

**VISVESVARAYA TECHNOLOGICAL
UNIVERSITY**

“JnanaSangama”, Belgaum -590014, Karnataka.



LAB RECORD

Bio Inspired Systems (23CS5BSBIS)

Submitted by

Sameksha(1BM23CS292)

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 **Sameksha (1BM23CS292)**, 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

Sl. No.	Date	Experiment Title	Page No.
1	29/08/2025	Genetic Algorithm for Optimization Problems	4
2	12/09/2025	Particle Swarm Optimization for Function Optimization	8
3	10/10/2025	Ant Colony Optimization for the Traveling Salesman Problem	10
4	17/10/2025	Cuckoo Search (CS)	12
5	17/10/2025	Grey Wolf Optimizer (GWO)	16
6	07/11/2025	Parallel Cellular Algorithms and Programs	21
7	29/08/2025	Optimization via Gene Expression Algorithms	22

Github Link:

<https://github.com/sameksha-1bm23cs292/BIS-lab/tree/main>

Program 1

Genetic Algorithm for Optimization Problems:

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

Algorithm:

Experiment : 1		29/08/25		PAGE NO: 3 DATE: / /	
	GA				
	Genetic Algorithm : 5 main phases				
• Initialization					
• Fitness assignment					
• Selection					
• Crossover					
• Termination					
	$f(x) = x^2$				
STEPS:	Step 1: Select encoding technique (0 to 18)				
1) Selecting encoding technique (0 to 18)					
2) Select the initial population - "4"					
String	Initial no.	x	Fitness	Offspring	X
no.	Population	value	$f(x) = x^2$	prob.	value
1	01100	12	144	0.1247	12.47
2	11001	25	625	0.8411	84.11
3	00101	5	25	0.0216	2.16
4	10011	19	361	0.3125	31.25
Burn		1155			
Avg		288.75			
Max		625			
3) Select Mating Pool.					
String	Mating pool no.		Crossover	Offspring	X
no.	Pool		prob.	value	fitness
1	01100	4	01101	13	169
2	11001	4	11000	24	676
3	11001	2	11011	37	729
4	10011	2	10001	17	289
Sum		176.3			
Avg		44.075			
Max		72.9			
4) Crossover Random:	4 1 4 2				
5) Mutation					
String	Offspring		Crossover	Offspring	X
no.	after		prob.	value	fitness
1	01101	10000	11011	29	841
2	11000	0.0000	11000	24	676
3	10011	0.0000	11011	37	729
4	10001	00101	10100	20	400
Sum				8546	
Avg				2136.5	
Max				841	
5) Selection of fit & un-fit - 841					
6) New Population containing 841					
7) Repeat until fit & un-fit are same					

<p>Step 0: Pseudocode :</p> <p>START</p> <ol style="list-style-type: none"> 1. Define fitness function : $\text{fitness}(x) = x^2$ 2. Initialize parameters <ul style="list-style-type: none"> ◦ population size = 6 ◦ number of generations = 5 ◦ mutation rate = 0.1 ◦ crossover rate = 0.7 ◦ chromosome length = 5 bits to represent numbers 0 to 81 3. Create initial population : <ul style="list-style-type: none"> ◦ generate 6 random 5bit binary strings. 4. For each generation from 1 to 5 do : <ol style="list-style-type: none"> a] evaluate fitness of each individual <ul style="list-style-type: none"> - convert binary string of each to individual integer x - calculate fitness (x) = x^2 b] Select parents based on fitness. c] Perform crossover . <ul style="list-style-type: none"> ◦ For pairs of parents with probability 0.7, swap bits after random crossover point. d] perform mutation : <ul style="list-style-type: none"> ◦ for each bit in offspring, flip bit with probability 0.1 e] Replace population with new offspring f] keep track of the best individual found so far. 	<p style="text-align: right;">PAGE EDGE DATE: / /</p> <p style="text-align: center;">AJP</p> <p>5. After last generation output the best solution :</p> <ul style="list-style-type: none"> - Decode best individual's binary to integer x. - Output x and fitness (x) <p>OUTPUT :</p> <p>Generation 1: Best Individual = 10010 ($x = 18$), Fitness = 324</p> <p>Generation 2: Best Individual = 11010 ($x = 26$), Fitness = 676</p> <p>Generation 3: Best Individual = 11010 ($x = 26$), Fitness = 676</p> <p>Generation 4: Best Individual = 11010 ($x = 26$), Fitness = 676</p> <p>Generation 5: Best Individual = 11011 ($x = 27$), Fitness = 729</p> <p>Best Solution found : 11011 ($x = 27$), Fitness = 729</p>
---	---

