VISVESVARAYA TECHNOLOGICAL UNIVERSITY

"JnanaSangama", Belgaum -590014, Karnataka.

LAB RECORD

Bio Inspired Systems (23CS5BSBIS)

Submitted by

Spurthi Reddy P (1BM23CS338)

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 **Spurthi Reddy P (1BM23CS338),** 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

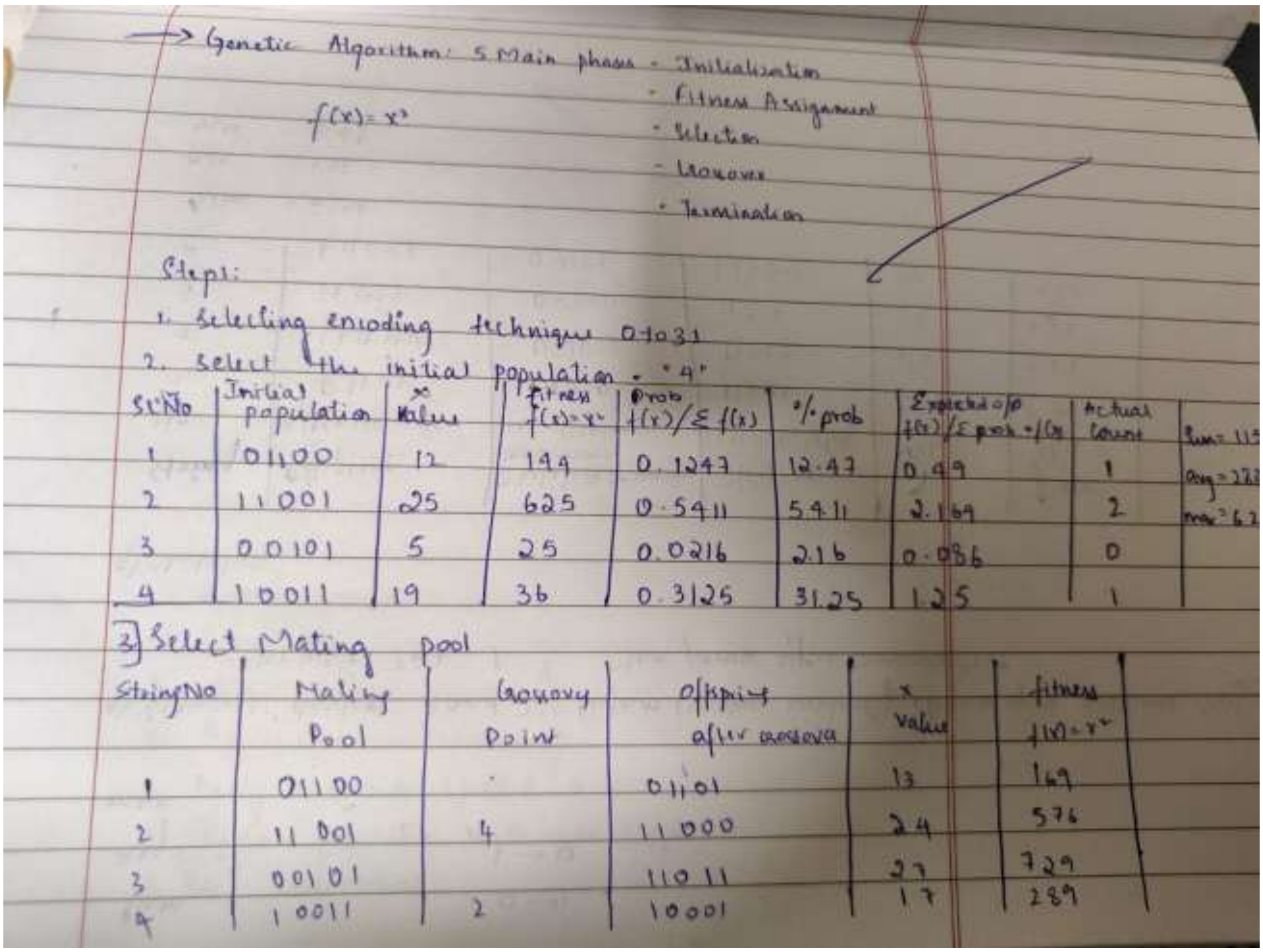
Index

Github Link:
https://github.com/Spurthi-338/BISLAB-338

Program 1

GENETIC ALGORITHM - A salesman must visit a given list of n-cities exactly once and return to the starting city. The distance between each pair of cities is known. The goal is to determine the shortest possible route that visits all cities.
Use Genetic Algorithm to find a near-optimal solution to the Travelling Salesman Problem by evolving candidate routes toward the minimum total travel distance.

Algorithm:



→ Genetic Algorithm: 5 Main phases
- Initialization
- Fitness Assignment
- Selection
- Crossover
- Termination

$f(x) = x^2$

Steps:

1. Selecting encoding technique 0-1031
2. Select the initial population - "4"

Sl No	Initial population	x Value	fitness $f(x)=x^2$	Prob $f(x)/\Sigma f(x)$	% prob	Expected o/p $f(x)/\Sigma$ prob $\cdot f(x)$	Actual Count	
1	01100	12	144	0.1247	12.47	0.49	1	Sum = 115[illegible]
2	11001	25	625	0.5411	54.11	2.164	2	Avg = 28[illegible]
3	00101	5	25	0.0216	2.16	0.086	0	max = 62[illegible]
4	10011	19	36	0.3125	31.25	1.25	1	

3] Select Mating pool

String No	Mating Pool	Crossover Point	Offspring after crossover	x value	fitness $f(x)=x^2$
1	01100		01101	13	169
2	11001	4	11000	24	576
3	00101		11011	27	729
4	10011	2	10001	17	289

Sum

avg $0 \rightarrow 1$

max $1 \rightarrow 0$

4) Crossover: Random 4 and 2 / Perform crossover randomly chosen gene positions (not single bits)

Max value: 729 / Max fitness after crossover = 729

5) Mutation:

String No	Offspring after crossover	Mutation chromosome for offspring	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

