
  Global best route so far: [2, np.int64(1), np.int64(0), np.int64(3), 2]
  Length of global best route so far: 14.0000

Iteration 14:
  Best route this iteration: [0, np.int64(3), np.int64(2), np.int64(1), 0]
  Length of best route this iteration: 14.0000
  Global best route so far: [2, np.int64(1), np.int64(0), np.int64(3), 2]
  Length of global best route so far: 14.0000

Iteration 15:
  Best route this iteration: [0, np.int64(1), np.int64(2), np.int64(3), 0]
  Length of best route this iteration: 14.0000
  Global best route so far: [2, np.int64(1), np.int64(0), np.int64(3), 2]
  Length of global best route so far: 14.0000

Iteration 16:
  Best route this iteration: [0, np.int64(1), np.int64(2), np.int64(3), 0]
  Length of best route this iteration: 14.0000
  Global best route so far: [2, np.int64(1), np.int64(0), np.int64(3), 2]
  Length of global best route so far: 14.0000

Iteration 17:
  Best route this iteration: [3, np.int64(2), np.int64(1), np.int64(0), 3]
  Length of best route this iteration: 14.0000
  Global best route so far: [2, np.int64(1), np.int64(0), np.int64(3), 2]
  Length of global best route so far: 14.0000

Iteration 18:
  Best route this iteration: [3, np.int64(0), np.int64(1), np.int64(2), 3]
  Length of best route this iteration: 14.0000
  Global best route so far: [2, np.int64(1), np.int64(0), np.int64(3), 2]
```

```
  Length of global best route so far: 14.0000

Iteration 7:
  Best route this iteration: [2, np.int64(1), np.int64(0), np.int64(3), 2]
  Length of best route this iteration: 14.0000
  Global best route so far: [2, np.int64(1), np.int64(0), np.int64(3), 2]
  Length of global best route so far: 14.0000

Iteration 8:
  Best route this iteration: [1, np.int64(0), np.int64(3), np.int64(2), 1]
  Length of best route this iteration: 14.0000
  Global best route so far: [2, np.int64(1), np.int64(0), np.int64(3), 2]
  Length of global best route so far: 14.0000

Iteration 9:
  Best route this iteration: [0, np.int64(1), np.int64(2), np.int64(3), 0]
  Length of best route this iteration: 14.0000
  Global best route so far: [2, np.int64(1), np.int64(0), np.int64(3), 2]
  Length of global best route so far: 14.0000

Iteration 10:
  Best route this iteration: [0, np.int64(1), np.int64(2), np.int64(3), 0]
  Length of best route this iteration: 14.0000
  Global best route so far: [2, np.int64(1), np.int64(0), np.int64(3), 2]
  Length of global best route so far: 14.0000

Iteration 11:
  Best route this iteration: [1, np.int64(0), np.int64(3), np.int64(2), 1]
  Length of best route this iteration: 14.0000
  Global best route so far: [2, np.int64(1), np.int64(0), np.int64(3), 2]
  Length of global best route so far: 14.0000

Iteration 12:
  Best route this iteration: [2, np.int64(1), np.int64(0), np.int64(3), 2]
  Length of best route this iteration: 14.0000
  Global best route so far: [2, np.int64(1), np.int64(0), np.int64(3), 2]
  Length of global best route so far: 14.0000
```

```
Iteration 1:
  Best route this iteration: [2, np.int64(1), np.int64(0), np.int64(3), 2]
  Length of best route this iteration: 14.0000
  Global best route so far: [2, np.int64(1), np.int64(0), np.int64(3), 2]
  Length of global best route so far: 14.0000

Iteration 2:
  Best route this iteration: [0, np.int64(1), np.int64(2), np.int64(3), 0]
  Length of best route this iteration: 14.0000
  Global best route so far: [2, np.int64(1), np.int64(0), np.int64(3), 2]
  Length of global best route so far: 14.0000

Iteration 3:
  Best route this iteration: [3, np.int64(0), np.int64(1), np.int64(2), 3]
  Length of best route this iteration: 14.0000
  Global best route so far: [2, np.int64(1), np.int64(0), np.int64(3), 2]
  Length of global best route so far: 14.0000

Iteration 4:
  Best route this iteration: [1, np.int64(2), np.int64(3), np.int64(0), 1]
  Length of best route this iteration: 14.0000
  Global best route so far: [2, np.int64(1), np.int64(0), np.int64(3), 2]
  Length of global best route so far: 14.0000

Iteration 5:
  Best route this iteration: [1, np.int64(0), np.int64(3), np.int64(2), 1]
  Length of best route this iteration: 14.0000
  Global best route so far: [2, np.int64(1), np.int64(0), np.int64(3), 2]
  Length of global best route so far: 14.0000

