

# **VISVESVARAYA TECHNOLOGICAL UNIVERSITY**

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



## **LAB RECORD**

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

*Submitted by*

**Ayaan Shrestha (1BM23CS056)**

*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 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 **Ayaan Shrestha(1BM23CS056)**, 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.

Ms. Prameetha Pai 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	29/08/2025	Genetic Algorithm for Optimization Problems	1-8
2	12/09/2025	Optimization via Gene Expression Algorithms	9-14
3	10/10/2025	Ant Colony Optimization for the Traveling Salesman Problem	15-20
4	10/10/2025	Particle Swarm Optimization for Function Optimization	21-24
5	17/10/2025	Cuckoo Search for the Traveling Salesman Problem	25-29
6	24/10/2025	Grey Wolf Optimizer for Function Optimization	30-34
7	31/10/2025	Parallel Cellular Algorithm for Function Optimization	35-37

Github Link:

<https://github.com/ayaansth/BIOS-LAB>

## Program 1

### **Problem Statement:**

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

Find the best routes for delivery vehicles with limited capacity so that all customers are served once and total travel distance is minimized using a Genetic Algorithm.

### **Algorithm:**

Paper : "An enhanced genetic algorithm for parameter estimation of sinusoidal signals." S Date : 29/8  
LAB - 1 Page No. : \_\_\_\_\_

\* Genetic algorithm for optimisation problems

Genetic algorithm is a search and optimization technique inspired by the process of natural evolution. It evolves a population of potential solutions using.

- Selection
- crossover (Recombination)
- Mutation

Steps :

- 1) Define the problem  
To maximize the function  
 $f(x) = x \sin(10\pi x) + 1$  for  $x \in [0,1]$   
This function is non-linear with multiple peaks.  
 $\therefore$  It is a good test for GA's.
- 2) Initialise parameters  
variables how GA runs.  
pop-size, genes, mutation rate, crossover rate, generation  
 $x$  range  $\downarrow$  bit size  
 $(0,1)$
- 3) Encoding (Binary representation)  
initialise population  
i.e., each chromosome is a 16-bit string
- 4) Evaluate fitness  
measure of how good a solution is.  
Higher, the better.

Random selection based on the fitness proportion	
	Date : _____ Page No. : _____
5) Selection (roulette wheel)	we select parents to reproduce. Better val have higher chance of being chosen.
6) Crossover	combine 2 parents to create 2 children by swapping parts of their binary strings
	Crossover rate: probability that crossover happens
7) Mutation	(Random bit flip to introduce variation. → prevents premature convergence at to a local maximum by adding randomness.)
8) Run the GA for a fixed number of generations	
9) Display / select the best individual after all generations.	
* Pseudocode	
	Initialise population: create N random routes for gen = 1 to maxgen. for each route in population compute fitness select parents based on fitness perform crossover to produce offspring routes apply mutation to offspring form new population from offspring end for. Print out best route found.

### Code:

```
import math
import random
import copy
from typing import List, Tuple
```

```
random.seed(1)
```

```
# -----
# Utility functions
# -----
```

```
def euclidean(a: Tuple[float, float], b: Tuple[float, float]) -> float:
```

```

    return math.hypot(a[0] - b[0], a[1] - b[1])

def compute_distance_matrix(coords: List[Tuple[float, float]]) -> List[List[float]]:
    n = len(coords)
    dist = [[0.0]*n for _ in range(n)]
    for i in range(n):
        for j in range(n):
            dist[i][j] = euclidean(coords[i], coords[j])
    return dist

# -----
# Problem instance generator
# -----


def generate_instance(num_customers=15, grid_size=100, max_demand=10, depot=(50,50)):
    """
    Generates random customers with demands and coordinates.
    Index 0 is the depot, customers are 1..num_customers
    """
    coords = [depot]
    demands = [0]
    for _ in range(num_customers):
        coords.append((random.uniform(0, grid_size), random.uniform(0, grid_size)))
        demands.append(random.randint(1, max_demand))
    return coords, demands

# -----
# Chromosome -> Routes (split by capacity)
# -----


def split_routes_from_perm(perm: List[int], demands: List[int], capacity: int) -> List[List[int]]:
    """
    Greedy split: take customer sequence in perm, keep adding to current route
    while capacity is not exceeded. Start new route when needed.
    """
    routes = []
    cur_route = []
    cur_load = 0
    for cust in perm:
        d = demands[cust]
        if cur_load + d <= capacity:
            cur_route.append(cust)
            cur_load += d
        else:
            # close route and start new
            if cur_route:
                routes.append(cur_route)
            cur_route = []
            cur_load = 0
    if cur_route:
        routes.append(cur_route)
    return routes