Code:

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

1. Define the function to optimize

```
def func(x):
    return x * x # Adjusted for new range
```

2. Parameters

```
POP_SIZE = 50
MUTATION_RATE = 0.01
CROSSOVER_RATE = 0.8
GENERATIONS = 10
CHROMOSOME_LENGTH = 10 # Now allows x in [0, 1023]
```

3. Decode chromosome to integer

```
def decode(chromosome):
    return int("".join(str(bit) for bit in chromosome), 2)
```

4. Create initial population

```
def create_population():
    return np.random.randint(2, size=(POP_SIZE, CHROMOSOME_LENGTH))
```

```

# 5. Evaluate fitness
def evaluate_fitness(population):
    decoded = np.array([decode(chrom) for chrom in population])
    fitness = func(decoded)
    return fitness

# 6. Selection (Roulette Wheel)
def select(population, fitness):
    min_fitness = np.min(fitness)
    if min_fitness < 0:
        fitness = fitness - min_fitness + 1e-6
    total_fitness = np.sum(fitness)
    probabilities = fitness / total_fitness
    indices = np.random.choice(np.arange(POP_SIZE), size=POP_SIZE, p=probabilities)
    return population[indices]

# 7. Crossover (Single-point)
def crossover(population):
    new_population = []
    for i in range(0, POP_SIZE, 2):
        parent1 = population[i]
        parent2 = population[(i + 1) % POP_SIZE]
        if np.random.rand() < CROSSOVER_RATE:
            point = np.random.randint(1, CHROMOSOME_LENGTH - 1)
            child1 = np.concatenate([parent1[:point], parent2[point:]])
            child2 = np.concatenate([parent2[:point], parent1[point:]])
            new_population.extend([child1, child2])
        else:
            new_population.extend([parent1, parent2])
    return np.array(new_population)

# 8. Mutation
def mutate(population):
    for i in range(POP_SIZE):
        for j in range(CHROMOSOME_LENGTH):
            if np.random.rand() < MUTATION_RATE:
                population[i, j] = 1 - population[i, j]
    return population

# 9. Main GA loop
def genetic_algorithm():
    population = create_population()
    best_solution = None
    best_fitness = -np.inf
    best_fitness_list = []

    for generation in range(GENERATIONS):
        fitness = evaluate_fitness(population)
        max_idx = np.argmax(fitness)

```

```

current_best_fitness = fitness[max_idx]
current_best_solution = decode(population[max_idx])

# Update global best
print(f"Generation {generation + 1}: x = {current_best_solution}, f(x) = {current_best_fitness:.4f}")

best_fitness_list.append(current_best_fitness)

# Elitism
elite = population[max_idx].copy()

# GA steps
population = select(population, fitness)
population = crossover(population)
population = mutate(population)

# Preserve elite
population[np.random.randint(POP_SIZE)] = elite

# Plot fitness over generations
plt.figure(figsize=(10, 5))
plt.plot(range(1, GENERATIONS + 1), best_fitness_list, label='Best Fitness')
plt.xlabel('Generation')
plt.ylabel('Fitness')
plt.title('Best Fitness Over Generations')
plt.grid(True)
plt.legend()
plt.tight_layout()
plt.show()

return current_best_solution, current_best_fitness

# Run the GA
best_x, best_val = genetic_algorithm()
print(f"\nFinal Best Solution: x = {best_x}, f(x) = {best_val:.4f}")

```

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.

