

# **VISVESVARAYA TECHNOLOGICAL UNIVERSITY**

**“JnanaSangama”, Belgaum -590014, Karnataka.**



## **LAB RECORD**

### **Bio Inspired Systems (23CS5BSBIS)**

*Submitted by*

**G M Kusuma (1BM24CS405)**

*in partial fulfilment 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 Dec-2025**

**B.M.S. College of Engineering,  
Bull Temple Road, Bangalore 560019**  
(Affiliated To Visvesvaraya Technological University, Belgaum)  
**Department of Computer Science and Engineering**



**CERTIFICATE**

This is to certify that the Lab work entitled “ Bio Inspired Systems (23CS5BSBIS)” carried out by **G M Kusuma (1BM24CS405)**, 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.

Soumya T Assistant Professor Department of CSE, BMSCE	Dr. Kavitha Sooda Professor & HOD Department of CSE, BMSCE
---	--

## Index

<b>Sl. No.</b>	<b>Date</b>	<b>Experiment Title</b>	<b>Page No.</b>
1	18/08/2025	Genetic Algorithm for Optimization Problems	4-8
2	01/09/2025	Particle Swarm Optimization for Function Optimization	9-11
3	08/09/2025	Ant Colony Optimization for the Traveling Salesman Problem	12-16
4	15/09/2025	Cuckoo Search (CS)	17-20
5	29/09/2025	Grey Wolf Optimizer (GWO)	21-25
6	13/10/2025	Parallel Cellular Algorithms and Programs	26-30
7	25/08/2025	Optimization via Gene Expression Algorithms	30-34

Github Link:

<https://github.com/Kusumagm07/BIS>

## **Program 1**

### **Genetic Algorithm for Optimization Problems:**

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

Implementation Steps:

1. Define the Problem: Create a mathematical function to optimize.
2. Initialize Parameters: Set the population size, mutation rate, crossover rate, and number of generations.
3. Create Initial Population: Generate an initial population of potential solutions.
4. Evaluate Fitness: Evaluate the fitness of each individual in the population.
5. Selection: Select individuals based on their fitness to reproduce.
6. Crossover: Perform crossover between selected individuals to produce offspring.
7. Mutation: Apply mutation to the offspring to maintain genetic diversity.
8. Iteration: Repeat the evaluation, selection, crossover, and mutation processes for a fixed number of generations or until convergence criteria are met.
9. Output the Best Solution: Track and output the best solution found during the generations.

Algorithm:

PEEVOCODE  
function fitness(x)  
 $f(x) = x^2$   
function decode(chromosomes)  
 convert the binary list to decimal numbers  
 return decimal\_value;  
function create\_population()  
 population = []  
 for i = 1 to 10:  
 chromosome = random list of 6 bits (0 or 1)  
 ADD chromosome to population  
 return population.  
function evaluation\_population(population)  
 fitness\_list = []  
 for each chromosome in population:  
 x = decode(chromosome)  
 f = fitness(x)  
 ADD f to fitness\_list  
 return fitness\_list.  
function select\_parents(population, fitness\_list)  
 use roulette wheel selection based on fitness values.  
 return selected\_parents.  
function crossover(parent1, parent2)  
 if random < 0.7:  
 choose a random crossover point  
 child1 = first part of parent1 + second part  
 of parent2.  
 child2 = first part of parent2 + second part  
 else  
 child1 = copy of parent1  
 child2 = copy of parent2  
 Return child1, child2.  
function mutate(chromosome)  
 for each bit in chromosome:  
 if random < 0.1:  
 flip the bit, i.e. because 1 becomes 0 and vice versa.  
 return chromosome.

Function genetic\_algorithm()  
 population = create\_population()  
 best\_chromosome = None;  
 best\_fitness = -∞  
 for generation = 1 to 10:  
 fitness\_list = evaluate\_population(population)  
 find chromosome with highest fitness  
 if f(x) > best\_fitness:  
 best\_chromosome = that chromosome  
 best\_fitness = f(x)  
 print generation number, best & best\_fitness  
 selected = select\_parents(population, fitness\_list)  
 next\_generation = []  
 for i = 0 to population\_size:  
 parent1 = selected[i]  
 parent2 = selected[i+1]  
 child1, child2 = crossover(parent1, parent2)  
 child1 = mutate(child1)  
 child2 = mutate(child2)  
 ADD child1 and child2 to next-generation  
 population = next-generation  
 return decode(best\_chromosome)  
 best\_fitness  
 call genetic\_algorithm().