```

```

cur_route=[cust]
cur_load = d
if cur_route:
    routes.append(cur_route)
return routes

def route_cost(route: List[int], dist_matrix: List[List[float]]) -> float:
    if not route:
        return 0.0
    cost = 0.0
    prev = 0 # depot
    for cust in route:
        cost += dist_matrix[prev][cust]
        prev = cust
    cost += dist_matrix[prev][0] # return to depot
    return cost

def total_cost(routes: List[List[int]], dist_matrix: List[List[float]]) -> float:
    return sum(route_cost(r, dist_matrix) for r in routes)

# -----
# Genetic Algorithm components
# -----


def init_population(pop_size: int, customer_ids: List[int]) -> List[List[int]]:
    pop = []
    for _ in range(pop_size):
        perm = customer_ids[:]
        random.shuffle(perm)
        pop.append(perm)
    return pop

def tournament_selection(pop: List[List[int]], fitnesses: List[float], k=3) -> List[int]:
    selected = random.sample(list(range(len(pop))), k)
    best = min(selected, key=lambda i: fitnesses[i])
    return copy.deepcopy(pop[best])

def order_crossover(parent1: List[int], parent2: List[int]) -> Tuple[List[int], List[int]]:
    """Order Crossover (OX) for permutations."""
    n = len(parent1)
    a, b = sorted(random.sample(range(n), 2))
    def ox(p1, p2):
        child = [None]*n
        # copy slice
        child[a:b+1] = p1[a:b+1]
        # fill remaining from p2 order
        pos = (b+1) % n

```

```

p2_i = (b+1) % n
while None in child:
    val = p2[p2_i]
    if val not in child:
        child[pos] = val
        pos = (pos+1) % n
    p2_i = (p2_i+1) % n
return child
return ox(parent1, parent2), ox(parent2, parent1)

def swap_mutation(perm: List[int], mut_rate=0.2) -> None:
    """In-place swap mutation (may perform multiple swaps based on mut_rate)."""
    n = len(perm)
    for i in range(n):
        if random.random() < mut_rate:
            j = random.randrange(n)
            perm[i], perm[j] = perm[j], perm[i]

# -----
# GA main loop
# -----


def evaluate_population(pop: List[List[int]], demands: List[int], capacity: int, dist_matrix: List[List[float]]):
    fitnesses = []
    for indiv in pop:
        routes = split_routes_from_perm(indiv, demands, capacity)
        fitnesses.append(total_cost(routes, dist_matrix))
    return fitnesses

def ga_vrp(
    coords: List[Tuple[float, float]],
    demands: List[int],
    capacity: int,
    pop_size=100,
    gens=300,
    crossover_rate=0.8,
    mutation_rate=0.15,
    elitism=2
):
    n = len(coords) - 1 # number of customers
    customer_ids = list(range(1, n+1))
    dist_matrix = compute_distance_matrix(coords)

    pop = init_population(pop_size, customer_ids)
    fitnesses = evaluate_population(pop, demands, capacity, dist_matrix)

```

```

best_history = []
for gen in range(gens):
    new_pop = []
    # Elitism: keep best individuals
    sorted_idx = sorted(range(len(pop)), key=lambda i: fitnesses[i])
    for i in range(elitism):
        new_pop.append(copy.deepcopy(pop[sorted_idx[i]]))

    while len(new_pop) < pop_size:
        # Selection
        p1 = tournament_selection(pop, fitnesses, k=3)
        p2 = tournament_selection(pop, fitnesses, k=3)
        # Crossover
        if random.random() < crossover_rate:
            c1, c2 = order_crossover(p1, p2)
        else:
            c1, c2 = p1, p2
        # Mutation
        swap_mutation(c1, mutation_rate)
        swap_mutation(c2, mutation_rate)
        new_pop.append(c1)
        if len(new_pop) < pop_size:
            new_pop.append(c2)

    pop = new_pop
    fitnesses = evaluate_population(pop, demands, capacity, dist_matrix)
    gen_best = min(fitnesses)
    best_history.append(gen_best)

    if (gen+1) % (max(1, gens//10)) == 0 or gen == 0:
        print(f'Gen {gen+1:4d} | best cost = {gen_best:.2f}')

# final best
best_idx = min(range(len(pop)), key=lambda i: fitnesses[i])
best_perm = pop[best_idx]
best_routes = split_routes_from_perm(best_perm, demands, capacity)
best_cost = fitnesses[best_idx]
return {
    'best_perm': best_perm,
    'best_routes': best_routes,
    'best_cost': best_cost,
    'dist_matrix': dist_matrix,
    'history': best_history
}

# -----
# Example / Run