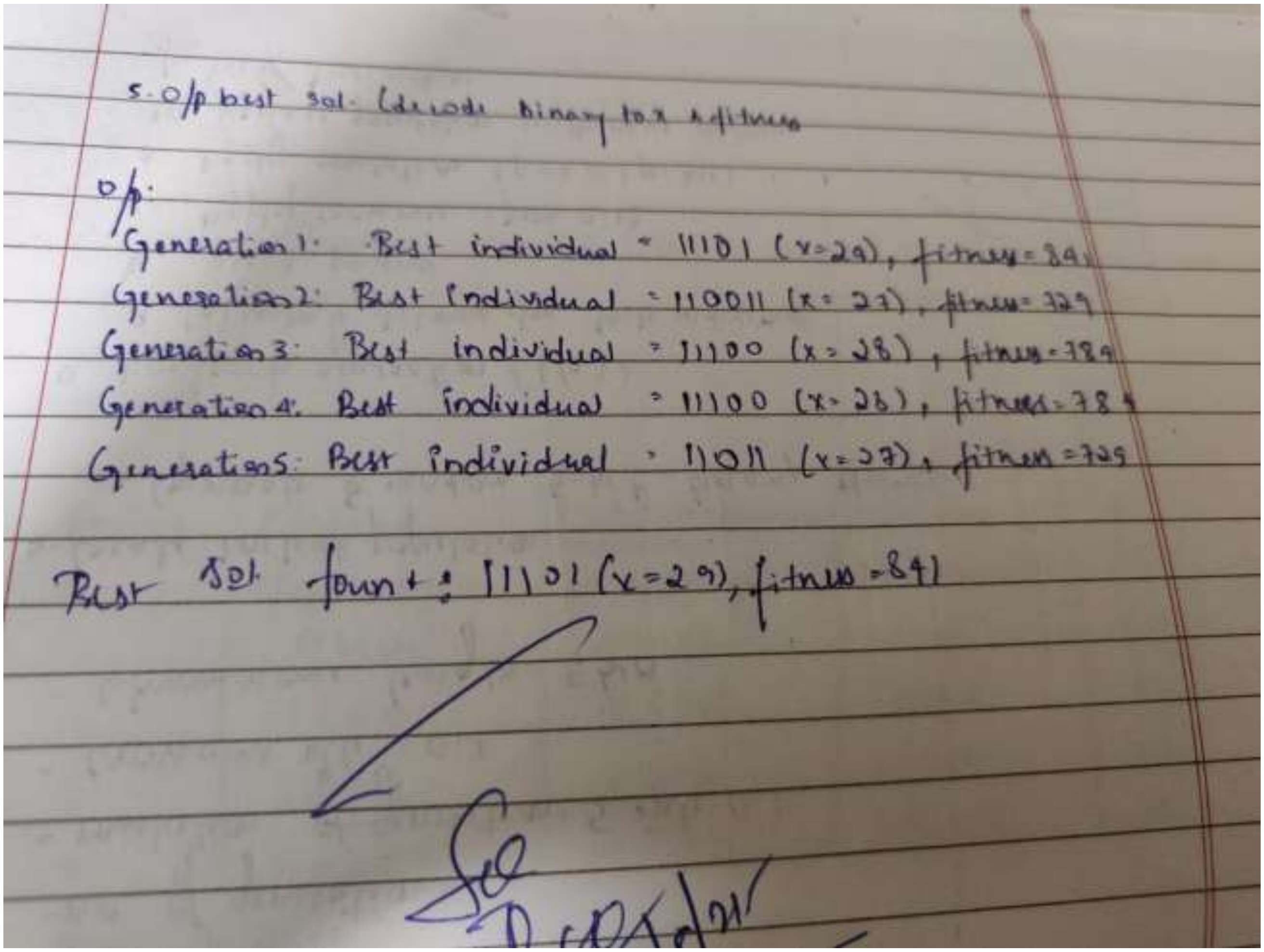
Sum = 2546

avg = 630.5

max = 841

Pseudocode

1. Define fitness fn:
   fitness(x) = x^2
2. Initialize parameters
   - population size = 6
   - no of generation = 5
   - mutation rate = 0.1
   - crossover rate = 0.7
   - Chromosome length = 5 bit
3. Create initial population
   Generate 6 random 5-bit binary strings
4. For each generation (1 to 5)
   a. Calculate fitness for each individual
   b. Select parents
   c. Apply crossover (prob 0.7)
   d. Apply mutation (prob 0.1 per bit)
   e. Replace population with new offspring
   f. Track Best Individual



5. O/p best sol. (decode binary to x & fitness

O/p:

Generation 1: Best individual = 11101 (x=29), fitness=841
Generation 2: Best Individual = 11011 (x= 27), fitness= 729
Generation 3: Best individual = 11100 (x = 28), fitness=784
Generation 4: Best individual = 11100 (x=28), fitness=784
Generation 5: Best Individual = 11011 (x=27), fitness=729

Best sol found: 11101 (x=29), fitness=841

Code:

```
import random

# Parameters
POP_SIZE = 6
GENES = 5       # 5 bits to represent 0-31
GENERATIONS = 5
CROSSOVER_RATE = 0.7
MUTATION_RATE = 0.1

# Fitness function: f(x) = x^2
def fitness(binary_str):
    x = int(binary_str, 2)
    return x * x

# Create initial population of random 5-bit binary strings
def create_population():
    population = []
```

```
    for _ in range(POP_SIZE):
        individual = ''.join(random.choice('01') for _ in range(GENES))
        population.append(individual)
    return population

# Selection: Tournament Selection of size 2
def tournament_selection(pop):
    i1, i2 = random.sample(pop, 2)
    return i1 if fitness(i1) > fitness(i2) else i2

# Crossover: Single-point crossover
def crossover(parent1, parent2):
    if random.random() < CROSSOVER_RATE:
        point = random.randint(1, GENES - 1)
        child1 = parent1[:point] + parent2[point:]
        child2 = parent2[:point] + parent1[point:]
        return child1, child2
    else:
        return parent1, parent2

# Mutation: Bit flip mutation
def mutate(individual):
    new_ind = ''
    for bit in individual:
        if random.random() < MUTATION_RATE:
            new_ind += '1' if bit == '0' else '0'
        else:
            new_ind += bit
    return new_ind

# Main GA function
def genetic_algorithm():
    population = create_population()
    best_individual = None
    best_fitness = -1

    for gen in range(1, GENERATIONS + 1):
        new_population = []

        # Evaluate and keep track of best
        for ind in population:
            ind_fit = fitness(ind)
            if ind_fit > best_fitness:
                best_fitness = ind_fit
                best_individual = ind

        # Print best in current generation
```

```
        print(f"Generation {gen}: Best Individual = {best_individual} (x={int(best_individual, 2)}),
Fitness = {best_fitness}")

        # Create new generation
        while len(new_population) < POP_SIZE:
            parent1 = tournament_selection(population)
            parent2 = tournament_selection(population)
            child1, child2 = crossover(parent1, parent2)
            child1 = mutate(child1)
            child2 = mutate(child2)
            new_population.extend([child1, child2])

        population = new_population[:POP_SIZE]

    print(f"\nBest solution found: {best_individual} (x={int(best_individual, 2)}), Fitness =
{best_fitness}")

if __name__ == "__main__":
    genetic_algorithm()
```

Output:

```
Enter 4 chromosomes (each of 5 bits, e.g., '10101'):
Chromosome 1:  01100
Chromosome 2:  11001
Chromosome 3:  00101
Chromosome 4:  10011
Generation 1: Best Fitness = 625, Best x = 25
Generation 2: Best Fitness = 784, Best x = 28
Generation 3: Best Fitness = 900, Best x = 30
Generation 4: Best Fitness = 900, Best x = 30
Generation 5: Best Fitness = 900, Best x = 30
Generation 6: Best Fitness = 900, Best x = 30
Generation 7: Best Fitness = 900, Best x = 30
Generation 8: Best Fitness = 900, Best x = 30
Generation 9: Best Fitness = 900, Best x = 30
Generation 10: Best Fitness = 900, Best x = 30
Generation 11: Best Fitness = 900, Best x = 30
Generation 12: Best Fitness = 900, Best x = 30
Generation 13: Best Fitness = 900, Best x = 30
Generation 14: Best Fitness = 900, Best x = 30
Generation 15: Best Fitness = 900, Best x = 30
Generation 16: Best Fitness = 900, Best x = 30
Generation 17: Best Fitness = 900, Best x = 30
Generation 18: Best Fitness = 900, Best x = 30
Generation 19: Best Fitness = 900, Best x = 30
Generation 20: Best Fitness = 900, Best x = 30

Best solution found:
Chromosome: 11110
x = 30
f(x) = 900
```

Program 2

PARTICLE SWARM OPTIMIZATION - Training a neural network involves finding an optimal set of weights and biases that minimize prediction error. Traditional gradient-based optimization methods.
Use Particle Swarm Optimization to optimize the weights and biases of a neural network by treating each particle as a potential weight vector and iteratively updating their positions to minimize the network's loss function.

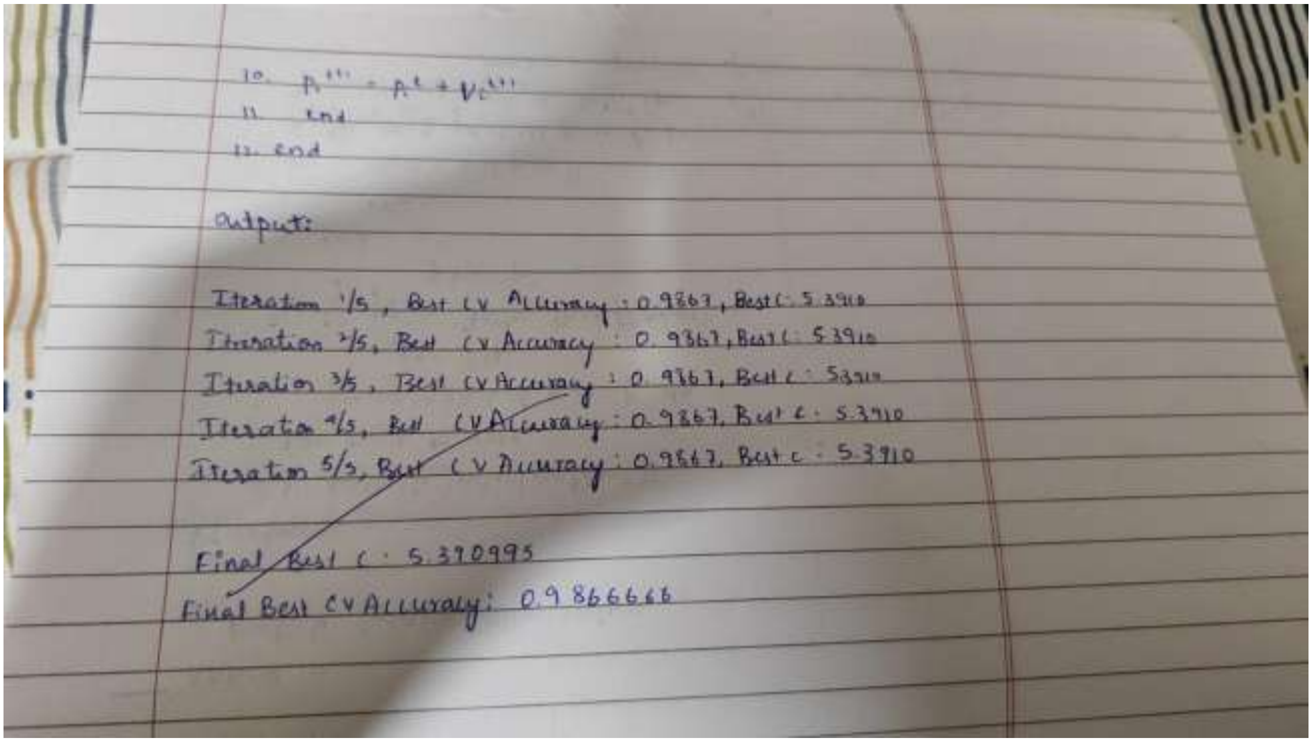
Algorithm:

→ Lab-2

Particle Swarm Optimization

Psuedocode:

1. P = particle initialization();
2. For i = 1 to max
3. for each particle p in P do
   fp = f(p)
4. If fp is better than f(pbest)
   p best = p;
5. end
6. end
7. gbest = best p in P.
8. for each particle p in P do
9. $v_i^{t+1} = \underbrace{v_i^t}_{\text{inertia}} + \underbrace{c_1 U_1^t (pb_i^t - p_i^t)}_{\text{personal influence}} + \underbrace{c_2 U_2^t (gb^t - p_i^t)}_{\text{social influence}}$



10. $p_i^{t+1} = p_i^t + v_i^{t+1}$
11. end
12. end

Output:

Iteration 1/5, Best CV Accuracy: 0.9867, Best C: 5.3910
Iteration 2/5, Best CV Accuracy: 0.9367, Best C: 5.3910
Iteration 3/5, Best CV Accuracy: 0.9367, Best C: 5.3910
Iteration 4/5, Best CV Accuracy: 0.9867, Best C: 5.3910
Iteration 5/5, Best CV Accuracy: 0.9867, Best C: 5.3910

Final Best C: 5.390995
Final Best CV Accuracy: 0.9866666

Code:

```
import random

def fitness_function(position):
    x, y = position
    return x**2 + y**2

num_particles = 10
num_iterations = 50
W = 0.3
C1 = 2
C2 = 2

particles = [[random.uniform(-10, 10), random.uniform(-10, 10)] for _ in range(num_particles)]
velocities = [[0.0, 0.0] for _ in range(num_particles)]

pbest_positions = [p[:] for p in particles]
pbest_values = [fitness_function(p) for p in particles]
```

```
gbest_index = pbest_values.index(min(pbest_values))
gbest_position = pbest_positions[gbest_index][:]
gbest_value = pbest_values[gbest_index]

for iteration in range(num_iterations):
    for i in range(num_particles):
        r1, r2 = random.random(), random.random()

        velocities[i][0] = (W * velocities[i][0] +
                            C1 * r1 * (pbest_positions[i][0] - particles[i][0]) +
                            C2 * r2 * (gbest_position[0] - particles[i][0]))
        velocities[i][1] = (W * velocities[i][1] +
                            C1 * r1 * (pbest_positions[i][1] - particles[i][1]) +
                            C2 * r2 * (gbest_position[1] - particles[i][1]))

        particles[i][0] += velocities[i][0]
        particles[i][1] += velocities[i][1]

        current_value = fitness_function(particles[i])

        if current_value < pbest_values[i]:
            pbest_positions[i] = particles[i][:]
            pbest_values[i] = current_value

            if current_value < gbest_value:
                gbest_value = current_value
                gbest_position = particles[i][:]

    print(f"Iteration {iteration+1}/{num_iterations} | Best Value: {gbest_value:.6f} at
{gbest_position}")

print("\nOptimal Solution Found:")
print(f"Best Position: {gbest_position}")
print(f"Minimum Value: {gbest_value}")
```