Lab 2

Experiment : 2

PSO

12/09/25

PAGE EDGE

DATE: / /

Particle Swarm Optimization

Algorithm Pseudo code

```

• P = particle initialization();
• for i = 1 to max:
  3 for each particle p in P do:
    fp = f(p)
    if fp is better than f(pbist):
      pbist = p;
    end
  end
  gbest = best p in P
  for each particle p in P do:
  
```

$$v_i^{t+1} = v_i^t + c_1 \cdot r_1 (p_{bi}^t - p_i^t) + c_2 \cdot r_2 (g_b^t - p_i^t)$$

$$p_i^{t+1} = p_i^t + v_i^{t+1}$$

end

end

Fitness function : De Jong function

$$\min F(x, y) = x^2 + y^2$$

where x and y are the dimensions of the problem, the velocities of all the particles are initialized to zero and inertia (w) = 0.3 and the value of the cognitive and social constants $c_1 = 2$ and $c_2 = 2$.

The initial best solutions of all the particles are set to 1000. P_i fitness value = $1^2 + 1^2 = 2$.

particle	initial positions		BestSol	Best	fitness
	x	y			
P ₁	1	1	1000	-	2
P ₂	-1	1	1000	-	2
P ₃	0.5	-0.5	1000	-	0.5
P ₄	1	-1	1000	-	2
P ₅	0.25	0.25	1000	-	0.125

particle	BestSol	
	x	y
P ₁	1	1
P ₂	-1	1
P ₃	0.5	-0.5
P ₄	1	-1
P ₅	0.25	0.25

Output :

Best Position : 2.5

Best = 26.2500

```

import random
from math import sqrt

c1, c2 = 1, 1

def fitness(x):
    return -x**2 + 5*x + 20

def init():
    n = int(input("Enter no. of particles: "))
    v = [0 for i in range(n)]
    x = list(map(float, input("Enter positions of particles:").split()))
    p = x.copy()
    fp = [fitness(xi) for xi in x]
    return n, v, fp, p, x

def find(n, fp, p):
    max_fitness = float('-inf')
    pos = -1
    for i in range(n):
        if fp[i] > max_fitness:
            max_fitness = fp[i]
            pos = i
    return pos

def update(n, v, fp, p, x, max_pos):
    r1, r2 = sqrt(random.random()), sqrt(random.random())

    for i in range(n):
        v[i] = v[i] + c1 * r1 * (p[i] - x[i]) + c2 * r2 * (p[max_pos] - x[i])
        x[i] = x[i] + v[i]

    for i in range(n):
        fp[i] = fitness(x[i])
        if fp[i] > fitness(p[i]):
            p[i] = x[i]

def print_state(v, fp, p, x):
    print(f"\n{x}\n{p}\n{v}\n{fp}\n\"")

```

```

n, v, fp, p, x = init()
print_state(v, fp, p, x)
max_pos = find(n, fp, p)
gbest = p[max_pos]

while True:
    update(n, v, fp, p, x, max_pos)
    max_pos = find(n, fp, p)
    if fitness(gbest) == fitness(p[max_pos]):
        break
    print_state(v, fp, p, x)
    gbest = p[max_pos]

print(f"Global Best Solution: {gbest} with fitness: {fitness(gbest)}")

```

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.

ACO

Ant Colony Optimization for the
Travelling Salesman Problem

Pseudocode :

Input : list of cities with coordinates.

Compute distance between 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 = $(\text{pheromone}^{\alpha}) * (1/\text{distance})^{\beta}$

complete the route by returning to start

routes.append(route)

compute distance for route

If distance < best_distance :

update best route and best distance.

for each child in newpopulation :

compute tour length lk

update best tour and total distance

END.

```
import numpy as np
import random
```

```
def initialize_pheromone(num_cities, initial_pheromone=1.0):
    return np.ones((num_cities, num_cities)) * initial_pheromone
```

```
def calculate_probabilities(pheromone, distances, visited, alpha=1, beta=2):
    pheromone = np.copy(pheromone)
    pheromone[list(visited)] = 0 # zero out visited cities
```

```
heuristic = 1 / (distances + 1e-10) # inverse of distance
heuristic[list(visited)] = 0
```

```

prob = (pheromone ** alpha) * (heuristic ** beta)
total = np.sum(prob)
if total == 0:
    # If no options (all visited), choose randomly among unvisited
    choices = [i for i in range(len(distances)) if i not in visited]
    return choices, None
prob = prob / total
return range(len(distances)), prob

def select_next_city(probabilities, cities):
    if probabilities is None:
        return random.choice(cities)
    return np.random.choice(cities, p=probabilities)

def path_length(path, distances):
    length = 0
    for i in range(len(path)):
        length += distances[path[i-1]][path[i]]
    return length

def ant_colony_optimization(distances, n_ants=5, n_iterations=50, decay=0.5, alpha=1, beta=2):
    num_cities = len(distances)
    pheromone = initialize_pheromone(num_cities)
    best_path = None
    best_length = float('inf')

    for iteration in range(n_iterations):
        all_paths = []
        for _ in range(n_ants):
            path = [0] # start at city 0
            visited = set(path)

            for _ in range(num_cities - 1):
                current_city = path[-1]
                cities, probabilities = calculate_probabilities(pheromone[current_city],
                distances[current_city], visited, alpha, beta)
                next_city = select_next_city(probabilities, cities)
                path.append(next_city)
                visited.add(next_city)

            length = path_length(path, distances)
            all_paths.append((path, length))

        if length < best_length:
            best_length = length
            best_path = path

    # Evaporate pheromone