```

```

# ----

if __name__ == "__main__":
    # Example instance
    NUM_CUSTOMERS = 12
    VEHICLE_CAPACITY = 30
    coords, demands = generate_instance(num_customers=NUM_CUSTOMERS, grid_size=100,
                                         max_demand=12, depot=(50,50))

    print("Depot coords:", coords[0])
    for i in range(1, len(coords)):
        print(f"Customer {i:2d} at {coords[i]} demand={demands[i]}")

    result = ga_vrp(
        coords=coords,
        demands=demands,
        capacity=VEHICLE_CAPACITY,
        pop_size=120,
        gens=300,
        crossover_rate=0.9,
        mutation_rate=0.12,
        elitism=4
    )

    print("\n==== BEST SOLUTION ===")
    print(f"Total cost: {result['best_cost']:.2f}")
    print("Routes (customers listed, depot implicit at start/end):")
    for idx, r in enumerate(result['best_routes'], start=1):
        load = sum(demands[c] for c in r)
        rcost = route_cost(r, result['dist_matrix'])
        print(f" Vehicle {idx}: {r} | load={load} | route_cost={rcost:.2f}")
    print("\nBest permutation (customer visit order):", result['best_perm'])

```

**Output:**

```
----- READING: D:/DATA/Engineering/VRP SEM NOTES/VRPS/VRP_Genetic.py -----
Depot coords: (50, 50)
Customer 1 at (13.436424411240122, 84.74337369372327) demand=2
Customer 2 at (25.50690257394217, 49.54350870919409) demand=8
Customer 3 at (47.224524357611664, 37.96152233237278) demand=4
Customer 4 at (9.385958677423488, 2.834747652200631) demand=7
Customer 5 at (43.27670679050534, 76.2280082457942) demand=1
Customer 6 at (69.58328667684435, 26.633056045725954) demand=4
Customer 7 at (59.11534350013039, 10.222715811004823) demand=6
Customer 8 at (3.0589983033553536, 2.54458609934608) demand=9
Customer 9 at (0.9204938554384978, 88.12338589221554) demand=11
Customer 10 at (21.659939713061338, 42.21165755827173) demand=1
Customer 11 at (52.76294143623982, 76.37009951314894) demand=8
Customer 12 at (55.28595762929651, 34.570041470875246) demand=11
Gen   1 | best cost = 483.53
Gen   30 | best cost = 393.75
Gen   60 | best cost = 381.97
Gen   90 | best cost = 381.97
Gen  120 | best cost = 381.97
Gen  150 | best cost = 381.97
Gen  180 | best cost = 381.97
Gen  210 | best cost = 381.97
Gen  240 | best cost = 381.97
Gen  270 | best cost = 381.97
Gen  300 | best cost = 381.97

==== BEST SOLUTION ====
Total cost: 381.97
Routes (customers listed, depot implicit at start/end):
  Vehicle 1: [9, 1, 5, 11] | load=22 | route_cost=142.14
  Vehicle 2: [12, 6, 7, 3] | load=25 | route_cost=94.66
  Vehicle 3: [2, 10, 8, 4] | load=25 | route_cost=145.16

Best permutation (customer visit order): [9, 1, 5, 11, 12, 6, 7, 3, 2, 10, 8, 4]
```

## Program 2

### **Problem Statement:**

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

Use Gene Expression Programming to find the best mathematical expression or variable values that optimize a given function.

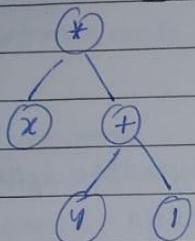
### **Algorithm:**

LAB - 2	
	Date : 9 Page No.: 12/9
* Gene expression Algorithm. ↓ process of using the information stored in genes to produce protein	
DNA $\xrightarrow{\text{transcription}}$ RNA $\xrightarrow{\text{translation}}$ Protein	
Transcription: copying of DNA sequence into RNA by RNA polymerase	
RNA processing • introns (non coding regions removed) by splicing • 5' cap tail added to pre-mRNA • Results: mature mRNA	
Translation: ribosomes read mRNA and build protein by linking amino acids together.	
Post-translation - new protein is modified to become functional.	
Application: Gene expression optimisation	
* Pseudocode <del>if</del> $f(x) = \sin(x)$ maximise Begin with a population algorithms each number represents a gene. for every gene in the population, run it into model through these stages (transcription, processing, export, translation, degradation) if any stage fails, product will have low fitness level.	

evaluate fitness : maximise  $\sin(x)$   
 $\sin(x)$

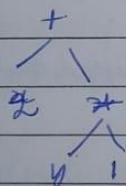
select & perform reproduction  
 loop through many generations reward best fitness  
 display best fitness.