## Code:

```

import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import make_classification
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score

# Creating a sample dataset
X, y = make_classification(n_samples=500, n_features=10, n_informative=8, n_classes=2)
X_train, X_val, y_train, y_val = train_test_split(X, y, test_size=0.2)

# Neural Network Structure
input_size = X.shape[1]
hidden_size = 5
output_size = 1

# Helper functions for the Neural Network
def sigmoid(x):
    return 1 / (1 + np.exp(-x))
def forward_pass(X, weights1, weights2):
    hidden_input = np.dot(X, weights1)
    hidden_output = sigmoid(hidden_input)
    output_input = np.dot(hidden_output, weights2)
    output = sigmoid(output_input)
  
```

```

    return output
def compute_fitness(weights):
    predictions = forward_pass(X_train, weights['w1'], weights['w2'])
    predictions = (predictions > 0.5).astype(int)
    accuracy = accuracy_score(y_train, predictions)
    return accuracy
# Genetic Algorithm Parameters
population_size = 20
generations = 10
mutation_rate = 0.1

# Initialize Population
population = []
for _ in range(population_size):
    individual = {
        'w1': np.random.randn(input_size, hidden_size),
        'w2': np.random.randn(hidden_size, output_size)
    }
    population.append(individual)

# Tracking performance
best_fitness_history = []
average_fitness_history = []

# Main Genetic Algorithm Loop
for generation in range(generations):
    # Evaluate Fitness of each Individual
    fitness_scores = np.array([compute_fitness(individual) for individual in population])
    best_fitness = np.max(fitness_scores)
    average_fitness = np.mean(fitness_scores)
    best_fitness_history.append(best_fitness)
    average_fitness_history.append(average_fitness)

    # Selection: Select top half of the population
    sorted_indices = np.argsort(fitness_scores)[-1:-population_size//2:-1]
    population = [population[i] for i in sorted_indices]

    # Crossover and Mutation
    new_population = []
    while len(new_population) < population_size:
        parents = np.random.choice(population, 2, replace=False)
        child = {
            'w1': (parents[0]['w1'] + parents[1]['w1']) / 2,
            'w2': (parents[0]['w2'] + parents[1]['w2']) / 2
        }
        # Mutation
        if np.random.rand() < mutation_rate:
            child['w1'] += np.random.randn(*child['w1'].shape) * 0.1
        new_population.append(child)

```

```

        child['w2'] += np.random.randn(*child['w2'].shape) * 0.1
    new_population.append(child)
population = new_population
print(f"Generation {generation+1}, Best Fitness: {best_fitness:.4f}")

# Evaluate the best individual on validation set
best_individual = population[np.argmax(fitness_scores)]
predictions = forward_pass(X_val, best_individual['w1'], best_individual['w2'])
predictions = (predictions > 0.5).astype(int)
final_accuracy = accuracy_score(y_val, predictions)
print(f'Final Accuracy on Validation Set: {final_accuracy:.4f}')
# Plotting the results
plt.figure(figsize=(10, 5))
plt.plot(best_fitness_history, label='Best Fitness')
plt.plot(average_fitness_history, label='Average Fitness')
plt.title('Fitness Over Generations')
plt.xlabel('Generation')
plt.ylabel('Fitness')
plt.legend()
plt.grid(True)
plt.show()

```

## Program 2

### **Particle Swarm Optimization for Function Optimization:**

Particle Swarm Optimization (PSO) is inspired by the social behavior of birds flocking or fish schooling. PSO is used to find optimal solutions by iteratively improving a candidate solution with regard to a given measure of quality. Implement the PSO algorithm using Python to optimize a mathematical function.

Implementation Steps:

1. Define the Problem: Create a mathematical function to optimize.
2. Initialize Parameters: Set the number of particles, inertia weight, cognitive and social coefficients.
3. Initialize Particles: Generate an initial population of particles with random positions and velocities.
4. Evaluate Fitness: Evaluate the fitness of each particle based on the optimization function.
5. Update Velocities and Positions: Update the velocity and position of each particle based on its own best position and the global best position.
6. Iterate: Repeat the evaluation, updating, and position adjustment for a fixed number of iterations or until convergence criteria are met.
7. Output the Best Solution: Track and output the best solution found during the iterations.

### **Algorithm:**

Lab-3  
1/09/27

particle swarm optimization function

Implementation steps

- ① Define the problem: Create a mathematical function to optimize.
- ② Initialize parameters: set the no. of particles, weight, cognitive and social coefficients.
- ③ Initialize particles: Generate an initial population of particles based on random position and velocities.
- ④ Evaluate fitness based on the optimization function.
- ⑤ Update velocity and position.
- ⑥ Iterate: Repeat the evaluation, update and position.
- ⑦ Output the best solution found during iteration.

Pseudocode

```

Initialize Swarm:
    for each particle i = 1 to N:
        - Randomly initialize position  $x_i$  within bounds
        - Randomly initialize velocity  $v_i$ 
        - Set personal best  $p_i = x_i$ 
        - Evaluate fitness  $f(p_i)$ 
    Set global best  $g = \arg\min(f(p_i))$ 
    Repeat until stopping condition (max iteration)
        for each particle i:
            → for each dim d:
                 $v_i[d] = w * v_i[d]$ 
                 $+ c_1 * r_1 * (p_i[d] - x_i[d])$ 
                 $+ c_2 * r_2 * (g[d] - x_i[d])$ 
                (Clamp  $v_i[d]$  within  $v_{min}, v_{max}$ )
                 $x_i[d] = x_i[d] + v_i[d]$ 
                (if  $x_i[d]$  outside bounds → set bound)
            → evaluate fitness  $f(x_i)$ 
            → if  $f(x_i) < f(p_i)$ 
                → if  $f(p_i) < f(g)$ 
                     $p_i = x_i$ 
                     $g = p_i$ 

```

Return G as best solution.