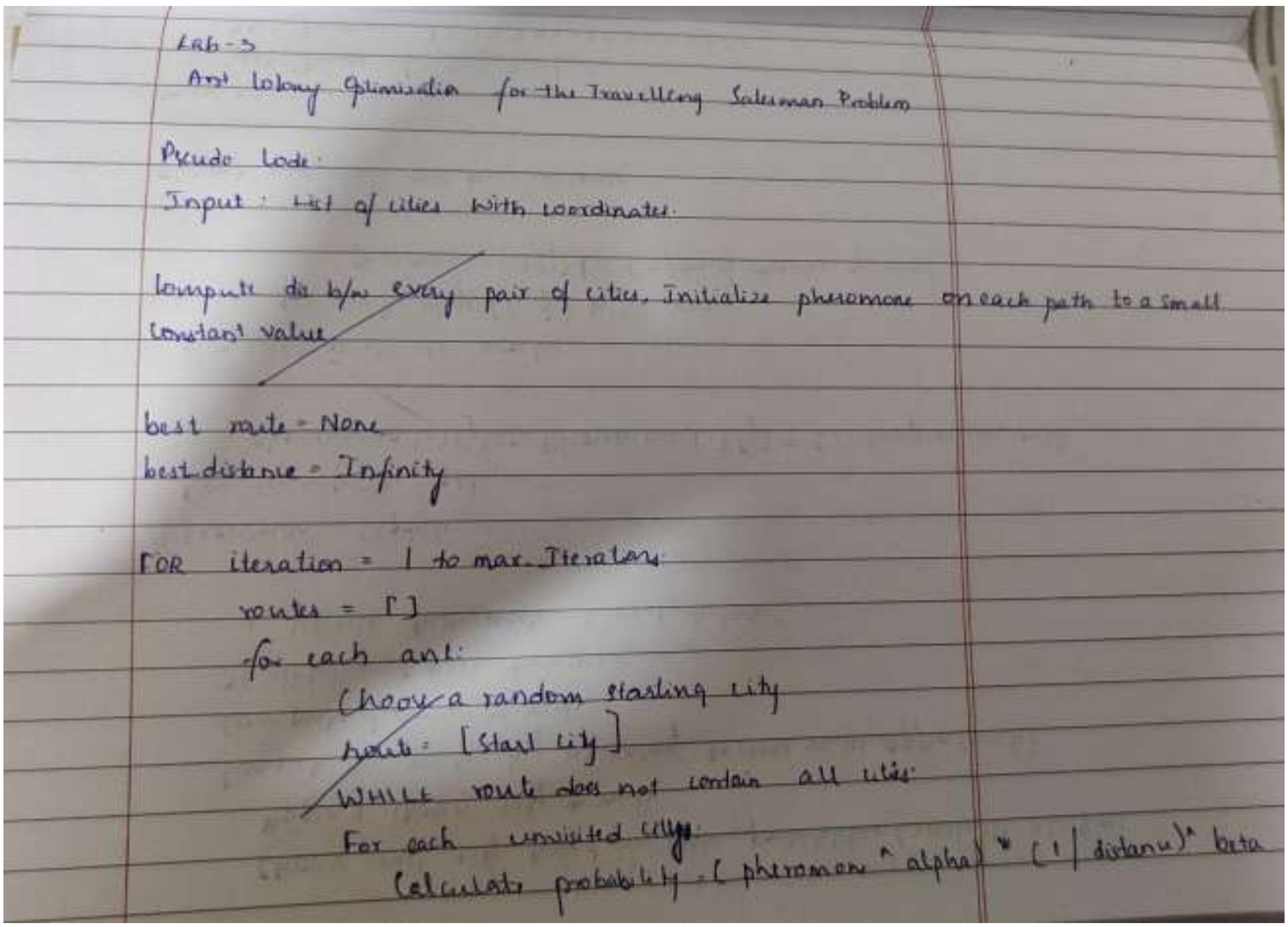
Output:

```
Iteration 1/50 | Best Value: 0.786887 at [-0.4426024797504242, -0.7687588668138685]
Iteration 2/50 | Best Value: 0.446482 at [-0.661044737940379, -0.09748000273518276]
Iteration 3/50 | Best Value: 0.047498 at [-0.09652864018059026, -0.1953982369013946]
Iteration 4/50 | Best Value: 0.016464 at [0.07681172754027843, 0.10278352042963124]
Iteration 5/50 | Best Value: 0.016464 at [0.07681172754027843, 0.10278352042963124]
Iteration 6/50 | Best Value: 0.016464 at [0.07681172754027843, 0.10278352042963124]
Iteration 7/50 | Best Value: 0.000145 at [-0.000645134915834289, 0.012028671752867981]
Iteration 8/50 | Best Value: 0.000145 at [-0.000645134915834289, 0.012028671752867981]
Iteration 9/50 | Best Value: 0.000145 at [-0.000645134915834289, 0.012028671752867981]
Iteration 10/50 | Best Value: 0.000145 at [-0.000645134915834289, 0.012028671752867981]
Iteration 11/50 | Best Value: 0.000145 at [-0.000645134915834289, 0.012028671752867981]
Iteration 12/50 | Best Value: 0.000005 at [-0.0012625430962713681, 0.0019240463815136666]
Iteration 13/50 | Best Value: 0.000005 at [-0.0012625430962713681, 0.0019240463815136666]
Iteration 14/50 | Best Value: 0.000005 at [-0.0012625430962713681, 0.0019240463815136666]
Iteration 15/50 | Best Value: 0.000005 at [-0.0012625430962713681, 0.0019240463815136666]
Iteration 16/50 | Best Value: 0.000005 at [-0.0012625430962713681, 0.0019240463815136666]
Iteration 17/50 | Best Value: 0.000005 at [-0.0012625430962713681, 0.0019240463815136666]
Iteration 18/50 | Best Value: 0.000005 at [-0.0012625430962713681, 0.0019240463815136666]
Iteration 19/50 | Best Value: 0.000002 at [-0.001366414074890062, 7.860269175524043e-06]
Iteration 20/50 | Best Value: 0.000002 at [-0.001366414074890062, 7.860269175524043e-06]
Iteration 21/50 | Best Value: 0.000002 at [-0.001366414074890062, 7.860269175524043e-06]
Iteration 22/50 | Best Value: 0.000002 at [-0.001366414074890062, 7.860269175524043e-06]
Iteration 23/50 | Best Value: 0.000001 at [-0.000727987098077961, -0.0005378750732827055]
Iteration 24/50 | Best Value: 0.000001 at [-0.0006916036998355873, -0.0005692491455515479]
Iteration 25/50 | Best Value: 0.000000 at [0.0001901152881446116, 2.3846687120860754e-05]
Iteration 26/50 | Best Value: 0.000000 at [0.0001901152881446116, 2.3846687120860754e-05]
Iteration 27/50 | Best Value: 0.000000 at [0.0001901152881446116, 2.3846687120860754e-05]
Iteration 28/50 | Best Value: 0.000000 at [9.051927524815777e-05, -1.1140007252095427e-05]
Iteration 29/50 | Best Value: 0.000000 at [5.93792641303459e-05, -3.121022569179998e-05]
Iteration 30/50 | Best Value: 0.000000 at [5.003726079500234e-05, -3.723129122371135e-05]
Iteration 31/50 | Best Value: 0.000000 at [4.7234659794399273e-05, -3.903761088328476e-05]
Iteration 32/50 | Best Value: 0.000000 at [2.7525309271407527e-05, 4.181434783550373e-05]
Iteration 33/50 | Best Value: 0.000000 at [1.6704543518187442e-05, 2.3161839136237273e-05]
Iteration 34/50 | Best Value: 0.000000 at [7.365513424750287e-06, 1.578665152668639e-05]
Iteration 35/50 | Best Value: 0.000000 at [-4.529706024454551e-06, 1.2057994367703944e-05]
Iteration 36/50 | Best Value: 0.000000 at [-2.0990070118447196e-06, 1.2085319067613795e-05]
Iteration 37/50 | Best Value: 0.000000 at [2.8449374055543557e-06, 6.92671898082449e-06]
Iteration 38/50 | Best Value: 0.000000 at [1.2219920647251537e-06, 3.6281892947483025e-06]
Iteration 39/50 | Best Value: 0.000000 at [-3.159629004034961e-08, 1.146132031451891e-06]
Iteration 40/50 | Best Value: 0.000000 at [-4.076727964700006e-07, 4.0151485246296753e-07]
Iteration 41/50 | Best Value: 0.000000 at [-5.204957483988959e-07, 1.7812969876629052e-07]
Iteration 42/50 | Best Value: 0.000000 at [-5.204957483988959e-07, 1.7812969876629052e-07]
Iteration 43/50 | Best Value: 0.000000 at [-5.204957483988959e-07, 1.7812969876629052e-07]
Iteration 44/50 | Best Value: 0.000000 at [-2.591920946149815e-07, 3.8732564263110067e-07]
Iteration 45/50 | Best Value: 0.000000 at [-3.904717963143233e-07, 4.58298204719951e-08]
Iteration 46/50 | Best Value: 0.000000 at [-6.493059825080607e-08, -2.9007028903858653e-08]
Iteration 47/50 | Best Value: 0.000000 at [3.922776049090721e-08, -1.7403223034182387e-08]
Iteration 48/50 | Best Value: 0.000000 at [3.922776049090721e-08, -1.7403223034182387e-08]
Iteration 49/50 | Best Value: 0.000000 at [9.119794577206948e-09, -2.0757413670574333e-08]
Iteration 50/50 | Best Value: 0.000000 at [9.119794577206948e-09, -2.0757413670574333e-08]

Optimal Solution Found:
Best Position: [9.119794577206948e-09, -2.0757413670574333e-08]
Minimum Value: 5.140408754217994e-16
```

Program 3

ANT COLONY OPTIMIZATION - In a communication network, data packets must be routed from a source node to a destination node through multiple possible paths. As the network grows larger and more dynamic, finding the shortest and least congested path becomes increasingly complex for traditional deterministic routing algorithms.
Use Ant Colony Optimization to compute the optimal or near-optimal routing path between nodes in a network

Algorithm:



Lab-3

Ant Colony Optimization for the Travelling Salesman Problem

Pseudo Code:

Input: List of cities with coordinates.

Compute dis b/w every pair of cities, Initialize pheromone on each path to a small constant value

best route = None
best distance = Infinity

FOR iteration = 1 to max_Iterations
routes = []
for each ant:
Choose a random starting city
route = [start city]
WHILE route does not contain all cities:
For each unvisited city:
Calculate probability = (pheromone ^ alpha) * (1 / distance)^ beta

```
        Choose next city based on these probabilities (roulette selection)
        Append chosen city to route
    Complete the route by returning to start routes.append(route)
    Compute distance of route
    If distance < best distance:
        update best route & best_distance

// pheromone update:
  For each path (i,j)
      pheromone[i][j] = Pheromone[i][j] * (1 - evaporation_rate)

  For each route in routes:
      For each edge (i, j) in the route:
          pheromone[i][j] += (1 / total distance of route)

Output best route and best_distance

Output:
    Best Route : [2, 4, 3, 1, 0]
    Best Distance : 17.832452649316352
```