expression tree for  $f(x, y) = x^+ (y + 1)$



expression tree for chromosome

$+2x^*y_1$



### Code:

```

import random
import math

# Target function to approximate
def target_function(x):
    return x**2 + x + 1

# Terminal and function sets
TERMINALS = ['x']
FUNCTIONS = ['+', '-', '*', '/']

# Parameters
POP_SIZE = 100
  
```

```

MUTATION_RATE = 0.1
TOURNAMENT_SIZE = 3
GENERATIONS = 100

# -----
# Utility functions
# -----


def safe_div(a, b):
    """Safe division to avoid divide-by-zero."""
    try:
        return a / b if b != 0 else 1
    except:
        return 1

def random_gene(max_depth=3):
    """Randomly create an expression (gene) of limited depth."""
    if max_depth == 0 or (max_depth > 1 and random.random() < 0.3):
        return random.choice(TERMINALS)
    else:
        op = random.choice(FUNCTIONS)
        left = random_gene(max_depth - 1)
        right = random_gene(max_depth - 1)
        return f"({left} {op} {right})"

def evaluate_expression(expr, x):
    """Safely evaluate the evolved expression."""
    try:
        # Define the environment so eval can use math and safe_div properly
        env = {"x": x, "math": math, "safe_div": safe_div}
        # Replace '/' with safe_div(a,b)
        expr_safe = expr.replace("/", "safe_div")
        result = eval(expr_safe, env)
        if callable(result): # in case a function is accidentally returned
            return 0
        return float(result)
    except Exception:
        return 0

def fitness(expr):
    """Mean squared error against target function."""
    xs = [i for i in range(-10, 11)]
    errors = []
    for x in xs:
        y_true = target_function(x)
        y_pred = evaluate_expression(expr, x)
        errors.append((y_true - y_pred)**2)

```

```

mse = sum(errors) / len(errors)
return mse

# -----
# Genetic operators
# -----


def mutate(expr):
    """Randomly mutate part of the expression."""
    expr_list = list(expr)
    for i in range(len(expr_list)):
        if random.random() < MUTATION_RATE:
            expr_list[i] = random.choice(['x', '+', '-', '*', '/'])
    return ''.join(expr_list)

def crossover(parent1, parent2):
    """Single-point crossover."""
    if len(parent1) < 2 or len(parent2) < 2:
        return parent1, parent2
    cut1 = random.randint(1, len(parent1) - 1)
    cut2 = random.randint(1, len(parent2) - 1)
    child1 = parent1[:cut1] + parent2[cut2:]
    child2 = parent2[:cut2] + parent1[cut1:]
    return child1, child2

def tournament_selection(pop, fits):
    """Select the best of k random individuals."""
    chosen = random.sample(list(zip(pop, fits)), TOURNAMENT_SIZE)
    return min(chosen, key=lambda x: x[1][0])

# -----
# Main GEP loop
# -----


def gene_expression_optimization():
    population = [random_gene() for _ in range(POP_SIZE)]

    for gen in range(GENERATIONS):
        fitnesses = [fitness(expr) for expr in population]
        best_idx = min(range(len(fitnesses)), key=lambda i: fitnesses[i])
        best_expr = population[best_idx]
        best_fit = fitnesses[best_idx]

        if gen % 10 == 0 or gen == GENERATIONS - 1:
            print(f"Generation {gen:3d} | Best Expr: {best_expr} | Fitness = {best_fit:.6f}")

        new_pop = [best_expr] # elitism

```

```

while len(new_pop) < POP_SIZE:
    p1 = tournament_selection(population, fitnesses)
    p2 = tournament_selection(population, fitnesses)
    c1, c2 = crossover(p1, p2)
    new_pop.extend([mutate(c1), mutate(c2)])
population = new_pop[:POP_SIZE]

print("\n✓ Optimization Complete!")
print(f'Best evolved expression: {best_expr}')
return best_expr

# -----
# Run
# -----
if __name__ == "__main__":
    best = gene_expression_optimization()

    print("\nSample comparison:")
    for x in [-3, 0, 2, 5]:
        print(f'x={x}>2} | Target={target_function(x):>8.3f} | Evolved={evaluate_expression(best, x):>8.3f}')