Output

Iteration 1/10 - Best score: 0.09766	→
Iteration 2/10 - Best score: 0.09766	→
Iteration 3/10 - Best score: 0.09766	→
Iteration 4/10 - Best score: 0.05867	→
Iteration 5/10 - Best score: 0.05867	→
Iteration 6/10 - Best score: 0.00555	→
Iteration 7/10 - Best score: 0.00555	→
Iteration 8/10 - Best score: 0.00555	→
Iteration 9/10 - Best score: 0.00244	→
Iteration 10/10 - Best score: 0.00156	→

Best solution found:  
 Position [-0.012423005953519098,  
 -0.03743473550008094]  
 Value: 0.0015571882883021925]

*Get  
19/01*

## Code:

```

import random

# Define the function to optimize
def objective_function(position):
    x, y = position
    return x**2 + y**2

# PSO parameters
num_particles = 30
num_dimensions = 2
max_iterations = 10 # Changed from 100 to 10

```

w = 0.5  
 c1 = 1.5  
 c2 = 1.5

# Initialize particles

```

particles = []
velocities = []
personal_best_positions = []
personal_best_scores = []

for _ in range(num_particles):
    position = [random.uniform(-10, 10) for _ in range(num_dimensions)]
    velocity = [random.uniform(-1, 1) for _ in range(num_dimensions)]
    particles.append(position)
    velocities.append(velocity)
    personal_best_positions.append(position[:])
    personal_best_scores.append(objective_function(position))

global_best_index = personal_best_scores.index(min(personal_best_scores))
global_best_position = personal_best_positions[global_best_index][:]
global_best_score = personal_best_scores[global_best_index]

for iteration in range(max_iterations):
    for i in range(num_particles):
        for d in range(num_dimensions):
            r1 = random.random()
            r2 = random.random()

            velocities[i][d] = (w * velocities[i][d] +
                c1 * r1 * (personal_best_positions[i][d] - particles[i][d]) +
                c2 * r2 * (global_best_position[d] - particles[i][d]))

            particles[i][d] += velocities[i][d]

        fitness = objective_function(particles[i])

        if fitness < personal_best_scores[i]:
            personal_best_positions[i] = particles[i][:]
            personal_best_scores[i] = fitness

        if fitness < global_best_score:
            global_best_position = particles[i][:]
            global_best_score = fitness

    print(f"Iteration {iteration+1}/{max_iterations} — Best Score: {global_best_score:.5f}")

print("\nBest solution found:")
print(f"Position: {global_best_position}")
print(f"Value: {global_best_score}")

```

**Program 3:**

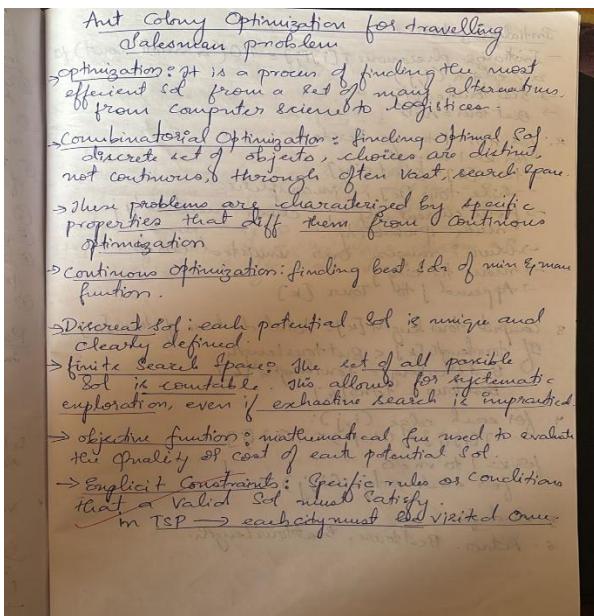
Ant Colony Optimization for the Traveling Salesman Problem:

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.

#### Implementation Steps:

1. Define the Problem: Create a set of cities with their coordinates.
2. Initialize Parameters: Set the number of ants, the importance of pheromone (alpha), the importance of heuristic information (beta), the evaporation rate (rho), and the initial pheromone value.
3. Construct Solutions: Each ant constructs a solution by probabilistically choosing the next city based on pheromone trails and heuristic information.
4. Update Pheromones: After all ants have constructed their solutions, update the pheromone trails based on the quality of the solutions found.
5. Iterate: Repeat the construction and updating process for a fixed number of iterations or until convergence criteria are met.
6. Output the Best Solution: Keep track of and output the best solution found during the iterations.