```

```

pheromone *= (1 - decay)

# Deposit pheromone proportional to path quality
for path, length in all_paths:
    deposit = 1 / length
    for i in range(len(path)):
        pheromone[path[i-1]][path[i]] += deposit

return best_path, best_length

# Example usage
if __name__ == "__main__":
    distances = np.array([
        [np.inf, 2, 2, 5, 7],
        [2, np.inf, 4, 8, 2],
        [2, 4, np.inf, 1, 3],
        [5, 8, 1, np.inf, 2],
        [7, 2, 3, 2, np.inf]
    ])

    best_path, best_length = ant_colony_optimization(distances)
    print(f'Best path: {[int(city) for city in best_path]} with length: {best_length:.2f}')

```

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.

Experiment 4 CS

Cuckoo Search Algorithm.

To implement cuckoo search algorithm for solving optimization problems inspired by the brood parasitic behavior of cuckoo birds.

It mimics the breeding behavior of cuckoo birds, which lay their eggs in the nests of other host birds.

If a host bird discovers an alien egg, it either abandons the nest / removes the egg away.

This represents the abandonment step in the algorithm.

- Each cuckoo lays 1 egg (new solution) in a randomly chosen nest.
- The best nests are carried over to the next generation.
- A fraction ρ_a of the worst nests is abandoned & replaced by new random ones.

Applications :

Optimization Problems
Neural network training
Scheduling and routing problems
Knapsack and TSP
Engineering Design Optimization

PAGE EDGE
DATE : / /

BEGIN CUCKOO SEARCH

INITIALIZE n nests $x_i (i=1 \dots n)$ randomly

Define fitness function $f(x)$

Set parameters: $\rho_a \in (0, 1)$, a , MaxIter

Evaluate $f(x_i)$ for all nests

Find best solution (x_{best})

For $t = 1$ to MaxIter:

For each cuckoo:

$$x_i^{new} = x_i + a * Levy(\lambda)$$

Evaluate $f(x_i^{new})$

$i \leftarrow \text{random}(1 \dots n)$

If $f(x_i^{new}) \geq f(x_i)$:

$x_i \leftarrow x_i^{new}$

End if

End for

Replace portion ρ_a of worst nests with new random ones

$x_{best} \leftarrow \text{best among all } x_i$

End for

Output x_{best} and $f(x_{best})$

END cuckoo search

Output :

Best Solution Found :

$[1 \ 1 \ 1 \ 0]$

Total Value : 360

Total Weight : 100

```
import numpy as np
```

```
import math
```

```
def knapsack_fitness(solution, values, weights, capacity):
```

```
    """Calculate fitness: total value if weight within capacity, else zero."""
```

```
    total_weight = np.sum(solution * weights)
```

```
    if total_weight > capacity:
```

```
        return 0 # Penalize overweight solutions
```

```
    return np.sum(solution * values)
```

```
def levy_flight(Lambda, size):
```

```
    """Generate Levy flight steps."""
```

```
    sigma = (math.gamma(1 + Lambda) * math.sin(math.pi * Lambda / 2) /
```

```
(math.gamma((1 + Lambda) / 2) * Lambda * 2 ** ((Lambda - 1) / 2))) ** (1 / Lambda)
u = np.random.normal(0, sigma, size)
v = np.random.normal(0, 1, size)
step = u / (np.abs(v) ** (1 / Lambda))
return step
```

```
def sigmoid(x):
    """Sigmoid function for mapping continuous to probability."""
    return 1 / (1 + np.exp(-x))
```

```
def cuckoo_search_knapsack(values, weights, capacity, n_nests=25, max_iter=100, pa=0.25):
    """
    Cuckoo Search for 0/1 Knapsack Problem.
```