Iteration 6:
  Best route this iteration: [3, np.int64(2), np.int64(1), np.int64(0), 3]
  Length of best route this iteration: 14.0000
  Global best route so far: [2, np.int64(1), np.int64(0), np.int64(3), 2]
  Length of global best route so far: 14.0000
```

Program 4

Cuckoo Search (CS) Cuckoo Search Algorithms: We need to maximize the total value of selected items without exceeding the knapsack's weight capacity. Using the Cuckoo Search Algorithm, each solution is a binary vector, new solutions are generated via Lévy flights, and the best feasible solution is iteratively improved while abandoning poor solutions with a probability.

Algorithum:

Lab-5

Cuckoo Searching Algorithm.

Algorithum

① Set the initial value of the host nest size n, $P_a \in (0,1)$ and maximum number of iteration Maxt.
② set t=0. & (counter initialization)
③ For (i=1 : i≤n) do
④ Generate initial population of n host x_i^t.
⑤ Evaluate Fitness function $f(x_i^t)$
⑥ End For
⑦ Generate a new solution (cuckoo) x_i^{t+1} randomly by levy flight
⑧ Evaluate fitness function x_i^{t+1} i.e $f(x_i^{t+1})$
⑨ Choose a nest x_i among n solution randomly
⑩ If $f(x_i^{t+1}) > f(x_i^t)$ then
⑪ Replace the solution x_i with the solution x_i^{t+1}
⑫ End If
⑬ Abandom a fraction p_a of worst nest
⑭ Build new nest at new location using leavy flight fraction p_a of worst nest.
⑮ Keep the best solution (nest with quality solution).
⑯ Rank the solution and find current best solution
⑰ Set t = t+1
⑱ until (t ≥ Maxt)
⑲ Produce the best solution

Application

Q. Minimize $f(x) = x^2$

Bafna Gold
Date: Page:

global minimum at $x = 0$ $f(0) = 0$

Step 1

$x_1 = 4$ $f(x_1) = 16$

$x_2 = -3$ $f(x_2) = 9$

$x_3 = 6$ $f(x_3) = 36$

Best soln

$x_2 = -3$ fitness = 9

Step 2:

$x_{new} = x_{old} + L.levy(x)$

$L = 1$

$x_1^{new} = 4 + (-2) = 2$ $f(2) = 4$

$x_2^{new} = -3 + (1) = -2$ $f(-2) = 4$

$x_3^{new} = 6 + (-4) = 2$ $f(2) = 4$

Best soln $x = 2$ $f(2) = 4$

Step 3

Evaluate and select best

Nest 1 = 2 (fitness 4)

Nest 2 = -2 (fitness 4)

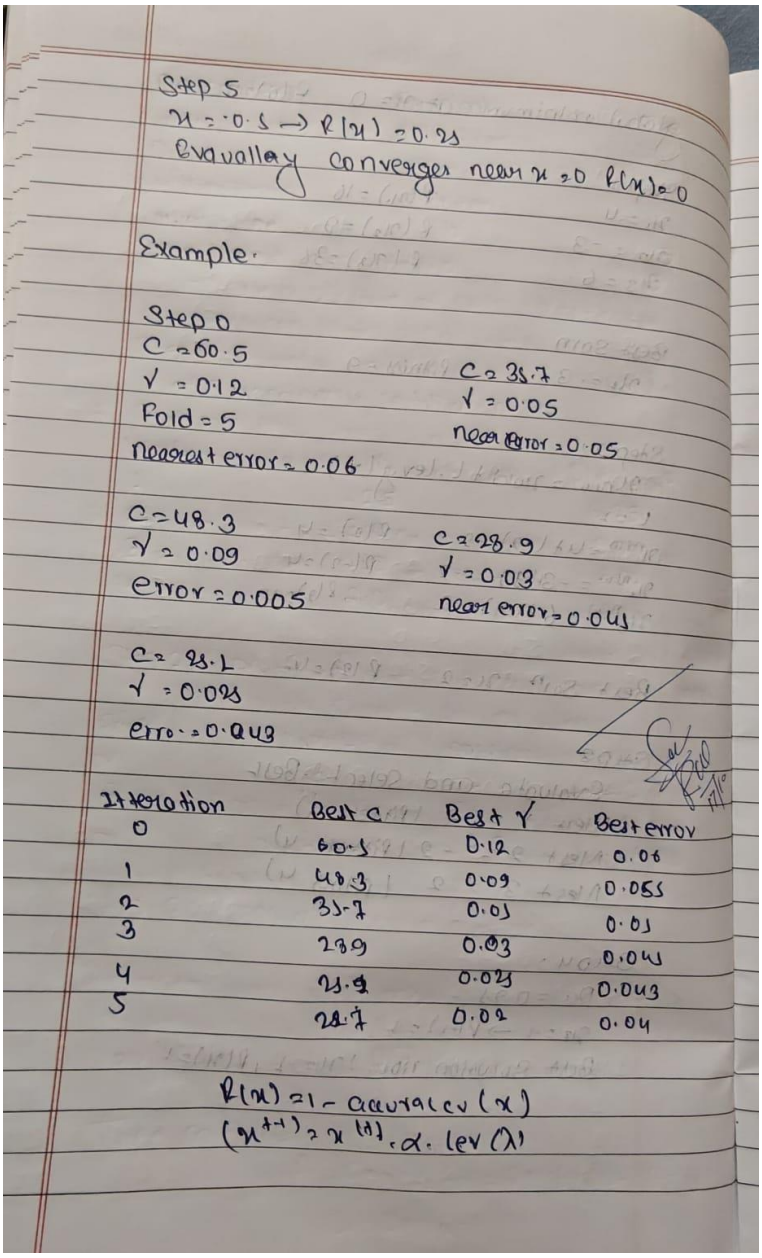
Nest 3 = 2 (fitness 4)

Step 4:

$pa = 0.25$

$x_3 = -1 \rightarrow f(-1) = 1$

Best solution now: $x = -1$, $f(x) = 1$



Step 5
$x = 0.5 \rightarrow R(x) = 0.25$
Evaually converges near $x = 0$ $R(x) = 0$

Example.

Step 0
C = 60.5
γ = 0.12
Fold = 5
Nearest error = 0.06

C = 35.7
γ = 0.05
near error = 0.05

C = 48.3
γ = 0.09
error = 0.005

C = 28.9
γ = 0.03
near error = 0.045

C = 23.1
γ = 0.025
erro = 0.043

Iteration	Best C	Best γ	Best error
0	60.5	0.12	0.06
1	48.3	0.09	0.055
2	35.7	0.05	0.05
3	28.9	0.03	0.045
4	23.1	0.025	0.043
5	28.7	0.02	0.04

$R(x) = 1 - \text{accuracy}(x)$
$(x^{t+1}) = x^{(t)} \cdot \alpha \cdot \text{lev}(\lambda)$

Code:

```
import numpy as np

 import math from sklearn

import datasets from sklearn.model_selection

 import cross_val_score from sklearn.svm
```

import SVC

iris = datasets.load_iris() X = iris.data y = iris.target

def svm_error_rate(params):

C, gamma = params # Make sure parameters are within bounds

if C <= 0 or gamma <= 0: return 1.0 # Max error

```
svm = SVC(C=C, gamma=gamma)
# Use 5-fold cross-validation accuracy
scores = cross_val_score(svm, X, y, cv=5)
error = 1 - scores.mean()  # Minimize error
return error
```

def levy_flight(Lambda, size):

sigma_u = (math.gamma(1 + Lambda) * np.sin(np.pi * Lambda / 2) / (math.gamma((1 + Lambda) / 2) * Lambda * 2 ** ((Lambda - 1) / 2))) ** (1 / Lambda)

u = np.random.normal(0, sigma_u, size) v = np.random.normal(0, 1, size)

step = u / (np.abs(v) ** (1 / Lambda)) return step

def cuckoo_search(objective_func, n=20, dim=2, lb=[0.1, 0.0001],

ub=[100, 1], pa=0.25, max_iter=100):

lb = np.array(lb) ub = np.array(ub)

```
nests = lb + (ub - lb) * np.random.rand(n, dim)
fitness = np.array([objective_func(nest) for nest in nests])

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

Lambda = 1.5

for iteration in range(max_iter):
```

```
    for i in range(n):
        step = levy_flight(Lambda, dim)
        step_size = 0.01 * step * (nests[i] - best_nest)
        new_nest = nests[i] + step_size
        new_nest = np.clip(new_nest, lb, ub)
        new_fitness = objective_func(new_nest)
        if new_fitness < fitness[i]:
            nests[i] = new_nest
            fitness[i] = new_fitness
            if new_fitness < best_fitness:
                best_fitness = new_fitness
                best_nest = new_nest.copy()

    # Abandon worst nests and create new ones
    K = np.random.rand(n, dim) > pa
    new_nests = lb + (ub - lb) * np.random.rand(n, dim)
    nests = nests * K + new_nests * (1 - K)
    fitness = np.array([objective_func(nest) for nest in nests])
    current_best_idx = np.argmin(fitness)
    if fitness[current_best_idx] < best_fitness:
        best_fitness = fitness[current_best_idx]
        best_nest = nests[current_best_idx].copy()

    if iteration % 10 == 0 or iteration == max_iter - 1:
        print(f"Iteration {iteration}: Best error rate =
{best_fitness:.4f}")

return best_nest, best_fitness


best_params, best_error = cuckoo_search(svm_error_rate)

print("\nBest hyperparameters found:")

 print(f"C = {best_params[0]:.4f}, gamma = {best_params[1]:.6f}")

print(f"Cross-validated error rate = {best_error:.4f}")
```

Output:

```
Iteration 0: Best error rate = 0.0267
Iteration 10: Best error rate = 0.0133
Iteration 20: Best error rate = 0.0133
Iteration 30: Best error rate = 0.0133
Iteration 40: Best error rate = 0.0133
Iteration 50: Best error rate = 0.0133
Iteration 60: Best error rate = 0.0133
Iteration 70: Best error rate = 0.0133
Iteration 80: Best error rate = 0.0133
Iteration 90: Best error rate = 0.0133
Iteration 99: Best error rate = 0.0133