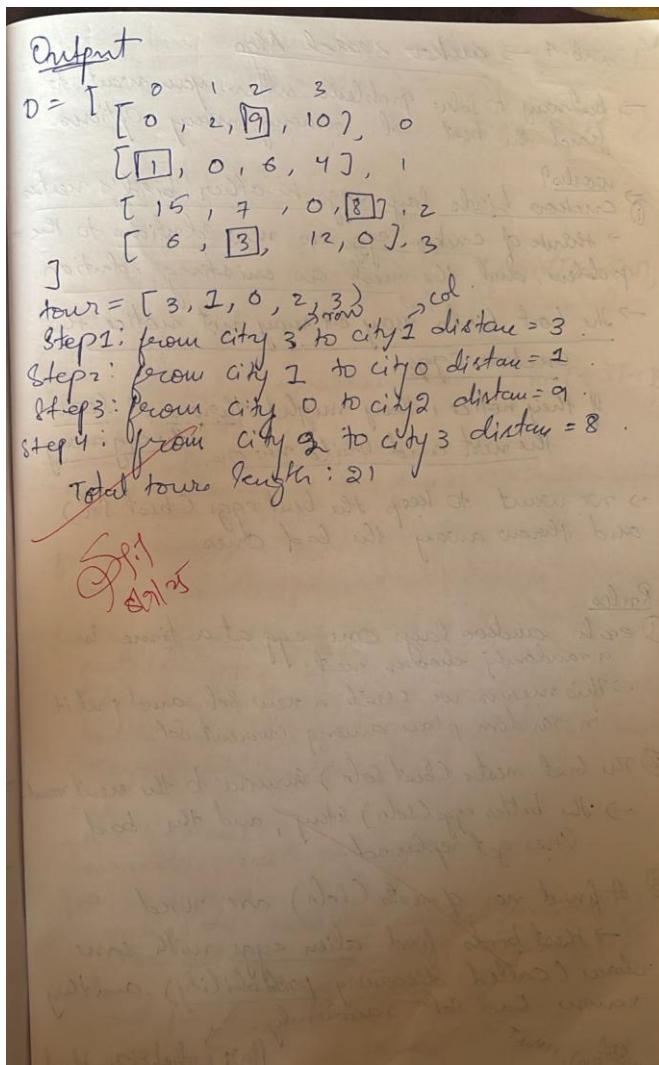
#### Algorithm:



Initialization

- Initialize pheromone  $\tau_{ij}$  =  $T_0 / \text{a small const}$  all edges  $(i, j)$
- BestTourLength =  $\infty$
- BestTour = null

2. for Iter = 1 to max Iter do
  - for k = 1 to m do
    - tour[k] = Start city
    - while tour [k] is incomplete:
      - from current city i, choose next city j with probability
      - value ranges over unvisited cities
      - move to city j
      - Append j to tour [k]
  - 3. ComputeTourLength[tour] = total distance of tour[tour]
    - If TourLength[tour] < BestTourLength
    - BestTourLength = TourLength[tour]
    - BestTour = tour[tour]
  - 4. for each edge  $(i, j)$ :
    - $T(i, j) \leftarrow (1 - \rho) * T(i, j) + \eta_j$
    - for k = 1 to m do
      - for each edge  $(i, j)$
  - 5. end for
  - 6. Return BestTour, BestTourLength.



### Code:

```

import random
import math

class ACO_TSP:
    def __init__(self, distances, n_ants=10, n_iterations=100, alpha=1, beta=5, rho=0.5, Q=100):
        self.distances = distances
        self.n = len(distances)
        self.n_ants = n_ants
        self.n_iterations = n_iterations
        self.alpha = alpha
        self.beta = beta
        self.rho = rho
        self.Q = Q
        self.pheromone = [[1 for _ in range(self.n)] for _ in range(self.n)]
        self.best_length = float("inf")
    
```

```

self.best_tour = None

def run(self):
    for it in range(self.n_iterations):
        all_tours = []
        all_lengths = []

        for ant in range(self.n_ants):
            tour = self.construct_solution()
            length = self.compute_length(tour)

            all_tours.append(tour)
            all_lengths.append(length)

            if length < self.best_length:
                self.best_length = length
                self.best_tour = tour

        self.update_pheromones(all_tours, all_lengths)

    return self.best_tour, self.best_length

def construct_solution(self):
    start = random.randint(0, self.n - 1)
    tour = [start]
    unvisited = set(range(self.n))
    unvisited.remove(start)

    current = start
    while unvisited:
        next_city = self.choose_next_city(current, unvisited)
        tour.append(next_city)
        unvisited.remove(next_city)
        current = next_city

    return tour

def choose_next_city(self, current, unvisited):
    probs = []
    total = 0
    for city in unvisited:
        tau = self.pheromone[current][city] ** self.alpha
        eta = (1.0 / self.distances[current][city]) ** self.beta
        value = tau * eta
        probs.append((city, value))
        total += value