Args:

- values: numpy array of item values
- weights: numpy array of item weights
- capacity: max capacity of knapsack
- n_nests: number of nests (population size)
- max_iter: max iterations
- pa: probability of abandoning nests

Returns:

- best_solution: binary numpy array with item selection
- best_fitness: total value of best_solution

"""

```
n_items = len(values)
nests = np.random.randint(0, 2, size=(n_nests, n_items))
fitness = np.array([knapsack_fitness(n, values, weights, capacity) for n in nests])
```

```
best_idx = np.argmax(fitness)
best_solution = nests[best_idx].copy()
best_fitness = fitness[best_idx]
```

```
Lambda = 1.5 # Levy flight exponent
```

```
for iteration in range(max_iter):
    for i in range(n_nests):
        step = levy_flight(Lambda, n_items)
        current = nests[i].astype(float)
        new_solution_cont = current + step
        probs = sigmoid(new_solution_cont)
        new_solution_bin = (probs > 0.5).astype(int)
```

```

new_fitness = knapsack_fitness(new_solution_bin, values, weights, capacity)

# Greedy selection
if new_fitness > fitness[i]:
    nests[i] = new_solution_bin
    fitness[i] = new_fitness

    if new_fitness > best_fitness:
        best_fitness = new_fitness
        best_solution = new_solution_bin.copy()

# Abandon worst nests with probability pa
n_abandon = int(pa * n_nests)
if n_abandon > 0:
    abandon_indices = np.random.choice(n_nests, n_abandon, replace=False)
    for idx in abandon_indices:
        nests[idx] = np.random.randint(0, 2, n_items)
        fitness[idx] = knapsack_fitness(nests[idx], values, weights, capacity)

# Update global best after abandonment
current_best_idx = np.argmax(fitness)
if fitness[current_best_idx] > best_fitness:
    best_fitness = fitness[current_best_idx]
    best_solution = nests[current_best_idx].copy()

# Print progress: every 10 iterations and first iteration
if iteration == 0 or (iteration + 1) % 10 == 0:
    print(f"Iteration {iteration + 1}/{max_iter}, Best Fitness: {best_fitness}")

return best_solution, best_fitness

if __name__ == "__main__":
    # Example knapsack problem
    values = np.array([60, 100, 120, 80, 30])
    weights = np.array([10, 20, 30, 40, 50])
    capacity = 100

    best_sol, best_val = cuckoo_search_knapsack(values, weights, capacity, n_nests=30,
max_iter=100, pa=0.25)

    print("\nBest solution found:")
    print(best_sol)

```

```
print("Total value:", best_val)
print("Total weight:", np.sum(best_sol * 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.

Experiment 5
LAB 6
(GWO)

Grey Wolf Optimization

The GWO is a nature inspired metaheuristic based on the social hierarchy and hunting behavior of grey wolves.

It mimics how wolves encircle, hunt and attack prey, using:

- α, β, δ to guide the search process.
- α (alpha) → best solution (leader)
- β (beta) → second best
- δ (delta) → third best
- ω (omega remaining)

- Simple and few control parameters
- Efficient balance between exploration & exploitation
- works well for continuous optimization problems

PAGE EDGE
DATE: / /