Best hyperparameters found:
C = 13.8572, gamma = 0.024144
Cross-validated error rate = 0.0133
```

Program 5

Grey Wolf Optimizer (GWO) Using the Grey Wolf Optimizer (GWO), we aim to find the shortest, obstacle-free path by modeling the search agents (wolves) to iteratively converge toward the best position (path node) in the environment. The algorithm simulates the grey wolves' hunting hierarchy and encircling behavior to efficiently navigate the space from the start point.

Algorithum:

Bafna Gold
Date: Page:

Grey Wolf optimizer (GWO)

Algorithum:

1. Initialize population x_i ($i = 1, 2, \ldots, N$) randomly.
2. Evaluate fitness of each x_i
3. Identify α (best), β (second best), δ (third best)
4. While iteration < max.iter:
   For each wolf x_i:
   Update coefficients A and C
   update position using:
   $D_\alpha = |C_1 * x_\alpha - x_i|$
   $D_\beta = |C_2 * x_\beta - x_i|$
   $D_\delta = |C_3 * x_\delta - x_i|$
   $x_1 = x_\alpha - A_1 * D_\alpha$
   $x_2 = x_\beta - A_2 * D_\beta$
   $x_3 = x_\delta - A_3 * D_\delta$
   $x_i(new) = (x_1 + x_2 + x_3)/3$
   End for
   Update fitness α, β, δ
   Decrease a linear from 2 to 0
   End while
5. Return x_α as best solution

Example:

$f(x) = x^2$

$x = 0$, $f(0) = 0$

Parameters	Symbol	Value
Number of Wolves	N	4
Dimension	dim	1
Max iteration	Max.iter	1
Lower bound	lb	5
Upper bound	ub	-5
		5

Step 1:

Let's randomly initilization 4

Wolf	x_i	$f(x_i) = x_i^2$
1	-2.5	6.25
2	3.0	9.00
3	-1.0	1.00
4	2.0	4.00

Step 2: α, β, δ

Rank	Wolf	position	Fitness
α	3	-1.0	1.00
β	4	2.0	4.00
δ	1	-2.5	6.25

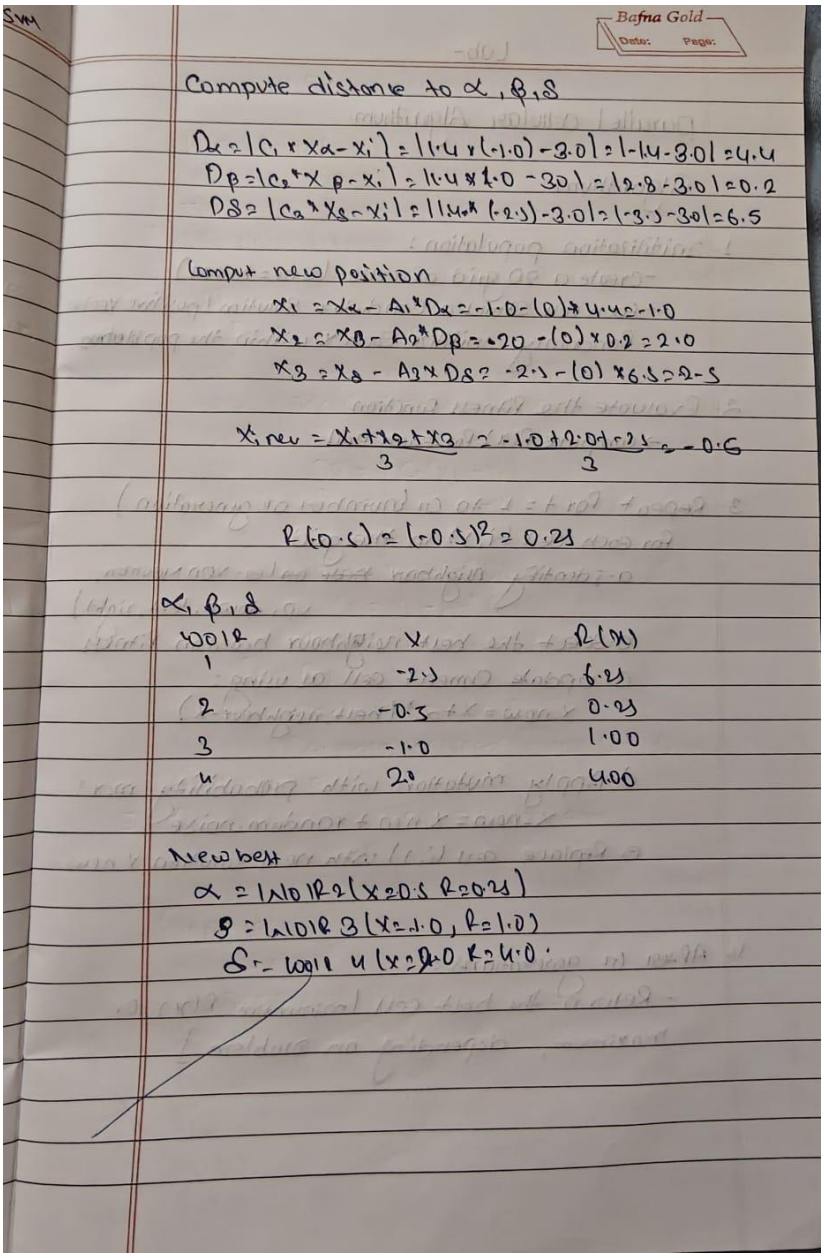
Step 3: Generate random numbers

Parameters	Symbol	value
r_1	0.3	
r_2	0.7	

Then calculate.

$$A = 2 * \alpha * r_1 - \alpha = 2 * 1.6 \times 0.3 - 1.6 = 0$$

$$C_1 = 2 * r_2 = 1.4$$

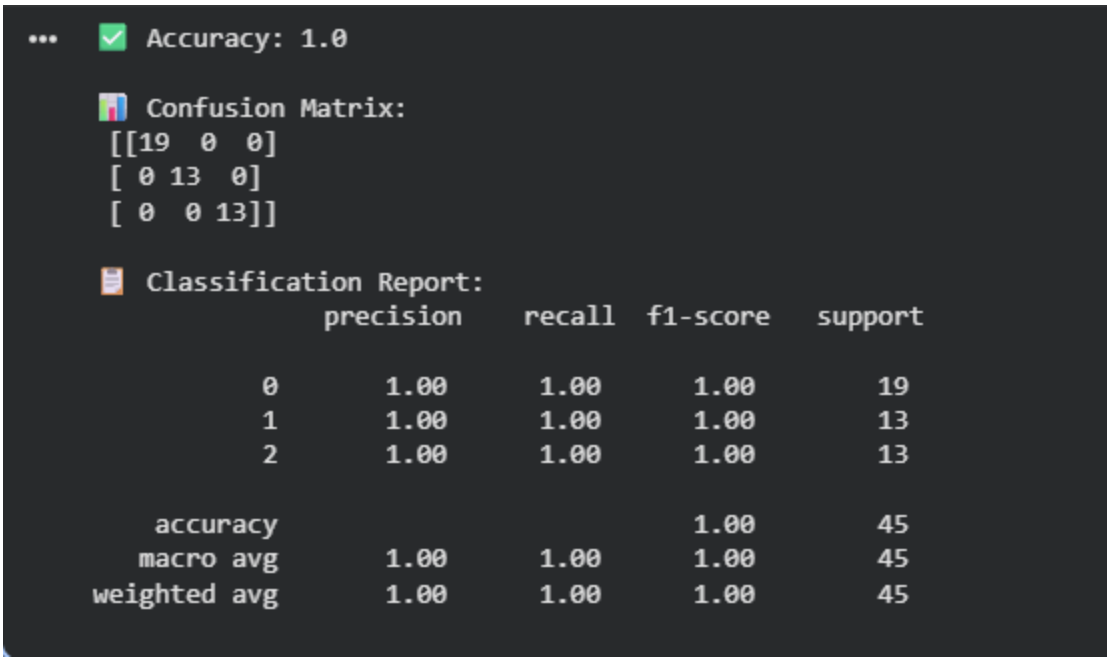


Code:

```
import numpy as np

from sklearn import datasets from sklearn.model_selection

 import train_test_split from sklearn.preprocessing i

mport StandardScaler from sklearn.svm
```

```
 import SVC from sklearn.metrics

import accuracy_score, confusion_matrix, classification_report

iris = datasets.load_iris() X = iris.data # features y = iris.target # labels

X_train, X_test, y_train, y_test = train_test_split( X, y, test_size=0.3, random_state=42 )

scaler = StandardScaler() X_train = scaler.fit_transform(X_train) X_test =
scaler.transform(X_test)

model = SVC(kernel='rbf', C=1.0, gamma='scale')

model.fit(X_train, y_train)

y_pred = model.predict(X_test)

print(" ✅ Accuracy:", accuracy_score(y_test, y_pred))

 print("\n 📊 Confusion Matrix:\n", confusion_matrix(y_test, y_pred))

 print("\n 📋 Classification Report:\n", classification_report(y_test, y_pred))
```

Output:



Program 6

Parallel Cellular Algorithms and Programs The task is to perform edge detection or noise reduction in an image using Parallel Cellular Automata (PCA), where each pixel (cell) interacts with its neighbors to enhance edges or reduce noise iteratively.

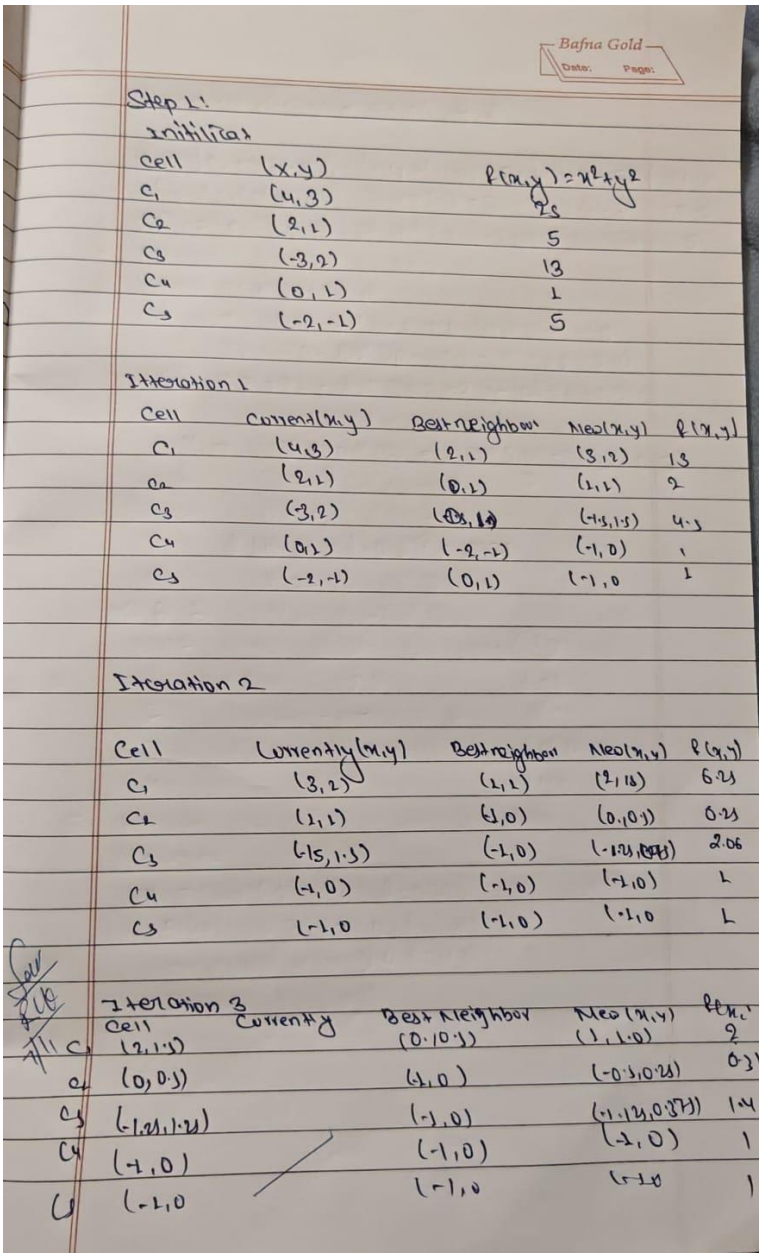
Algorithum:

Lab-

Parallel Cellular Algorithm.

Algorithm

1. Initilization population:
   - Create a 2D grid of cells (size = N x M)
   - Each cell stores a candidate solution (position x)
   - Randomly initilize each x within the population