```

```

r = random.random() * total
cumulative = 0
for city, value in probs:
    cumulative += value
    if cumulative >= r:
        return city
return probs[-1][0]

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

def update_pheromones(self, all_tours, all_lengths):
    for i in range(self.n):
        for j in range(self.n):
            self.pheromone[i][j] *= (1 - self.rho)

    for tour, length in zip(all_tours, all_lengths):
        for i in range(len(tour) - 1):
            a, b = tour[i], tour[i+1]
            self.pheromone[a][b] += self.Q / length
            self.pheromone[b][a] += self.Q / length
        a, b = tour[-1], tour[0]
        self.pheromone[a][b] += self.Q / length
        self.pheromone[b][a] += self.Q / length

# Example usage
if __name__ == "__main__":
    distances = [
        [0, 2, 9, 10, 7],
        [1, 0, 6, 4, 3],
        [15, 7, 0, 8, 3],
        [6, 3, 12, 0, 11],
        [9, 7, 5, 6, 0]
    ]

    aco = ACO_TSP(distances, n_ants=10, n_iterations=50, alpha=1, beta=5, rho=0.5, Q=100)
    best_tour, best_length = aco.run()

    # Format tour as edges
    path_str = " -> ".join(map(str, best_tour)) + f" -> {best_tour[0]}"
    print("\n==== Final Best Solution ===")
    print("Best path:", path_str)

```

```
print("Best path length:", best_length)
```

#### **Program 4:**

#### **Cuckoo Search (CS):**

Cuckoo Search (CS) is a nature-inspired optimization algorithm based on the brood parasitism of some cuckoo species. This behavior involves laying eggs in the nests of other birds, leading to the optimization of survival strategies. CS uses Lévy flights to generate new solutions, promoting global search capabilities and avoiding local minima. The algorithm is widely used for solving continuous optimization problems and has applications in various domains, including engineering design, machine learning, and data mining.

#### **Implementation Steps:**

1. Define the Problem: Create a mathematical function to optimize.
2. Initialize Parameters: Set the number of nests, the probability of discovery, and the number of iterations.
3. Initialize Population: Generate an initial population of nests with random positions.
4. Evaluate Fitness: Evaluate the fitness of each nest based on the optimization function.
5. Generate New Solutions: Create new solutions via Lévy flights.
6. Abandon Worst Nests: Abandon a fraction of the worst nests and replace them with new random positions.
7. Iterate: Repeat the evaluation, updating, and replacement process for a fixed number of iterations or until convergence criteria are met.
8. Output the Best Solution: Track and output the best solution found during the iterations.

#### **Algorithm:**

Algo / Pseudo code implementing in Travelling Salesman prob

```

Function euclidean-distance (c1, c2)
    Return sqrt ((c1[0] - c2[0])^2 + (c1[1])^2)

Function fitness(tour, cities)
    total-distance = 0
    for i = 1 to len (tour) - 1
        total-distance += euclidean-distance (cities[tour[i]], cities[tour[i+1]])
    End for
    Return total-distance + euclidean-distance (cities[tour[-1]], cities[tour[0]])
```

function levy flight (dim, beta=1.5)
 Return (random-normal (0, 1, dim) / abs (random-normal (0, 1, dim))) ^ (1/beta)

function initialize - population (n, dim)
 Returns [random-permutations (dim) for i in range(n)]

function cuckoo-search (cities, n, Pg, Mmax)
 nest = init-pop (n, len(cities))
 fitness-value = [fitness (nest, cities) for nest in nest]
 best-sol > best-fitness = min (zip (nest, fitness-value))
 Key = lambda x: x[1]

t = 0
while t < Mmax
 new-nests = nest.copy()
 for i = 1 to n
 swap-idx = random-choice (len(cities), 2)
 new-nests[i][swap-idx] = new-nests[i][t]
 swap-idx (t:-1)
 end for
Return best-sol, best-fitness.

Output

```

Iteration 1: Best Distance = 242130
Iteration 2: Best Distance = 242130
Iteration 3: Best Distance = 242130
:
Iteration 30: Best Distance = 19.7409
Best Sol(Tour): [3 5 6 7 4 2 1 0]
Best Distance (total tour length): 19.7409.
```

Solve  
R0119

**Code:**

`import numpy as np`

```

# Problem data (same as before)
weights = np.array([12, 7, 11, 8, 9])
values = np.array([24, 13, 23, 15, 16])
capacity = 26

n = 10      # Number of nests
Pa = 0.25   # Probability of abandoning worst nests
max_iter = 100

def fitness(solution):
    total_weight = np.sum(solution * weights)
    if total_weight > capacity:
        return 0
    else:
        return np.sum(solution * values)

def initial_nests(n, dim):
    return np.random.randint(0, 2, (n, dim))

def levy_flight(Lambda=1.5):
    sigma = (np.math.gamma(1 + Lambda) * np.sin(np.pi * Lambda / 2) /
             (np.math.gamma((1 + Lambda) / 2) * Lambda * 2 ** ((Lambda - 1) / 2))) ** (1 /
    Lambda)
    u = np.random.normal(0, sigma, size=weights.shape[0])
    v = np.random.normal(0, 1, size=weights.shape[0])
    step = u / np.abs(v) ** (1 / Lambda)
    return step

def get_new_solution(nest):
    step_size = levy_flight()
    new_sol_cont = nest + 0.01 * step_size * (nest - np.mean(nest))
    s = 1 / (1 + np.exp(-new_sol_cont))
    new_sol = np.array([1 if x > 0.5 else 0 for x in s])
    return new_sol

def abandon_worst_nests(nests, fitnesses, Pa):
    num_abandon = int(Pa * len(nests))
    worst_indices = np.argsort(fitnesses)[:num_abandon]
    for i in worst_indices:
        nests[i] = np.random.randint(0, 2, nests.shape[1])
        fitnesses[i] = fitness(nests[i])
    return nests, fitnesses

def cuckoo_search():
    dim = weights.shape[0]
    t = 0