```

**Output:**

```

Generation 0 | Best Expr: (x*x) | Fitness = 37.666667
Generation 10 | Best Expr: (x+x*x) | Fitness = 1.000000
Generation 20 | Best Expr: (x+x*x) | Fitness = 1.000000
Generation 30 | Best Expr: (x+x*x) | Fitness = 1.000000
Generation 40 | Best Expr: (x+x*x) | Fitness = 1.000000
Generation 50 | Best Expr: (x+x*x) | Fitness = 1.000000
Generation 60 | Best Expr: (x+x*x) | Fitness = 1.000000
Generation 70 | Best Expr: (x+x*x) | Fitness = 1.000000
Generation 80 | Best Expr: (x+x*x) | Fitness = 1.000000
Generation 90 | Best Expr: (x+x*x) | Fitness = 1.000000
Generation 99 | Best Expr: (x+x*x) | Fitness = 1.000000

 Optimization Complete!
Best evolved expression: (x+x*x)

Sample comparison:
x=-3 | Target= 7.000 | Evolved= 6.000
x= 0 | Target= 1.000 | Evolved= 0.000
x= 2 | Target= 7.000 | Evolved= 6.000
x= 5 | Target= 31.000 | Evolved= 30.000

```

## Program 3

### **Problem Statement:**

### **Ant Colony Optimization for the Traveling Salesman Problem:**

Use Ant Colony Optimization to find the shortest possible route that visits all cities exactly once and returns to the starting city.

### **Algorithm:**

The image shows a handwritten lab report titled "LAB-3 Particle Swarm optimisation". At the top right, there is a logo and the text "Date : 10/10/25" and "Page No. :". The main text discusses PSO as a population-based algorithm inspired by the social behavior of birds flocking or fish schooling. It explains that each particle represents a candidate solution to the optimization problem, which "fly" through the search space by adjusting their position and velocity based on their own best position ( $P_{best}$ ) (personal best) and the best position of the entire swarm ( $G_{best}$ ) (global best). The goal is to find the position that minimizes/maximum of a given function. The report also outlines the algorithmic pseudocode and initial parameters, including the number of particles, maximum iterations, and coefficients  $c_1$  and  $c_2$ .

LAB-3  
Particle Swarm optimisation

Date : 10/10/25  
Page No. :

It is a population-based algorithm inspired by the social behaviour of birds flocking or fish schooling.

- each particle represents a candidate solution to the optimisation problem.
- particles 'fly' through the search space by adjusting their position and velocity based on:
  - their own best position ( $P_{best}$ ) (personal best)
  - best position of the entire swarm ( $G_{best}$ ) (global best)

Goal: Find position that minimises/maximum of a given function

\* Algorithmic pseudocode

1) Define Problem  
 $f(x,y) = x^2 + y^2$

2) Initialise parameters

- num-particles
- max-iter
- $w$
- $c_1$  cognitive coefficients
- $c_2$  social coefficients

<p>3) initialise particles      random position <math>x_i</math>      velocity <math>v_i</math>      personal best <math>pbest = x_i</math></p> <p>evaluate <math>f(pbest)</math></p> <p>4) determine global Best.      particle with best fitness</p> <p><math>gbest = \arg \min_i f(x_{best,i})</math></p> <p>5) update velocity &amp; position</p> $v_i(t+1) = w \cdot v_i(t) + c_1 M_1 (gbest_i - x_i(t)) + c_2 M_2 (gbest - x_i(t))$ $x_i(t+1) = x_i(t) + v_i(t+1)$ <p><math>M_1, M_2</math> : random nos b/w 0 &amp; 1  <math>w, v_i(t)</math> <math>\Rightarrow</math> inertia term</p> <p>6) update pbest &amp; gbest      if new pos <math>x_i(t+1)</math> gives better fitness, update</p> $pbest_i = x_i(t+1)$ $gbest = pbest_i$ <p>7) iterate till stopping conditions</p> <p>8) output <math>\rightarrow gbest</math>.</p>	<p><u>Application</u></p> <p>eq: <math>f(x) = x^2 + 4 \sin(x)</math></p> <ul style="list-style-type: none"> <li>num of particles = 10</li> <li>iteration = 50</li> <li>inertia wt <math>w = 0.7</math></li> <li><math>c_1 = 1.5 = c_2</math></li> <li><math>x \in [-10, 10]</math></li> </ul> <p>opt: Best solution  <math>x = -1.395</math>  <math>f(x) = -4.759</math></p> <p><math>\downarrow</math> minimize.</p>
---	---

### Code:

```

import random
import math

# -----
# PARAMETERS
# -----
NUM_CITIES = 10
NUM_ANTS = 20
ALPHA = 1.0      # pheromone importance
BETA = 5.0       # distance importance
RHO = 0.5        # evaporation rate
Q = 100          # pheromone deposit factor
ITERATIONS = 100