```

BEGIN GREY_WOLF_OPTIMIZER
Initialize N wolves  $x_i$  randomly within bounds
Evaluate fitness  $f(x_i)$ 
Identify  $\alpha, \beta, \gamma$  (best 3 wolves)
For  $t = 1$  to MaxIter:
     $a = \alpha * (1 - t / MaxIter)$ 
    For each wolf  $X$  in population:
        For each leader  $L$  in  $(\alpha, \beta, \gamma)$ :
             $y_1, y_2 = \text{rand}(0, 1)$ 
             $A = \alpha * a + \alpha(1 - a) * L$ 
             $C = \beta * y_2$ 
             $D = 1 - C * X_L - X_1$ 
             $X_{\text{candidate}} = X_1 - A * D - 1$ 
        End for
         $X_{\text{new}} = (\alpha * \text{candidate} + \beta * \text{candidate} + \gamma * \text{candidate}) / 3$ 
         $X_{\text{new}} = \text{clip}(X_{\text{new}}, \text{lower bounds}, \text{upper bound})$ 
        Evaluate  $f(X_{\text{new}})$ 
        Update  $\alpha, \beta, \gamma$  by ranking population
    End for
    Output  $\alpha, \beta, \gamma$  (best 3)
END

```

Enter number of wolves : 5
 Enter number of dimensions : 4
 Enter max iterations : 10
 Best Position : [1.93066889 -1.53882047
 -1.64847339 0.64182997]
 Best Score : 8.895300799871155

```
import numpy as np
```

```
def sphere(x):
```

```
    return np.sum(x**2)
```

```
class GreyWolfOptimizer:
```

```
    def __init__(self, obj_func, n_wolves, dim, max_iter, lb=-10, ub=10):
```

```
        self.obj_func = obj_func
```

```
        self.n_wolves = n_wolves
```

```
        self.dim = dim
```

```
        self.max_iter = max_iter
```

```
        self.lb = lb
```

```
        self.ub = ub
```

```
    self.positions = np.random.uniform(self.lb, self.ub, (self.n_wolves, self.dim))
```

```
    self.alpha_pos = np.zeros(self.dim)
```

```
    self.alpha_score = float('inf')
```

```

self.beta_pos = np.zeros(self.dim)
self.beta_score = float('inf')

self.delta_pos = np.zeros(self.dim)
self.delta_score = float('inf')

def optimize(self):
    for iter in range(self.max_iter):
        for i in range(self.n_wolves):
            self.positions[i] = np.clip(self.positions[i], self.lb, self.ub)

            fitness = self.obj_func(self.positions[i])

            if fitness < self.alpha_score:
                self.alpha_score = fitness
                self.alpha_pos = self.positions[i].copy()
            elif fitness < self.beta_score:
                self.beta_score = fitness
                self.beta_pos = self.positions[i].copy()
            elif fitness < self.delta_score:
                self.delta_score = fitness
                self.delta_pos = self.positions[i].copy()

        a = 2 - iter * (2 / self.max_iter)

        for i in range(self.n_wolves):
            for j in range(self.dim):
                r1 = np.random.rand()
                r2 = np.random.rand()
                A1 = 2 * a * r1 - a
                C1 = 2 * r2
                D_alpha = abs(C1 * self.alpha_pos[j] - self.positions[i, j])
                X1 = self.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 * self.beta_pos[j] - self.positions[i, j])
                X2 = self.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 * self.delta_pos[j] - self.positions[i, j])
X3 = self.delta_pos[j] - A3 * D_delta

self.positions[i, j] = (X1 + X2 + X3) / 3

return self.alpha_pos, self.alpha_score

if __name__ == "__main__":
    # Take inputs from user
    n_wolves = int(input("Enter number of wolves: "))
    dim = int(input("Enter number of dimensions: "))
    max_iter = int(input("Enter max iterations: "))

    gwo = GreyWolfOptimizer(obj_func=sphere, n_wolves=n_wolves, dim=dim, max_iter=max_iter)
    best_pos, best_score = gwo.optimize()

    print(f"Best Position: {best_pos}")
    print(f"Best Score: {best_score}")

```

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.

Experiment 6

LAB 7

Parallel Cellular Algorithm

cellular automata and biological cell interactions.

It represents each cell as a potential solution where cells interact locally within a neighborhood and evolve collectively toward the global optimum.

- Combines parallelism, local interaction & distributed search for efficient optimization

Each cell \rightarrow one candidate solution.
Each cell interacts only with nearby cells.
All cells update their state simultaneously.

Global behavior emerges from local interactions.

Advantages:

- parallel and scalable
- combines local and global search naturally
- reduces computation time using distributed update

PAGE EDGE

DATE: / /

BEGIN PCA

1. Define problem and fitness function
 $f(x)$

2. Initialize Parameters:

grid size \times
num cells = grid size \times 2
neighborhood $\approx 3 \times 3$
maxitter ≈ 100

3. Initialize each cell x_i randomly within search range.

4. Evaluate fitness $f(x_i)$

5. For $i = 1$ to maxitter do:

For each cell i in parallel:

Identify N_i :
Find best neighbor $n-best$ in N_i
Update x_i

6. Output the best solution

END.