```

```

# Step 4 and 5: Initialize population and evaluate fitness
nests = initial_nests(n, dim)
fitnesses = np.array([fitness(nest) for nest in nests])

while t < max_iter:
    for i in range(n):
        # Step 7 and 8: Generate cuckoo and evaluate fitness
        cuckoo = get_new_solution(nests[i])
        cuckoo_fit = fitness(cuckoo)

        # Step 9: Choose a nest randomly
        j = np.random.randint(n)

        # Step 10-12: Replace if cuckoo is better
        if cuckoo_fit > fitnesses[j]:
            nests[j] = cuckoo
            fitnesses[j] = cuckoo_fit

    # Step 13 and 14: Abandon fraction Pa of worst nests and build new ones
    nests, fitnesses = abandon_worst_nests(nests, fitnesses, Pa)

    # Step 15 and 16: Keep and rank the best solution
    best_index = np.argmax(fitnesses)
    best_nest = nests[best_index].copy()
    best_fitness = fitnesses[best_index]

    print(f"Iteration {t+1}: Best fitness = {best_fitness}")

    t += 1

# Step 19: Output the best solution
return best_nest, best_fitness

best_solution, best_val = cuckoo_search()

print("\nBest solution found:")
print("Items selected:", best_solution)
print("Total value:", best_val)
print("Total weight:", np.sum(best_solution * weights))

```

### Program 5:

Grey Wolf Optimizer (GWO):

The Grey Wolf Optimizer (GWO) algorithm is a swarm intelligence algorithm inspired by the social hierarchy and hunting behavior of grey wolves. It mimics the leadership structure of alpha, beta, delta, and omega wolves and their collaborative hunting strategies. The GWO algorithm uses these social hierarchies to model the optimization process, where the alpha wolves guide the search process while beta and delta wolves assist in refining the search direction. This algorithm is effective for continuous optimization problems and has applications in engineering, data analysis, and machine learning.

**Implementation Steps:**

1. Define the Problem: Create a mathematical function to optimize.
2. Initialize Parameters: Set the number of wolves and the number of iterations.
3. Initialize Population: Generate an initial population of wolves with random positions.
4. Evaluate Fitness: Evaluate the fitness of each wolf based on the optimization function.
5. Update Positions: Update the positions of the wolves based on the positions of alpha, beta, and delta wolves.
6. Iterate: Repeat the evaluation and position updating process for a fixed number of iterations or until convergence criteria are met.
7. Output the Best Solution: Track and output the best solution found during the iterations

**Algorithm:**

Lab - 6  
gray wolf optimization. F.d 29/9/25

Pseudo code

Input: Dataset ( $x_i, y_i$ ), no. of features  $D$ , fitness function,  
 Wolves  $N$ , iterations MaxIterations, learning rate

Initialize wolf positions randomly into  $\mathbb{R}^{N \times D}$  (size  $N \times D$ )

Initialize Alpha, Beta, Delta wolves with worst fitness.

for  $t = 1$  to MaxIterations:

$a = 2 - 2 * (t / \text{MaxIterations})$

for each wolf  $i$ :

convert position to binary mask using sigmoid and threshold

time  $t$  for each wolf

for each wolf:

for each feature  $d$ :

update position  $[i, j, d]$  using Alpha, Beta & Delta rule, partition done by  $\lambda$  and storage

clamp position to  $[0, 1]$

convert Alpha's position to binary mask

Return best feature mask and feature

Output

Iteration 1/10, best fitness 0.8580.

Iteration 10/10, Best fit 0.8889

Selected feature indices [ 1, 4, 5, 9, 0, 6, 7, 3 ]

No. of features selected = 8 from 9 total

Spiral 29/9/25

### Code:

```

import numpy as np
from sklearn.datasets import load_breast_cancer
from sklearn.model_selection import cross_val_score
from sklearn.ensemble import RandomForestClassifier

# Load dataset
data = load_breast_cancer()
X = data.data
y = data.target
num_features = X.shape[1]

# Gray Wolf Optimizer parameters
num_wolves = 10 # Population size
max_iter = 10 # Number of iterations