```

```
random.seed(1)
```

```

# -----
# CREATE TSP INSTANCE
# -----

```

```

def generate_cities(n, grid_size=100):
    """Randomly generate city coordinates."""
    return [(random.uniform(0, grid_size), random.uniform(0, grid_size)) for _ in range(n)]

def distance(a, b):
    return math.hypot(a[0] - b[0], a[1] - b[1])

def compute_distance_matrix(cities):
    n = len(cities)
    dist = [[0]*n for _ in range(n)]
    for i in range(n):
        for j in range(n):
            if i != j:
                dist[i][j] = distance(cities[i], cities[j])
    return dist

# -----
# ACO Core
# -----
def initialize_pheromones(n):
    """Initialize pheromone levels between cities."""
    return [[1.0 for _ in range(n)] for _ in range(n)]

def select_next_city(probabilities):
    """Roulette-wheel selection."""
    r = random.random()
    cumulative = 0.0
    for i, p in enumerate(probabilities):
        cumulative += p
        if r <= cumulative:
            return i
    return len(probabilities) - 1

def ant_tour(dist_matrix, pheromone):
    """Construct a complete tour for one ant."""
    n = len(dist_matrix)
    unvisited = list(range(n))
    start = random.choice(unvisited)
    tour = [start]
    unvisited.remove(start)

    while unvisited:
        current = tour[-1]
        probs = []
        denom = 0.0
        for j in unvisited:
            denom += (pheromone[current][j] ** ALPHA) * ((1.0 / dist_matrix[current][j]) ** BETA)
            probs.append(denom)
        tour.append(unvisited[probs.index(max(probs))])
        unvisited.remove(tour[-1])

```

```

for j in unvisited:
    num = (pheromone[current][j] ** ALPHA) * ((1.0 / dist_matrix[current][j]) ** BETA)
    probs.append(num / denom if denom > 0 else 0)
next_city = unvisited[select_next_city(probs)]
tour.append(next_city)
unvisited.remove(next_city)
return tour

def tour_length(tour, dist_matrix):
    """Calculate total length of the tour."""
    total = 0.0
    for i in range(len(tour) - 1):
        total += dist_matrix[tour[i]][tour[i+1]]
    total += dist_matrix[tour[-1]][tour[0]] # return to start
    return total

def update_pheromones(pheromone, all_tours, all_lengths):
    """Evaporate and deposit pheromones based on ant tours."""
    n = len(pheromone)
    # Evaporation
    for i in range(n):
        for j in range(n):
            pheromone[i][j] *= (1 - RHO)
            if pheromone[i][j] < 1e-6:
                pheromone[i][j] = 1e-6

    # Deposit pheromones
    for tour, L in zip(all_tours, all_lengths):
        deposit = Q / L
        for i in range(len(tour) - 1):
            a, b = tour[i], tour[i+1]
            pheromone[a][b] += deposit
            pheromone[b][a] += deposit
        # return edge
        pheromone[tour[-1]][tour[0]] += deposit
        pheromone[tour[0]][tour[-1]] += deposit

# -----
# ACO MAIN LOOP
# -----
def aco_tsp(num_cities=NUM_CITIES):
    cities = generate_cities(num_cities)
    dist_matrix = compute_distance_matrix(cities)
    pheromone = initialize_pheromones(num_cities)

    best_tour = None
    best_length = float('inf')

```

```

for it in range(ITERATIONS):
    all_tours = []
    all_lengths = []

    for ant in range(NUM_ANTS):
        tour = ant_tour(dist_matrix, pheromone)
        L = tour_length(tour, dist_matrix)
        all_tours.append(tour)
        all_lengths.append(L)

        if L < best_length:
            best_length = L
            best_tour = tour

    update_pheromones(pheromone, all_tours, all_lengths)

    if (it + 1) % 10 == 0 or it == 0:
        print(f"Iteration {it+1:3d} | Best Length: {best_length:.2f}")

    print("\n◇ Optimization Complete!")
    print(f"Best Tour Length: {best_length:.2f}")
    print(f"Best Tour: {best_tour}")
    return best_tour, best_length, cities

# -----
# RUN
# -----
if __name__ == "__main__":
    best_tour, best_length, cities = aco_tsp()

```

*Output:*

```
===== RESTART: D:/Anu/Engineering/5th sem nc
Iteration  1 | Best Length: 345.03
Iteration 10 | Best Length: 344.77
Iteration 20 | Best Length: 344.77
Iteration 30 | Best Length: 344.77
Iteration 40 | Best Length: 344.77
Iteration 50 | Best Length: 344.77
Iteration 60 | Best Length: 344.77
Iteration 70 | Best Length: 344.77
Iteration 80 | Best Length: 344.77
Iteration 90 | Best Length: 344.77
Iteration 100 | Best Length: 344.77

 Optimization Complete!
Best Tour Length: 344.77
Best Tour: [7, 0, 8, 3, 5, 1, 9, 6, 4, 2]
>|
```

## Program 4

### **Problem Statement:**

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

Apply Particle Swarm Optimization to find the optimal solution (minimum or maximum) of a given mathematical function.