2. Evaluate the fitness function
   → compute f(x) for each cell
3. Repeat for t = 1 to G (number of generation)
   for each cell (i, j) in the grid
   a. Identify neighbour ~~base on~~ (eg: von Neuman, up, down, left, right)
   b. select the best neighbour based on fitness
   c. Update current cell as using:
      x_new = x + α * (best_neighbour - x)
   d. Apply mutation with probability m.r
      x_new = x_new + random_noise
   e. Replace cell (i, j) with new solution x_new
4. After G generations:
   - Return the best cell (minimum f(x) or maximum, depending on problem)



Step 1:

Initialization

Cell	(x,y)	f(x,y) = x² + y²
C_1	(4,3)	25
C_2	(2,1)	5
C_3	(-3,2)	13
C_4	(0,1)	1
C_5	(-2,-1)	5

Iteration 1

Cell	Current(x,y)	Best neighbor	Neo(x,y)	f(x,y)
C_1	(4,3)	(2,1)	(3,2)	13
C_2	(2,1)	(0,1)	(1,1)	2
C_3	(-3,2)	(-1.5,1.5)	(-1.5,1.5)	4.5
C_4	(0,1)	(-2,-1)	(-1,0)	1
C_5	(-2,-1)	(0,1)	(-1,0	1

Iteration 2

Cell	Currently(x,y)	Best neighbor	Neo(x,y)	f(x,y)
C_1	(3,2)	(2,1)	(2,1.5)	6.25
C_2	(1,1)	(1,0)	(0.5,0.5)	0.25
C_3	(-1.5,1.5)	(-1,0)	(-1.25,0.75)	2.06
C_4	(-1,0)	(-1,0)	(-1,0)	1
C_5	(-1,0	(-1,0)	(-1,0	1

Iteration 3

Cell	Currently	Best Neighbor	Neo(x,y)	f(x,y)
C_1	(2,1.5)	(0.5,0.5)	(1,1.0)	2
C_2	(0,0.5)	(1,0)	(-0.5,0.25)	0.3
C_3	(-1.25,1.25)	(-1,0)	(-1.125,0.875)	1.4
C_4	(-1,0)	(-1,0)	(-1,0)	1
C_5	(-1,0	(-1,0	(-1,0	1

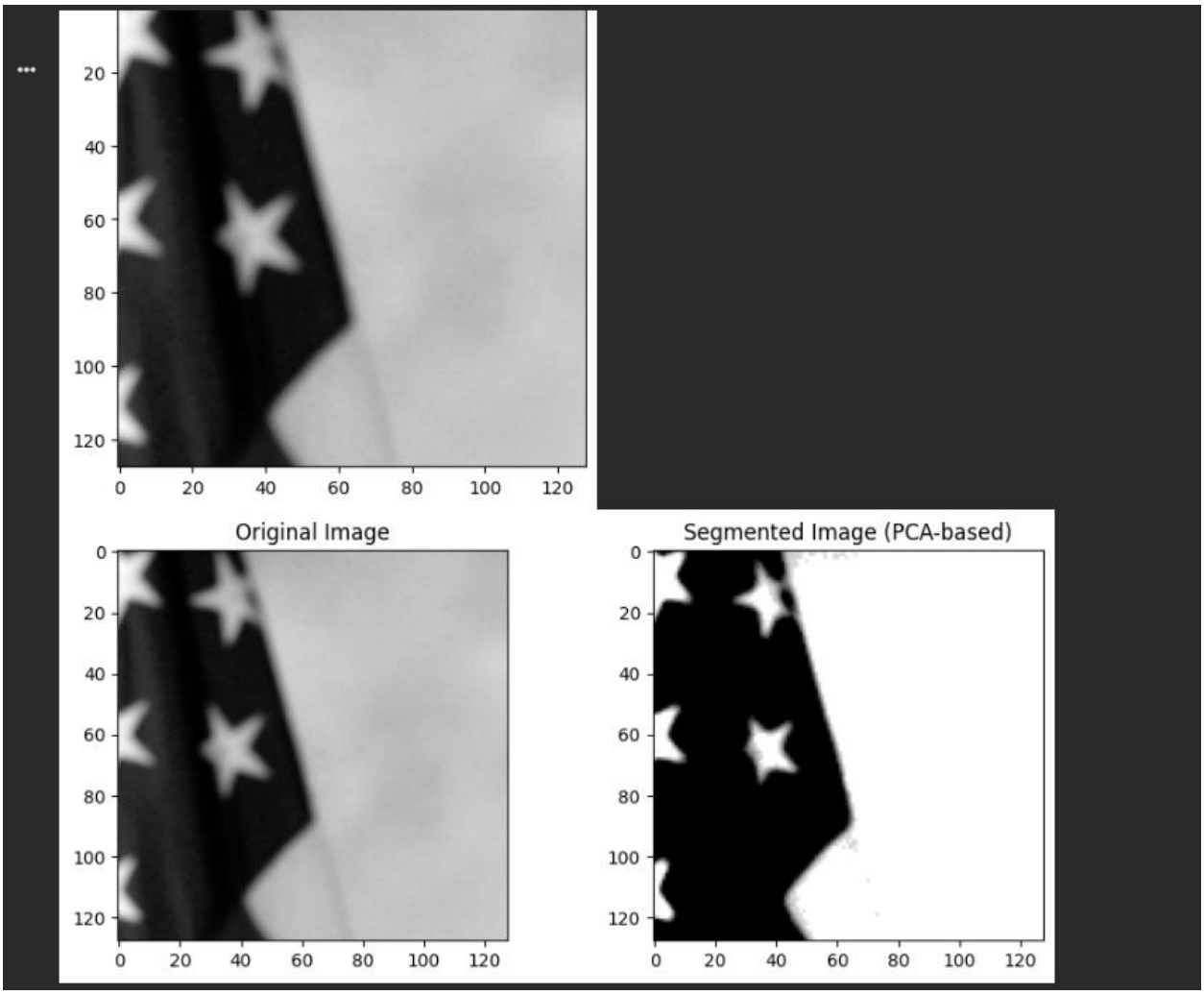
Code:

```
import numpy as np
 import matplotlib.pyplot as plt
from skimage import data, color, img_as_float from scipy.spatial.distance
 import cdist
```

```
image = color.rgb2gray(img_as_float(data.astronaut())) # sample image

 image = image[0:128, 0:128] # resize for speed

plt.imshow(image, cmap='gray') plt.title("Original Image") plt.show()

grid_size = 10 # 10x10

clusters clusters = np.random.uniform(0, 1, (grid_size, grid_size)) # cluster

centers num_generations = 50 learning_rate = 0.3 mutation_rate = 0.1

for gen in range(num_generations): new_clusters = np.copy(clusters)

 for i in range(grid_size):

for j in range(grid_size):

# Define neighborhood (Von Neumann)

neighbors = [((i-1)%grid_size, j), ((i+1)%grid_size, j), (i, (j-1)%grid_size), (i, (j+1)%grid_size)]

        # Compute average of neighbor cluster centers
         neighbor_values = [clusters[n] for n in neighbors]
         avg_neighbor = np.mean(neighbor_values)

         # Update cell towards neighbor average
         new_val = clusters[i, j] + learning_rate * (avg_neighbor -
clusters[i, j])

         # Mutation
         if np.random.rand() < mutation_rate:
             new_val += np.random.normal(0, 0.05)

         new_clusters[i, j] = np.clip(new_val, 0, 1)
clusters = new_clusters


flat_clusters = clusters.flatten().reshape(-1, 1)

flat_image = image.flatten().reshape(-1, 1)
```

```
distances = cdist(flat_image, flat_clusters)
 labels = np.argmin(distances, axis=1)
segmented_image = clusters.flatten()[labels].reshape(image.shape)
plt.figure(figsize=(10,4)) plt.subplot(1,2,1)
 plt.imshow(image, cmap='gray')
plt.title("Original Image")
 plt.subplot(1,2,2)
plt.imshow(segmented_image, cmap='gray')
 plt.title("Segmented Image (PCA-based)")
 plt.show()
```

Output:



Program 7

Optimization via Gene Expression Algorithms The Travelling Salesman Problem (TSP) asks for the shortest possible route that visits a given set of cities exactly once and returns to the starting city. The provided text describes using a Genetic Algorithm to solve this by evolving city sequences (chromosomes) through selection, crossover, and mutation to minimize the total tour distance.

Algorithum:

Bafna Gold
Date: Page:

Lab-7

Gren Express Algorithum

Step1 : fitness Function : $f(x) = x^2$

Encoding technique : 0 to 32

use chromosome of fixed length / genotype /

Step 2 : initial population

S.no	Genotype initial chromosome	Phenotype (expression	vaue	Fitness	p
1	+xx	x^2	12	144	0.1247
2	+xx	2x	25	625	0.5411
3	x	x	5	25	0.0216
4	$-x^2$	x-2	19	361	0.3025
Sum				1155	
avg				288.75	
max				625	

	Actual count	expected count
	1	0.5
	2	2.1
	0	0.08
	1	1.25

Step 3 :

S.no	Selected chromosome	chromosome	offspring	phenotype.
1	+xx	2	+x+	$x^*(x+\ldots)$
2	+xx	1	+xx	2x
3	+xx	3	+x-	x+(x--)
4	-x	1	-x2	x+2

x value	fitness
13	169
24	576
27	729
17	289

Step4: Crossover : perform crossover randomly chosen posion (no+ raw bit.)

Step5: Mutation

S.NO	offspring before mutation	mutation applied	offspring after mutation	phenotype
1	*x+	+x→-	*x-	x+(x-1)
2	+xx	None	+xx	2x
3	+x-	-->+	-x+	x+x+x
4	+x2	None	+x2	x+2

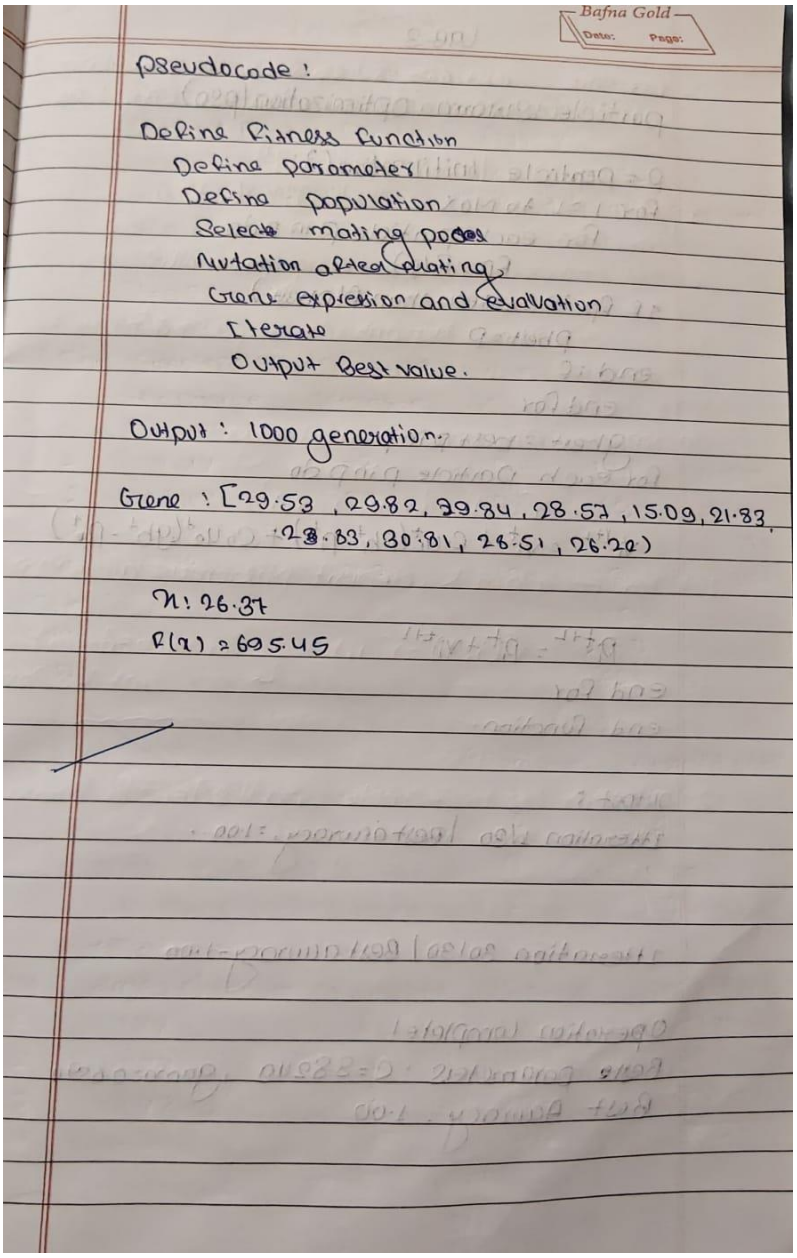
x value	fitness
29	841
24	576
27	729
20	400

Step6: Gene expression and evaluation
decode each genotype → phenotype
Calculate fitness

$\Sigma f(x) = 841+576+729+400 = 2546$
avg = 636.5
max = 841

Step7: Iterate until average

Repeat Step 3 to 6 until fitness improvement negligible or generator limit has reached



Code:

import numpy as np import random

def fitness(x): return x * np.sin(4 * x) + np.cos(2 * x)

POP_SIZE = 20 GENE_LENGTH = 8

LOWER_BOUND, UPPER_BOUND = 0, 10 GENERATIONS = 20 MUTATION_RATE = 0.1 CROSSOVER_RATE = 0.7

def create_population():

return ["".join(random.choice("01")

for _ in range(GENE_LENGTH))

for _ in range(POP_SIZE)]

def decode_gene(gene):

value = int(gene, 2) scaled = LOWER_BOUND + (UPPER_BOUND - LOWER_BOUND) * value / (2**GENE_LENGTH - 1)

return scaled

def selection(population):

k = 3 selected = random.sample(population, k)

return max(selected, key=lambda g: fitness(decode_gene(g)))

def crossover(parent1, parent2):

if random.random() < CROSSOVER_RATE:

point = random.randint(1, GENE_LENGTH - 1)

child1 = parent1[:point] + parent2[point:]

child2 = parent2[:point] + parent1[point:]

return child1, child2

return parent1, parent2

def mutation(gene):

gene_list = list(gene)

for i in range(GENE_LENGTH):

if random.random() < MUTATION_RATE:

gene_list[i] = "1" if gene_list[i] == "0" else "0" return "".join(gene_list)

def gene_expression_algorithm():

population = create_population() best_gene, best_fit = None, float("-inf")