Code:

```
import numpy as np
import random

class ACO_TSP:
  def __init__(self, distances, n_ants=10, n_iterations=50, alpha=1, beta=3, rho=0.5, Q=100):
    self.distances = distances
    self.num_cities = distances.shape[0]
    self.n_ants = n_ants
    self.n_iterations = n_iterations
    self.alpha = alpha
    self.beta = beta
    self.rho = rho
    self.Q = Q
    self.pheromone = np.ones((self.num_cities, self.num_cities))
    self.visibility = 1 / (distances + np.eye(self.num_cities))

  def run(self):
```

```
        best_distance = np.inf
        best_tour = None

        for iteration in range(self.n_iterations):
            all_tours = []
            all_distances = []

            for _ in range(self.n_ants):
                tour = self.construct_tour()
                distance = self.calculate_distance(tour)
                all_tours.append(tour)
                all_distances.append(distance)

            self.update_pheromones(all_tours, all_distances)

            min_distance = min(all_distances)
            if min_distance < best_distance:
                best_distance = min_distance
                best_tour = all_tours[np.argmin(all_distances)]

            print(f"Iteration {iteration+1}: Shortest Distance = {min_distance:.2f}")

        print("\nBest Tour:", best_tour)
        print("Shortest Distance Found:", best_distance)
        return best_tour, best_distance

    def construct_tour(self):
        start = random.randint(0, self.num_cities - 1)
        tour = [start]
        visited = set(tour)

        for _ in range(self.num_cities - 1):
            current = tour[-1]
            next_city = self.select_next_city(current, visited)
            tour.append(next_city)
            visited.add(next_city)

        tour.append(tour[0])
        return tour

    def select_next_city(self, current, visited):
        probabilities = []
        pheromone = np.copy(self.pheromone[current])
        visibility = np.copy(self.visibility[current])

        for city in range(self.num_cities):
            if city not in visited:
```

```
                probabilities.append((pheromone[city] ** self.alpha) * (visibility[city] ** self.beta))
            else:
                probabilities.append(0)

        probabilities = np.array(probabilities)
        probabilities = probabilities / probabilities.sum()
        return np.random.choice(range(self.num_cities), p=probabilities)

    def calculate_distance(self, tour):
        distance = 0
        for i in range(len(tour) - 1):
            distance += self.distances[tour[i], tour[i+1]]
        return distance

    def update_pheromones(self, all_tours, all_distances):
        self.pheromone *= (1 - self.rho)
        for tour, dist in zip(all_tours, all_distances):
            for i in range(len(tour) - 1):
                self.pheromone[tour[i], tour[i+1]] += self.Q / dist


if __name__ == "__main__":
    distance_matrix = np.array([
        [0, 2, 9, 10, 7, 3],
        [2, 0, 6, 4, 3, 8],
        [9, 6, 0, 5, 2, 7],
        [10, 4, 5, 0, 6, 4],
        [7, 3, 2, 6, 0, 5],
        [3, 8, 7, 4, 5, 0]
    ])

    aco = ACO_TSP(distance_matrix, n_ants=8, n_iterations=20, alpha=1, beta=3, rho=0.4)
    best_tour, best_distance = aco.run()
```

Output:

```
Iteration 1: Shortest Distance = 19.00
Iteration 2: Shortest Distance = 19.00
Iteration 3: Shortest Distance = 19.00
Iteration 4: Shortest Distance = 19.00
Iteration 5: Shortest Distance = 19.00
Iteration 6: Shortest Distance = 19.00
Iteration 7: Shortest Distance = 19.00
Iteration 8: Shortest Distance = 19.00
Iteration 9: Shortest Distance = 19.00
Iteration 10: Shortest Distance = 19.00
Iteration 11: Shortest Distance = 19.00
Iteration 12: Shortest Distance = 19.00
Iteration 13: Shortest Distance = 19.00
Iteration 14: Shortest Distance = 19.00
Iteration 15: Shortest Distance = 19.00
Iteration 16: Shortest Distance = 19.00
Iteration 17: Shortest Distance = 19.00
Iteration 18: Shortest Distance = 19.00
Iteration 19: Shortest Distance = 19.00
Iteration 20: Shortest Distance = 19.00

Best Tour: [4, np.int64(1), np.int64(0), np.int64(5), np.int64(3), np.int64(2), 4]
Shortest Distance Found: 19
```

Program 4

CUCKOO SEARCH OPTIMIZATION – Many engineering design problems, such as designing a spring, a gear system, or a pressure vessel, require determining a set of parameters that minimize cost while satisfying mechanical, safety, and performance constraints.
Use Cuckoo Search Optimization to determine the optimal design parameters for an engineering system

Algorithm:

17/10/25 Lab 4: Cuckoo Search Algorithm

Pseudocode:

```
Define objective_function(x):
    Return sum of (x[i]^2) for all dimensions
Define levy_flight(Lambda, size):
    Compute sigma1 using Gamma and sine functions
    Generate u and v from normal distributions
    step = u/|v|^(1/Lambda)
    return step
Define cuckoo_search(obj_func, n, d, max_iter, pa, alpha, bounds):
    Initialize nests randomly within bounds
    Compute fitness for all nests
    Find best solution and best fitness

    For each iteration t = 1 to max_iter:
        For each nest i:
        step = levy_flight(1.5, d)
        new_solution = nests[i] + alpha * step * (nests[i] - random_nest)
```

Clip new solution within bounds
If obj_func (new solution) < fitness

For each iteration = 1 to max_iter:
  for each nest i:
    step = levy_flight (1.5, d)
    new solution = nests [i] + alpha * step * (best [i] - random_nest)
    Clip new solution within bounds
    if obj_func (new_solution) < fitness [i]:
        Replace nest [i] and update fitness [i]

  Generate random mask K (probability > pa)
  Stepsize = random * (random_nest 1 - random_nest 2)
  clip newnests within bounds

  Evaluate new fitness
  If new fitness better, update nests and fitness
  Update best_sol if a better one found
Return best_sol, best_fitness

Main:
  Run cuckoo search (objective_function)
  Print best solution and fitness
def obj (x):
  C, gamma = x
  model = SVC (C = C, gamma = gamma)
  Scores = cross_val_score (model, x_train, y_train, cv=5)
  return -scores.mean()

output : Best solution: [-1.98273890e-05 1.25815893e-05 1.54131550e-08]
Best Fitness: 3.8040223549333365e-10

Code:

```
import numpy as np
import math

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

def initialize_nests(num_nests, dim, lower_bound, upper_bound):
    return np.random.uniform(lower_bound, upper_bound, size=(num_nests, dim))

def levy_flight(Lambda, size):
    sigma = (math.gamma(1 + Lambda) * math.sin(math.pi * Lambda / 2) /
             (math.gamma((1 + Lambda) / 2) * Lambda * 2**((Lambda - 1) / 2))) ** (1 / Lambda)
    u = np.random.randn(*size) * sigma
    v = np.random.randn(*size)
    step = u / np.abs(v) ** (1 / Lambda)
    return step

def cuckoo_search(num_nests=25, dim=2, lower_bound=-10, upper_bound=10,
                  pa=0.25, max_iter=100):

    nests = initialize_nests(num_nests, dim, lower_bound, upper_bound)
    fitness = np.apply_along_axis(objective_function, 1, nests)

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

    for t in range(max_iter):
        new_nests = nests + 0.01 * levy_flight(1.5, nests.shape) * (nests - best_nest)
        new_nests = np.clip(new_nests, lower_bound, upper_bound)

        new_fitness = np.apply_along_axis(objective_function, 1, new_nests)

        mask = new_fitness < fitness
        nests[mask] = new_nests[mask]
        fitness[mask] = new_fitness[mask]

        rand = np.random.rand(num_nests, dim)
        new_nests = np.where(rand > pa, nests,
                             initialize_nests(num_nests, dim, lower_bound, upper_bound))

        new_fitness = np.apply_along_axis(objective_function, 1, new_nests)
        mask = new_fitness < fitness
        nests[mask] = new_nests[mask]
        fitness[mask] = new_fitness[mask]
```

```python
        if np.min(fitness) < best_fitness:
            best_nest = nests[np.argmin(fitness)].copy()
            best_fitness = np.min(fitness)

        print(f"Iteration {t+1}/{max_iter} | Best Fitness: {best_fitness:.6f}")

    return best_nest, best_fitness

best_solution, best_value = cuckoo_search()
print("\nBest solution found:", best_solution)
print("Best fitness value:", best_value)
```

Output:

```
Iteration 1/100 | Best Fitness: 7.116416
Iteration 2/100 | Best Fitness: 2.736363
Iteration 3/100 | Best Fitness: 2.736363
Iteration 4/100 | Best Fitness: 2.736363
Iteration 5/100 | Best Fitness: 2.736363
Iteration 6/100 | Best Fitness: 2.736363
Iteration 7/100 | Best Fitness: 2.736363
Iteration 8/100 | Best Fitness: 2.736363
Iteration 9/100 | Best Fitness: 2.736363
Iteration 10/100 | Best Fitness: 0.310548
Iteration 11/100 | Best Fitness: 0.310548
Iteration 12/100 | Best Fitness: 0.310548
Iteration 13/100 | Best Fitness: 0.310548
Iteration 14/100 | Best Fitness: 0.310548
Iteration 15/100 | Best Fitness: 0.310548
Iteration 16/100 | Best Fitness: 0.310548
Iteration 17/100 | Best Fitness: 0.310548
Iteration 18/100 | Best Fitness: 0.310548
Iteration 19/100 | Best Fitness: 0.310548
Iteration 20/100 | Best Fitness: 0.160487
Iteration 21/100 | Best Fitness: 0.160487
Iteration 22/100 | Best Fitness: 0.160487
Iteration 23/100 | Best Fitness: 0.160487
Iteration 24/100 | Best Fitness: 0.013181
Iteration 25/100 | Best Fitness: 0.013181
Iteration 26/100 | Best Fitness: 0.013181
Iteration 27/100 | Best Fitness: 0.013181
Iteration 28/100 | Best Fitness: 0.013181
Iteration 29/100 | Best Fitness: 0.013181
Iteration 30/100 | Best Fitness: 0.013181
Iteration 31/100 | Best Fitness: 0.013181
Iteration 32/100 | Best Fitness: 0.013181
Iteration 33/100 | Best Fitness: 0.013181
Iteration 34/100 | Best Fitness: 0.013181
Iteration 35/100 | Best Fitness: 0.013181
Iteration 36/100 | Best Fitness: 0.013181
Iteration 37/100 | Best Fitness: 0.013181
Iteration 38/100 | Best Fitness: 0.013181
Iteration 39/100 | Best Fitness: 0.013181
Iteration 40/100 | Best Fitness: 0.013181
Iteration 41/100 | Best Fitness: 0.013181
```