### **Algorithm:**

LAB-4  
Ant colony optimisation  
for TSP.

1) TSP : n cities. Find shortest routes that visits all cities once and comes back to start.

2) numbs\_ant  
alpha : pheromone importance  
beta : heuristic's  
rho : pheromone evaporation rate  
Q : pheromone deposit constant  
pheromone\_init : initial value of rho for all paths  
max\_iter

3) each ant starts from a random city  
moves to next city based on

$$\sigma_{ij} = (\tau_{ij})^\alpha \cdot (\eta_{ij})^\beta$$

$$\eta_{ij} = 1/\eta_{ij}$$

$\tau_{ij}$  : pheromone on edge  $(i,j)$   
 $\eta_{ij}$  : heuristic's

4) update pheromones  
 - after all ants finish,  
 - update pheromone

$$\tau_{ij} = (1 - P) \cdot \tau_{ij}$$

$$\tau_{ij} += Q$$

Date : 10/10/25  
Page No.:

→ Iterate  
↓ Output

Example:  
 City 0 → (0,0)  
 City 1 → (0,2)  
 City 2 → (2,0)  
 City 3 → (2,2)

O/P : Best length = 3.00  
 Route : [0, 2, 3, 1]

Namita M.  
 10/10/25

### **Code:**

```
import random
import math

# Define the mathematical function to optimize (minimize)
def objective_function(x, y):
```

```

return x**2 + y**2 # Sphere function

# PSO parameters
num_particles = 30
max_iterations = 100
w = 0.7    # inertia weight
c1 = 1.5   # cognitive (particle) weight
c2 = 1.5   # social (swarm) weight

# Initialize particles
particles = []
for i in range(num_particles):
    x = random.uniform(-10, 10)
    y = random.uniform(-10, 10)
    velocity = [random.uniform(-1, 1), random.uniform(-1, 1)]
    fitness = objective_function(x, y)
    particles.append({
        "position": [x, y],
        "velocity": velocity,
        "best_pos": [x, y],
        "best_fit": fitness
    })

# Initialize global best
global_best = min(particles, key=lambda p: p["best_fit"])
global_best_pos = global_best["best_pos"]
global_best_fit = global_best["best_fit"]

# Main PSO loop
for t in range(max_iterations):
    for particle in particles:
        # Update velocity
        r1, r2 = random.random(), random.random()
        for i in range(2):
            cognitive = c1 * r1 * (particle["best_pos"][i] - particle["position"][i])
            social = c2 * r2 * (global_best_pos[i] - particle["position"][i])
            particle["velocity"][i] = w * particle["velocity"][i] + cognitive + social

        # Update position
        particle["position"][0] += particle["velocity"][0]
        particle["position"][1] += particle["velocity"][1]

        # Evaluate new fitness
        fit = objective_function(particle["position"][0], particle["position"][1])

        # Update personal best
        if fit < particle["best_fit"]:

```

```

particle["best_fit"] = fit
particle["best_pos"] = particle["position"][:]

# Update global best
if fit < global_best_fit:
    global_best_fit = fit
    global_best_pos = particle["position"][:]

print(f'Iteration {t+1}/{max_iterations} - Best Fitness: {global_best_fit:.6f}')

print("\n✓ Optimization complete!")
print(f'Best position found: {global_best_pos}')
print(f'Minimum value of function: {global_best_fit:.6f}')

```

**Output:**

---

```

Iteration 59/100 - Best Fitness: 0.000000
Iteration 60/100 - Best Fitness: 0.000000
Iteration 61/100 - Best Fitness: 0.000000
Iteration 62/100 - Best Fitness: 0.000000
Iteration 63/100 - Best Fitness: 0.000000
Iteration 64/100 - Best Fitness: 0.000000
Iteration 65/100 - Best Fitness: 0.000000
Iteration 66/100 - Best Fitness: 0.000000
Iteration 67/100 - Best Fitness: 0.000000
Iteration 68/100 - Best Fitness: 0.000000
Iteration 69/100 - Best Fitness: 0.000000
Iteration 70/100 - Best Fitness: 0.000000
Iteration 71/100 - Best Fitness: 0.000000
Iteration 72/100 - Best Fitness: 0.000000
Iteration 73/100 - Best Fitness: 0.000000
Iteration 74/100 - Best Fitness: 0.000000
Iteration 75/100 - Best Fitness: 0.000000
Iteration 76/100 - Best Fitness: 0.000000
Iteration 77/100 - Best Fitness: 0.000000
Iteration 78/100 - Best Fitness: 0.000000
Iteration 79/100 - Best Fitness: 0.000000
Iteration 80/100 - Best Fitness: 0.000000
Iteration 81/100 - Best Fitness: 0.000000
Iteration 82/100 - Best Fitness: 0.000000
Iteration 83/100 - Best Fitness: 0.000000
Iteration 84/100 - Best Fitness: 0.000000
Iteration 85/100 - Best Fitness: 0.000000
Iteration 86/100 - Best Fitness: 0.000000
Iteration 87/100 - Best Fitness: 0.000000
Iteration 88/100 - Best Fitness: 0.000000
Iteration 89/100 - Best Fitness: 0.000000
Iteration 90/100 - Best Fitness: 0.000000
Iteration 91/100 - Best Fitness: 0.000000
Iteration 92/100 - Best Fitness: 0.000000
Iteration 93/100 - Best Fitness: 0.000000
Iteration 94/100 - Best Fitness: 0.000000
Iteration 95/100 - Best Fitness: 0.000000
Iteration 96/100 - Best Fitness: 0.000000
Iteration 97/100 - Best Fitness: 0.000000
Iteration 98/100 - Best Fitness: 0.000000
Iteration 99/100 - Best Fitness: 0.000000
Iteration 100/100 - Best Fitness: 0.000000

 Optimization complete!
Best position found: [4.960805796605412e-07, -1.8244758085889348e-07]
Minimum value of function: 0.000000