```

```

# Binary GWO helper functions
def sigmoid(x):
    return 1 / (1 + np.exp(-x))

def binary_transform(x):
    return np.where(sigmoid(x) > np.random.rand(len(x)), 1, 0)

# Fitness function: classification accuracy
def fitness(position):
    selected_features = np.where(position == 1)[0]
    if len(selected_features) == 0:
        return 0
    X_selected = X[:, selected_features]
    clf = RandomForestClassifier(n_estimators=50)
    score = cross_val_score(clf, X_selected, y, cv=5).mean()
    return score

# Initialize wolves
wolves = np.random.uniform(-1, 1, (num_wolves, num_features))
binary_wolves = np.array([binary_transform(w) for w in wolves])
fitness_vals = np.array([fitness(w) for w in binary_wolves])

# Initialize alpha, beta, delta
alpha_idx = np.argmax(fitness_vals)
alpha = wolves[alpha_idx].copy()
alpha_score = fitness_vals[alpha_idx]

beta_idx = np.argsort(fitness_vals)[-2]
beta = wolves[beta_idx].copy()
beta_score = fitness_vals[beta_idx]

delta_idx = np.argsort(fitness_vals)[-3]
delta = wolves[delta_idx].copy()
delta_score = fitness_vals[delta_idx]

# Main loop
for t in range(max_iter):
    a = 2 - t * (2 / max_iter) # Linearly decreasing a

    for i in range(num_wolves):
        for j in range(num_features):
            r1, r2 = np.random.rand(), np.random.rand()
            A1 = 2 * a * r1 - a
            C1 = 2 * r2
            D_alpha = abs(C1 * alpha[j] - wolves[i][j])
            X1 = alpha[j] - A1 * D_alpha

```

```

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

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

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

# Update binary positions
binary_wolves = np.array([binary_transform(w) for w in wolves])
fitness_vals = np.array([fitness(w) for w in binary_wolves])

# Update alpha, beta, delta
sorted_idx = np.argsort(fitness_vals)[::-1]
alpha, alpha_score = wolves[sorted_idx[0]].copy(), fitness_vals[sorted_idx[0]]
beta, beta_score = wolves[sorted_idx[1]].copy(), fitness_vals[sorted_idx[1]]
delta, delta_score = wolves[sorted_idx[2]].copy(), fitness_vals[sorted_idx[2]]

print(f"Iteration {t+1}/{max_iter}, Best fitness: {alpha_score:.4f}")

# Best feature subset
best_features = np.where(binary_transform(alpha) == 1)[0]
print("Selected feature indices:", best_features)
print("Number of features selected:", len(best_features))

```

**Program 6:**

### Parallel Cellular Algorithms and Programs:

Parallel Cellular Algorithms are inspired by the functioning of biological cells that operate in a highly parallel and distributed manner. These algorithms leverage the principles of cellular automata and parallel computing to solve complex optimization problems efficiently. Each cell represents a potential solution and interacts with its neighbors to update its state based on predefined rules. This interaction models the diffusion of information across the cellular grid, enabling the algorithm to explore the search space effectively. Parallel Cellular Algorithms are particularly suitable for large-scale optimization problems and can be implemented on parallel computing architectures for enhanced performance.

#### Implementation Steps:

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

solution space.

- Evaluate Fitness: Evaluate the fitness of each cell based on the optimization function.
  - Update States: Update the state of each cell based on the states of its neighboring cells and predefined update rules.
  - Iterate: Repeat the evaluation and state updating process for a fixed number of iterations or until convergence criteria are met.
  - Output the Best Solution: Track and output the best solution found during the iterations.

### Algorithm:

Pseudocode  
 Input: Noise image  
 Define parameters  
 Gridize  $\leftarrow$  dimension I  
 neighborhood  $\leftarrow$  define neighbourhood\_struct  
 max-iteration  $\leftarrow$  maxIter  
 Smoothing-factor  $\epsilon \leftarrow$   
 Initialize cells  
 for each pixel  $(x, y)$  in I:  
 cells [ $x$ ][ $y$ ].state =  $I(x, y)$   
 cells [ $x$ ][ $y$ ].newstate  $\leftarrow$  0  
 for iter = 1 to max iterations do in parallel:  
 for each cell  $(x, y)$ :  
 neighbours  $\leftarrow$  get\_neighbours(cells [ $x$ ][ $y$ ],  
 neighbours)  
 neighbor\_avg  $\leftarrow$  avg\_state(neighbours)  
 cells [ $x$ ][ $y$ ].newstate  $\leftarrow (1 - \epsilon) * \text{cells } [x][y].$   
 state +  $\epsilon * \text{neighbor\_avg}$   
 end for  
 end for  
 Output  
 Desired image  $(x, y) = \text{cells } [x][y].\text{state}$   
 Iteration taking is 3  $\rightarrow$  pixels changed 21.20%

## Code:

```
import numpy as np  
import matplotlib.pyplot as plt  
from skimage import data, util
```

```

def get_neighbors_indices(row, col, max_row, max_col):
    neighbors = []
    for dr in [-1, 0, 1]:
        for dc in [-1, 0, 1]:
            if dr == 0 and dc == 0:
                continue
            nr, nc = row + dr, col + dc
            if 0 <= nr < max_row and 0 <= nc < max_col:
                neighbors.append((nr, nc))
    return neighbors

def pca_noise_reduction(image, max_iterations=10, sigma=15):
    rows, cols = image.shape
    denoised_image = image.copy().astype(float)

    for iteration in range(max_iterations):
        new_image = denoised_image.copy()

        for i in range(rows):
            for j in range(cols):
                neighbors = get_neighbors_indices(i, j, rows, cols)

                weights = []
                intensities = []

                for nr, nc in neighbors:
                    diff = abs(denoised_image[nr, nc] - denoised_image[i, j])
                    weight = np.exp(-diff / sigma)
                    weights.append(weight)
                    intensities.append(denoised_image[nr, nc])

                weights = np.array(weights)
                intensities = np.array(intensities)

                if weights.sum() > 0:
                    new_value = np.sum(weights * intensities) / np.sum(weights)
                    new_image[i, j] = new_value

        denoised_image = new_image

    return denoised_image.astype(np.uint8)

# Load sample grayscale image: "camera"
image = data.camera() # shape: (512, 512)

# Add salt & pepper noise