Program 5

GREY WOLF OPTIMIZATION - Support Vector Machines (SVMs) require optimal selection of hyperparameters—such as the regularization parameter C, kernel parameter γ, and kernel type—to achieve high classification accuracy.
Use Grey Wolf Optimization to automatically determine the optimal SVM hyperparameters by modelling each wolf as a candidate solution in the (C, γ)search space. The wolves will follow the leadership hierarchy (alpha, beta, delta) and encircling–hunting behavior to explore and exploit the parameter space.

Algorithm:

1/10/2025 Lab 5

Grey Wolf Optimizer (GWO)

Initialize population of wolves x randomly within the search space
Evaluate fitness of each wolf
Identify alpha (best), beta (second best), and delta (third best) wolves

For t = 1 to max iterations:
    Update coefficient vector a linearly from 2 to 0

    For each wolf in population:
        For each dimension d:
            Calculate A and C vectors with random r1, r2 in [0,1]
            Calculate dis D_alpha, D_beta, D_delta to beta, alpha, delta pos
            Calculate new position x1, x2, x3 based on encircling eq
            Update pos of wolf i as avg of x1, x2, x3

        Handle boundries ()
    Evaluate fitness of each wolf
    Update alpha, beta, delta wolves if better sol are found
Return alpha wolf pos as best sol & its fitness

Output

Best sol found: [-5.3922, -5.9162, 5.1235, 6.2023, -5.6963]
Best obj value: 1.60211

Code:

```
import numpy as np

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

def grey_wolf_optimizer(num_wolves=30, dim=2, max_iter=50, lower_bound=-10,
upper_bound=10):
    wolves = np.random.uniform(lower_bound, upper_bound, (num_wolves, dim))

    Alpha_pos = np.zeros(dim)
    Beta_pos = np.zeros(dim)
    Delta_pos = np.zeros(dim)

    Alpha_score = float("inf")
    Beta_score = float("inf")
    Delta_score = float("inf")

    for t in range(max_iter):
        for i in range(num_wolves):
            wolves[i] = np.clip(wolves[i], lower_bound, upper_bound)
            fitness = objective_function(wolves[i])

            if fitness < Alpha_score:
                Delta_score = Beta_score
                Delta_pos = Beta_pos.copy()
                Beta_score = Alpha_score
                Beta_pos = Alpha_pos.copy()
                Alpha_score = fitness
                Alpha_pos = wolves[i].copy()
            elif fitness < Beta_score:
                Delta_score = Beta_score
                Delta_pos = Beta_pos.copy()
                Beta_score = fitness
                Beta_pos = wolves[i].copy()
            elif fitness < Delta_score:
                Delta_score = fitness
                Delta_pos = wolves[i].copy()

        a = 2 - t * (2 / max_iter)

        for i in range(num_wolves):
            for j in range(dim):
                r1 = np.random.rand()
                r2 = np.random.rand()
```

```
            A1 = 2 * a * r1 - a
            C1 = 2 * r2
            D_alpha = abs(C1 * Alpha_pos[j] - wolves[i][j])
            X1 = Alpha_pos[j] - A1 * D_alpha

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

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

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

        print(f"Iteration {t+1}/{max_iter} | Best Fitness: {Alpha_score:.6f}")

    return Alpha_pos, Alpha_score
best_position, best_score = grey_wolf_optimizer()
print("\nBest solution found:", best_position)
print("Best fitness value:", best_score)
```

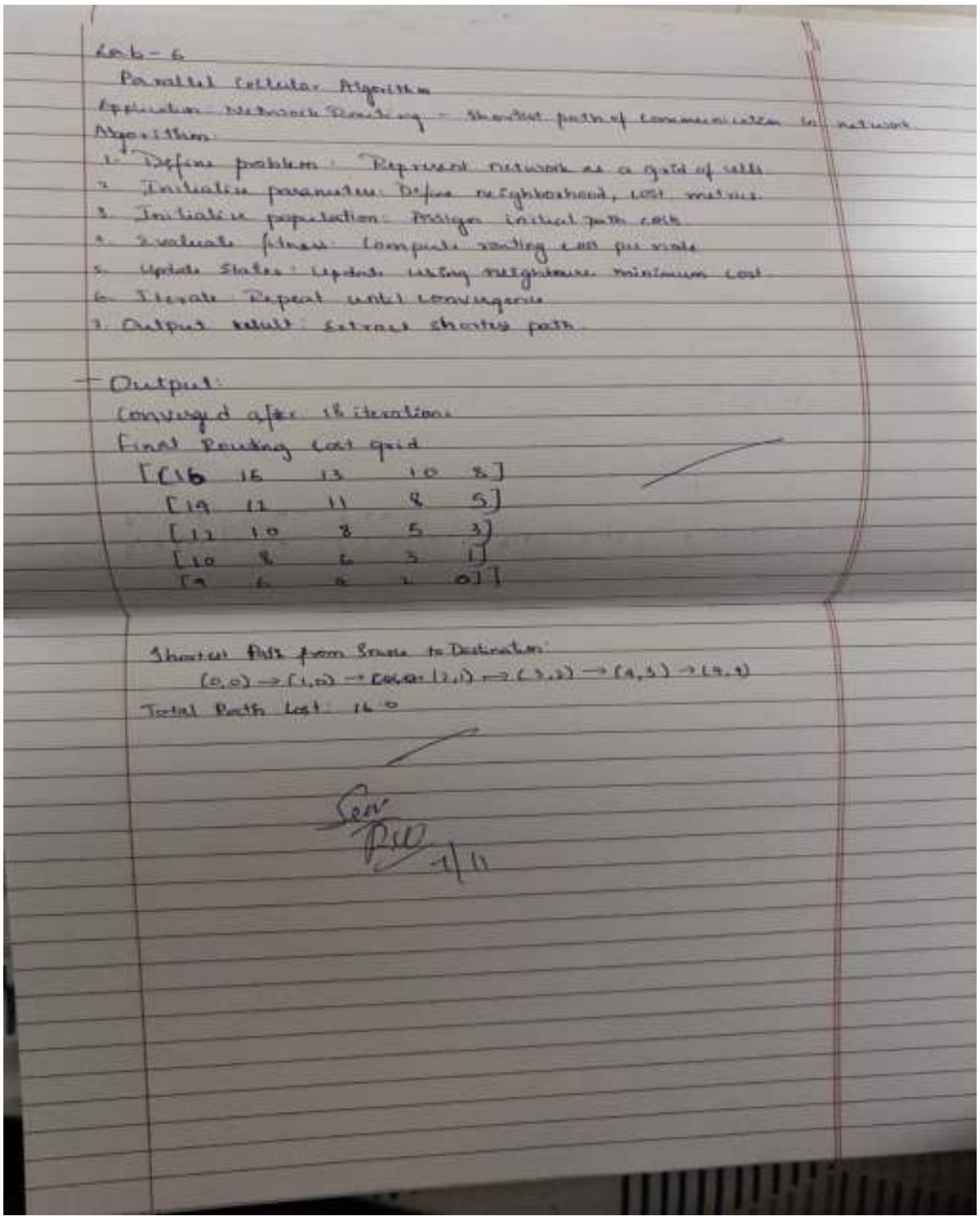
Output:

```
Iteration 1/50 | Best Fitness: 2.919390
Iteration 2/50 | Best Fitness: 1.128525
Iteration 3/50 | Best Fitness: 0.012965
Iteration 4/50 | Best Fitness: 0.012965
Iteration 5/50 | Best Fitness: 0.012965
Iteration 6/50 | Best Fitness: 0.002791
Iteration 7/50 | Best Fitness: 0.000128
Iteration 8/50 | Best Fitness: 0.000017
Iteration 9/50 | Best Fitness: 0.000017
Iteration 10/50 | Best Fitness: 0.000004
Iteration 11/50 | Best Fitness: 0.000000
Iteration 12/50 | Best Fitness: 0.000000
Iteration 13/50 | Best Fitness: 0.000000
Iteration 14/50 | Best Fitness: 0.000000
Iteration 15/50 | Best Fitness: 0.000000
Iteration 16/50 | Best Fitness: 0.000000
Iteration 17/50 | Best Fitness: 0.000000
Iteration 18/50 | Best Fitness: 0.000000
Iteration 19/50 | Best Fitness: 0.000000
Iteration 20/50 | Best Fitness: 0.000000
Iteration 21/50 | Best Fitness: 0.000000
Iteration 22/50 | Best Fitness: 0.000000
Iteration 23/50 | Best Fitness: 0.000000
Iteration 24/50 | Best Fitness: 0.000000
Iteration 25/50 | Best Fitness: 0.000000
Iteration 26/50 | Best Fitness: 0.000000
Iteration 27/50 | Best Fitness: 0.000000
Iteration 28/50 | Best Fitness: 0.000000
Iteration 29/50 | Best Fitness: 0.000000
Iteration 30/50 | Best Fitness: 0.000000
Iteration 31/50 | Best Fitness: 0.000000
Iteration 32/50 | Best Fitness: 0.000000
Iteration 33/50 | Best Fitness: 0.000000
Iteration 34/50 | Best Fitness: 0.000000
Iteration 35/50 | Best Fitness: 0.000000
Iteration 36/50 | Best Fitness: 0.000000
Iteration 37/50 | Best Fitness: 0.000000
Iteration 38/50 | Best Fitness: 0.000000
Iteration 39/50 | Best Fitness: 0.000000
Iteration 40/50 | Best Fitness: 0.000000
Iteration 41/50 | Best Fitness: 0.000000
Iteration 42/50 | Best Fitness: 0.000000
Iteration 43/50 | Best Fitness: 0.000000
Iteration 44/50 | Best Fitness: 0.000000
Iteration 45/50 | Best Fitness: 0.000000
Iteration 46/50 | Best Fitness: 0.000000
Iteration 47/50 | Best Fitness: 0.000000
Iteration 48/50 | Best Fitness: 0.000000
Iteration 49/50 | Best Fitness: 0.000000
Iteration 50/50 | Best Fitness: 0.000000

Best solution found: [4.93421853e-18 2.16997188e-18]
Best fitness value: 2.9055290410997664e-35
```