```
for gen in range(GENERATIONS):
    new_pop = []
```

```
    while len(new_pop) < POP_SIZE:
        p1, p2 = selection(population), selection(population)
        c1, c2 = crossover(p1, p2)
        c1, c2 = mutation(c1), mutation(c2)
        new_pop.extend([c1, c2])

    population = new_pop[:POP_SIZE]



    for gene in population:
        val = decode_gene(gene)
        fit = fitness(val)
        if fit > best_fit:
            best_fit, best_gene = fit, gene

    print(f"Generation {gen+1}: Best solution = {decode_gene(best_gene):.4f}, Fitness
= {best_fit:.4f}")

return decode_gene(best_gene), best_fit


best_solution, best_fitness = gene_expression_algorithm()

print("\nFinal Best Solution:", best_solution)

 print("Final Best Fitness:", best_fitness)
```

Output:

```
Generation 1: Best solution = 9.8824, Fitness = 10.1611
Generation 2: Best solution = 9.8824, Fitness = 10.1611
Generation 3: Best solution = 9.8824, Fitness = 10.1611
Generation 4: Best solution = 9.7647, Fitness = 10.3256
Generation 5: Best solution = 9.7647, Fitness = 10.3256
Generation 6: Best solution = 9.8431, Fitness = 10.4612
Generation 7: Best solution = 9.8039, Fitness = 10.5155
Generation 8: Best solution = 9.8039, Fitness = 10.5155
Generation 9: Best solution = 9.8039, Fitness = 10.5155
Generation 10: Best solution = 9.8039, Fitness = 10.5155
Generation 11: Best solution = 9.8039, Fitness = 10.5155
Generation 12: Best solution = 9.8039, Fitness = 10.5155
Generation 13: Best solution = 9.8039, Fitness = 10.5155
Generation 14: Best solution = 9.8039, Fitness = 10.5155
Generation 15: Best solution = 9.8039, Fitness = 10.5155
Generation 16: Best solution = 9.8039, Fitness = 10.5155
Generation 17: Best solution = 9.8039, Fitness = 10.5155
Generation 18: Best solution = 9.8039, Fitness = 10.5155
Generation 19: Best solution = 9.8039, Fitness = 10.5155
Generation 20: Best solution = 9.8039, Fitness = 10.5155

Final Best Solution: 9.803921568627452
Final Best Fitness: 10.515528186962054
```