```

```

noisy_image = util.random_noise(image, mode='s&p', amount=0.05)
noisy_image = (noisy_image * 255).astype(np.uint8)

# Apply PCA-based denoising
denoised = pca_noise_reduction(noisy_image, max_iterations=10, sigma=20)

# ⚡ Print only the 5x5 pixel values for comparison
print("\nOriginal Image 5x5 patch:")
print(image[100:105, 100:105])

print("\nNoisy Image 5x5 patch:")
print(noisy_image[100:105, 100:105])

print("\nDenoised Image 5x5 patch:")
print(denoised[100:105, 100:105])

```

### **Program 7:**

#### **Optimization via Gene Expression Algorithms:**

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

#### **Implementation Steps:**

1. Define the Problem: Create a mathematical function to optimize.
2. Initialize Parameters: Set the population size, number of genes, mutation rate, crossover rate, and number of generations.
3. Initialize Population: Generate an initial population of random genetic sequences.
4. Evaluate Fitness: Evaluate the fitness of each genetic sequence based on the optimization function.
5. Selection: Select genetic sequences based on their fitness for reproduction.
6. Crossover: Perform crossover between selected sequences to produce offspring.
7. Mutation: Apply mutation to the offspring to introduce variability.
8. Gene Expression: Translate genetic sequences into functional solutions.
9. Iterate: Repeat the selection, crossover, mutation, and gene expression processes for a fixed number of generations or until convergence criteria are met.
10. Output the Best Solution: Track and output the best solution found during the iterations.

#### **Algorithm:**

Lab 2 Gene Expression Algorithm

1. Define fitness function  
fitness ( $x$ ): sum of squares of  $x^2$
2. Initialize parameters  
population size = 20  
new\_gene = 5  
gene\_min = -5.0  
gene\_max = 5.0  
mutation\_rate = 0.1  
crossover\_rate = 0.8  
generation = 30
3. Initialize Population  
for each individual in population  
create a vector of num-genes  
random values b/w gene-min & max.
4. for generation 1 to generation 30:
  - a. evaluation fitness for all individuals  
for each individual.  
calculate fitness (individual)
  - b. find the best individual so far  
if same : if improved.
  - c. print generation number, best  
fitness and best solution.
  - d. select parents.
  - e. generate new generation  
for pairs of parent  
perform Crossover with probability  
perform mutation on children.
5. Replace population with next generation

Output		Best Solution (genes)			
Gen	Best f_new	[	-2.006	0.0863	-18.552 -0.34
1	7.626874	[	-0.4556	-1.6406	0.8779 0.3
2	4.239207	[	-0.4556	-1.644	-0.8779 0.3
3	4.239207	[	-0.4556	-1.6406	0.8779 0.3
4	4.239207	[	-0.4556	-1.6406	0.8779 0.3

gen1: Best fitness = 8.41, Best X = 2.9  
 gen2: Best fitness = 8.41 Best X = 2.9  
 gen3: Best fitness = 8.41 Best X = 2.9  
 ...  
 Gen 10: Best fitness = 9.61 Best X = 3.1  
 Best Solution:  $x=3.1$ , fitness = 9.61,

~~1.06200~~

## Code:

```

import numpy as np
import matplotlib.pyplot as plt
from gplearn.genetic import SymbolicRegressor
from gplearn.functions import make_function
from gplearn.fitness import make_fitness

# Generate training data
X = np.linspace(-10, 10, 100).reshape(-1, 1)
y = X**2 + np.sin(X) # True function to approximate

# Define symbolic regressor
est_gp = SymbolicRegressor(
    population_size=500,
    generations=20,
    stopping_criteria=0.01,
)
  
```

```

    p_crossover=0.7,
    p_subtree_mutation=0.1,
    p_hoist_mutation=0.05,
    p_point_mutation=0.1,
    max_samples=0.9,
    verbose=1,
    parsimony_coefficient=0.001,
    random_state=42
)

# Fit model
est_gp.fit(X, y)

# Predict on training data
y_pred = est_gp.predict(X)

# Print discovered expression
print("\nDiscovered expression:")
print(est_gp._program)

# Plot the results
plt.figure(figsize=(10, 6))
plt.plot(X, y, label='True Function', color='blue')
plt.plot(X, y_pred, label='GEP Prediction', color='red', linestyle='--')
plt.legend()
plt.title("Gene Expression Programming (Symbolic Regression)")
plt.xlabel("X")
plt.ylabel("y")
plt.grid(True)
plt.show()

```