<u>Program 6</u>

PARALLEL CELLULAR ALGORITHM - Modern communication networks require routing algorithms that can adapt quickly to changes in traffic load, link failures, and congestion. Traditional centralized routing strategies may suffer from slow updates, high computational cost, and poor scalability as network size increases.
Use a Parallel Cellular Algorithm to compute optimal routing paths in a dynamic communication network. Each cell in the cellular grid represents a router or network node and updates its routing information based on local interactions with neighbouring cells.

Algorithm:



Lab - 6
Parallel Cellular Algorithm
Application: Network Routing - shortest path of communication in network
Algorithm:
1. Define problem: Represent network as a grid of cells
2. Initialize parameters: Define neighbourhood, cost matrix
3. Initialize population: Assign initial path costs
4. Evaluate fitness: Compute routing cost per node
5. Update States: Update using neighbours minimum cost
6. Iterate: Repeat until convergence
7. Output result: Extract shortest path

Output:
Converged after 18 iterations
Final Routing cost grid
[[16 15 13 10 8]
[14 12 11 8 5]
[12 10 8 5 3]
[10 8 6 3 1]
[9 6 4 2 0]]

Shortest Path from Source to Destination:
(0,0) → (1,0) → ~~(2,0)~~ (2,1) → (3,2) → (4,3) → (4,4)
Total Path Cost: 16.0

Code:

```python
import numpy as np

GRID_SIZE = 5
MAX_ITER = 100
INF = 1e9

source = (0, 0)
destination = (4, 4)

np.random.seed(42)
cost_matrix = np.random.randint(1, 10, size=(GRID_SIZE, GRID_SIZE))

state = np.full((GRID_SIZE, GRID_SIZE), INF)
state[destination] = 0

neighbors = [(-1, 0), (1, 0), (0, -1), (0, 1)]

def get_neighbors(i, j):
    """Return valid neighboring cells"""
    valid_neighbors = []
    for dx, dy in neighbors:
        ni, nj = i + dx, j + dy
        if 0 <= ni < GRID_SIZE and 0 <= nj < GRID_SIZE:
            valid_neighbors.append((ni, nj))
    return valid_neighbors

for iteration in range(MAX_ITER):
    new_state = state.copy()
    for i in range(GRID_SIZE):
        for j in range(GRID_SIZE):
            if (i, j) == destination:
                continue
            neighbor_costs = []
            for ni, nj in get_neighbors(i, j):
                total_cost = cost_matrix[ni, nj] + state[ni, nj]
                neighbor_costs.append(total_cost)
            if neighbor_costs:
                new_state[i, j] = min(neighbor_costs)
    if np.allclose(new_state, state):
        print(f"Converged after {iteration} iterations.")
        break
    state = new_state

path = [source]
current = source
```

```python
while current != destination:
    i, j = current
    nbs = get_neighbors(i, j)
    next_cell = min(nbs, key=lambda n: state[n])
    path.append(next_cell)
    current = next_cell

print("Final Routing Cost Grid:")
print(np.round(state, 2))
print("\nShortest Path from Source to Destination:")
print(" → ".join([str(p) for p in path]))
print(f"\nTotal Path Cost: {state[source]}")
```

Output:

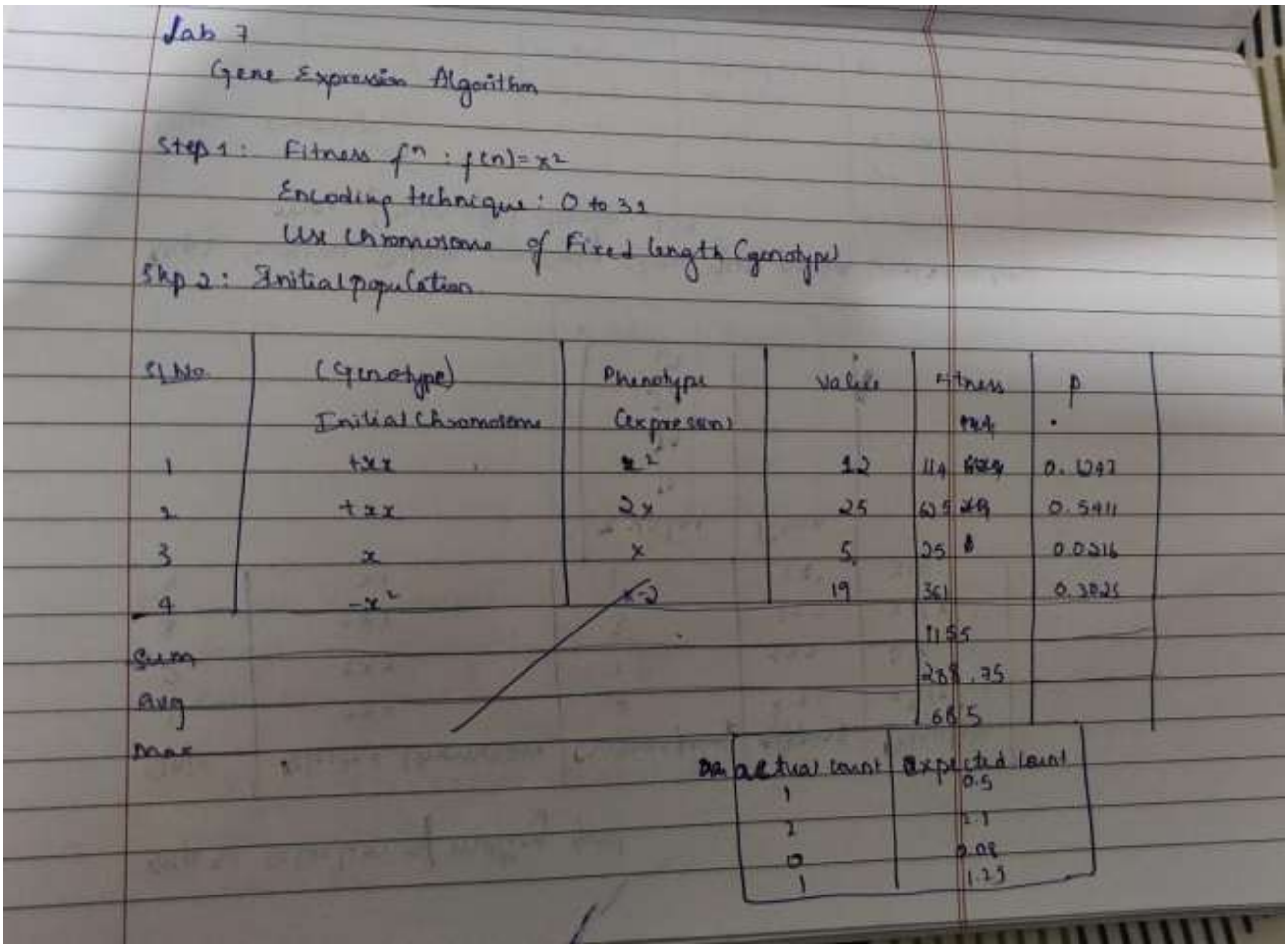
```
Converged after 8 iterations.
Final Routing Cost Grid:
[[33. 30. 22. 17. 17.]
 [30. 23. 15. 12. 13.]
 [22. 15. 12.  6.  8.]
 [20. 12.  6.  4.  3.]
 [19. 13.  4.  3.  0.]]

Shortest Path from Source to Destination:
(0, 0) → (1, 0) → (2, 0) → (2, 1) → (3, 1) → (3, 2) → (4, 2) → (4, 3) → (4, 4)

Total Path Cost: 33.0
```