```
import numpy as np
```

```
# Initialize
```

```
grid = np.random.uniform(low=-10, high=10, size=(10, 10))
```

```
num_iterations = 100
```

```
# Define fitness function
```

```

def fitness_function(x):
    return x**2 - 4*x + 4

# Iterate
for iteration in range(num_iterations):
    new_grid = np.zeros_like(grid)
    for r in range(grid.shape[0]):
        for c in range(grid.shape[1]):
            neighbor_values = []
            for dr in [-1, 0, 1]:
                for dc in [-1, 0, 1]:
                    nr = (r + dr) % grid.shape[0]
                    nc = (c + dc) % grid.shape[1]
                    neighbor_values.append(grid[nr, nc])
            # Update to average of neighbor values (per algorithm spec)
            new_grid[r, c] = np.mean(neighbor_values)
    grid = new_grid.copy()

# Find best solution
fitness_values = fitness_function(grid)
best_fitness_overall = np.min(fitness_values)
best_x_overall = grid[np.unravel_index(np.argmin(fitness_values), grid.shape)]

# Verbose Output
print("== Parallel Cellular Algorithm Results ==")
print(f"Total iterations performed: {num_iterations}")
print(f"Best x value found: {best_x_overall:.6f}")
print(f"Corresponding fitness (minimum f(x)): {best_fitness_overall:.6f}")
print("Algorithm converged toward x ≈ 2, where f(x) = 0 (expected optimum).")

```

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.

Experiment 1

GEA

Optimization via Gene Expression Algorithm

Steps:

START

1. Fitness function: $f(x) = x^2$

Encoding technique: 0 to 31

Use chromosome of fixed length (genotype)

2. Initial Population.

S. No.	Genotype	Phenotype	Value	Fitness	P	Actual Count	Exps Count
	Initial (exp.)	(x)	$f(x)$				
1	$+xx$	x^2	12	144	0.1241	1	1
2	$+xx$	$2x^2$	25	625	0.5411	2	0.5
3	x	x	5	25	0.0216	2	2.1
4	$-x2$	$x-2$	19	361	0.3125	0	0.08
Sum:				1155			1.25
Avg				288.75			
M _{b2}				68.5			

3. Selection of Mating Pool

S. No.	Selected Chromosome	Crossover point	Offspring	Phenotype	x	Value	Fitness
1	$+xx$	2	$*x + x^2(x-)$	x^2	13	169	
2	$+xx$	1	$+xx$	$2x^2$	24	576	
3	$+xx$	3	$+x - x^2(x_)$	$x - x^2$	27	729	
4	$-x2$	1	$+x_2$	$x+2$	17	289	

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

5. Mutation

S.No.	Offspring	Mutation applied	Offspring Phenotype	x value	Fitness
1	$*x + + \rightarrow - *x - x^2(x-)$		x^2	29	841
2	$+xx$	None	$+xx$	2x	24
3	$+x -$	$\rightarrow + - x + x+x^2x$	$x - x^2$	27	729
4	$+x_2$	None	$+x_2$	$x+2$	20

6. Gene expression and evaluation
decode each genotype \rightarrow phenotype
Calculate fitness

$$\sum f(x) = 841 + 576 + 729 + 400 = 2546$$

$$\text{Avg} = 636.5$$

$$\text{Max} = 841$$

7. Iterate until convergence

Repeat Step 3 to 6 until fitness improvement is negligible / generation limit has reached.

Pseudocode

Define Fitness function

Define Parameters

Create Population

Select mating pool

mutation after mating

Gene expression and evaluation

Iterate

Output Best Value

Output

1000 generations

Genes: [29.33, 29.82, 29.84, 28.57,
 15.09, 21.83, 23.83, 30.81,
 28.61, 26.22]

 $x = 26.37$ $f(x) = 695.45$

④

```

import random
import math

# Example: f(x) = x * sin(10*pi*x) + 2
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
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()
```