[ ]:

Program 7

GENE EXPRESSION ALGORITHM - Machine learning models often perform poorly when the original input features do not sufficiently capture the underlying patterns in the data. Manually engineering new features is time-consuming and requires domain expertise.
Use the Gene Expression and Evaluation Algorithm to automatically construct new features from existing input variables for a supervised learning task.

Algorithm:



Lab 7
Gene Expression Algorithm

Step 1: Fitness f^n : $f(x) = x^2$
Encoding technique : 0 to 31
Use Chromosome of Fixed length (Genotype)

Step 2: Initial population

Sl.No.	(Genotype) Initial Chromosome	Phenotype (expression)	Value	Fitness	p
1	$+x x$	x^2	12	144	0.1247
2	$+x x$	$2x$	25	625	0.5411
3	x	x	5	25	0.0216
4	$-x^2$	$x-2$	19	361	0.3125
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: Selection of mating pool

SlNo	Selected Chromosome	Crossover point	offspring	Phenotype
1	+xx	2	* x+	$x^*(x+\ldots)$
2	+xx	1	+xx	2x
3	+xx	3	+x-	$x+(x\ldots)$
4	-xx	1	$+x^2$	x+1

x value	Fitness
13	
29	
27	
17	

Step 4: Crossover Perform randomly chosen gene position (not raw bits).
max fitness after crossover = 729

Step 5: mutation

SlNo	offspring before mutation	mutation applied	offspring after mutation	Phenotype
1	* x+	$+ \to -$	+x-	$x^*(x-\ldots)$
2	+xx	None	+xx	2x
3	+x-	$- \to +$	-x+	$x+x^*x$
4	+x2	None	$+x^2$	x+2

x value	Fitness
29	841
24	576
27	729
20	400

Step 6: Gene Exp and evaluation
decode each genotype → phenotype
calculate fitness

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

Step 7: Iterate until convergence
Repeat step 3 to 6 until fitness improvement is negligible or generation limit has reached.

Pseudocode:

Define fitness f^n
Define parameters
Generate population
Select mating pool
mutation after mating
Gene exp & evaluation
Iterate
o/p best value.

Output: [1000 generations]

Genes: [29.33, 29.82, 29.34, 28.57, 15.09, 21.53, 25.33, 30.61, 21.51, 26.37]

x = 26.37
f(x) = 695.45

Code:

```
import random
import math

def fitness_function(x):
    return x * math.sin(10 * math.pi * x) + 2

POPULATION_SIZE = 6
GENE_LENGTH = 10
MUTATION_RATE = 0.05
CROSSOVER_RATE = 0.8
```

```python
GENERATIONS = 20
DOMAIN = (-1, 2)

def random_gene():
    return random.uniform(DOMAIN[0], DOMAIN[1])

def create_chromosome():
    return [random_gene() for _ in range(GENE_LENGTH)]

def initialize_population(size):
    return [create_chromosome() for _ in range(size)]

def evaluate_population(population):
    return [fitness_function(express_gene(chrom)) for chrom in population]

def express_gene(chromosome):
    return sum(chromosome) / len(chromosome)

def select(population, fitnesses):
    total_fitness = sum(fitnesses)
    pick = random.uniform(0, total_fitness)
    current = 0
    for individual, fitness in zip(population, fitnesses):
        current += fitness
        if current > pick:
            return individual
    return random.choice(population)

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 mutate(chromosome):
    new_chromosome = []
    for gene in chromosome:
        if random.random() < MUTATION_RATE:
            new_chromosome.append(random_gene())
        else:
            new_chromosome.append(gene)
    return new_chromosome

def gene_expression_algorithm():
    population = initialize_population(POPULATION_SIZE)
```

```
    best_solution = None
    best_fitness = float("-inf")

    for generation in range(GENERATIONS):
        fitnesses = evaluate_population(population)

        for i, chrom in enumerate(population):
            if fitnesses[i] > best_fitness:
                best_fitness = fitnesses[i]
                best_solution = chrom[:]

        print(f"Generation {generation+1}: Best Fitness = {best_fitness:.4f}, Best x = {express_gene(best_solution):.4f}")

        new_population = []
        while len(new_population) < POPULATION_SIZE:
            parent1 = select(population, fitnesses)
            parent2 = select(population, fitnesses)
            offspring1, offspring2 = crossover(parent1, parent2)
            offspring1 = mutate(offspring1)
            offspring2 = mutate(offspring2)
            new_population.extend([offspring1, offspring2])

        population = new_population[:POPULATION_SIZE]

    print("\nBest solution found:")
    print(f"Genes: {best_solution}")
    x_value = express_gene(best_solution)
    print(f"x = {x_value:.4f}")
    print(f"f(x) = {fitness_function(x_value):.4f}")

if __name__ == "__main__":
    gene_expression_algorithm()
```

Output:

```
Generation 1: Best Fitness = 2.6411, Best x = 0.6570
Generation 2: Best Fitness = 2.6411, Best x = 0.6570
Generation 3: Best Fitness = 2.6411, Best x = 0.6570
Generation 4: Best Fitness = 2.6411, Best x = 0.6570
Generation 5: Best Fitness = 2.6411, Best x = 0.6570
Generation 6: Best Fitness = 2.6411, Best x = 0.6570
Generation 7: Best Fitness = 2.6411, Best x = 0.6570
Generation 8: Best Fitness = 2.6411, Best x = 0.6570
Generation 9: Best Fitness = 2.6411, Best x = 0.6570
Generation 10: Best Fitness = 2.6493, Best x = 0.6494
Generation 11: Best Fitness = 2.6493, Best x = 0.6494
Generation 12: Best Fitness = 2.6493, Best x = 0.6494
Generation 13: Best Fitness = 2.6493, Best x = 0.6494
Generation 14: Best Fitness = 2.6493, Best x = 0.6494
Generation 15: Best Fitness = 2.6493, Best x = 0.6494
Generation 16: Best Fitness = 2.6493, Best x = 0.6494
Generation 17: Best Fitness = 2.6493, Best x = 0.6494
Generation 18: Best Fitness = 2.6493, Best x = 0.6494
Generation 19: Best Fitness = 2.6493, Best x = 0.6494
Generation 20: Best Fitness = 2.6493, Best x = 0.6494

Best solution found:
Genes: [0.4390976923728207, 1.2526878024513985, -0.4825669181112343, 1.2100668505221361, -0.46407671239571313, 1.3715894583648138, 0.61
51068898319401, 0.16056055888077347, 1.2202911837851609, 1.1714745345907573]
x = 0.6494
f(x) = 2.6493
```