```

## Program 5

### **Problem Statement:**

#### **Cuckoo Search (CS) for travel salesman problem :**

Use the Cuckoo Search Algorithm to determine the shortest route for a salesman to visit all cities and return to the starting point.

### **Algorithm:**

Lab-5 CUCKOO search Algorithm	PSEVMOLODE
<ul style="list-style-type: none"> <li>- used for optimization</li> <li>- uses breeding behaviour of cuckoo birds</li> <li>- cuckoo lay eggs in the nest of other birds</li> <li>- Meta-heuristic approach           <ul style="list-style-type: none"> <li>LEVY FLIGHT : helps find suitable nest.</li> </ul> </li> </ul> <p>cuckoo host discovery probability : <math>P_d \in [0, 1]</math></p> <ul style="list-style-type: none"> <li>• The host can discover eggs</li> <li>• - " - may not</li> <li>• Throw the egg</li> <li>• Build a new nest.</li> </ul>	<ol style="list-style-type: none"> <li>1. Set initial value of the host nest size <math>n</math>, probability <math>P_d \in [0, 1]</math> and max number of iteration <math>Max</math>.</li> <li>2. Set <math>t = 0</math></li> <li>3. For (<math>i=1</math>; <math>i \leq n</math>) do</li> <li>4. Generate initial population of <math>n</math> host <math>x^t_i</math>.</li> <li>5. Evaluate Fitness function <math>f(x^t_i)</math>.</li> <li>6. End for.</li> <li>7. Generate a new solution (cuckoo) <math>x^{t+1}_i</math> randomly by Levy Flight.</li> <li>8. Evaluate fitness function <math>x^{t+1}_i</math> i.e., <math>f(x^{t+1}_i)</math></li> <li>9. Choose a nest <math>x_j</math> among <math>n</math> solutions randomly.</li> <li>10. If <math>f(x^{t+1}_i) &gt; f(x_j)</math> then</li> <li>11. Replace the sol<sup>t</sup> <math>x_j</math> with the sol<sup>t+1</sup> <math>x^{t+1}_i</math></li> <li>12. End if.</li> <li>13. Abandon a fraction <math>P_a</math> of worst nest.</li> <li>14. Build new nest at new location.</li> <li>15. Keep the best sol<sup>t</sup></li> <li>16. Rank solution and find current sol<sup>t</sup></li> <li>17. Set <math>t = t + 1</math></li> <li>18. Until (<math>t \geq Max</math>)</li> <li>19. Procedure best sol<sup>t</sup></li> </ol>

### **Code:**

```

import random
import math

# -----
# Helper functions
# -----


def distance(city1, city2):
    """Euclidean distance between two cities"""

```

```

    return math.sqrt((city1[0] - city2[0])**2 + (city1[1] - city2[1])**2)

def total_distance(path, cities):
    """Compute total distance of the path"""
    dist = 0
    for i in range(len(path)):
        city1 = cities[path[i]]
        city2 = cities[path[(i + 1) % len(path)]]
        dist += distance(city1, city2)
    return dist

def levy_flight(Lambda):
    """Generate step size using Lévy distribution"""
    sigma = (math.gamma(1 + Lambda) * math.sin(math.pi * Lambda / 2) /
             (math.gamma((1 + Lambda) / 2) * Lambda * 2**((Lambda - 1) / 2)))**(1 / Lambda)
    u = random.gauss(0, sigma)
    v = random.gauss(0, 1)
    step = u / abs(v)**(1 / Lambda)
    return step

def get_new_solution(path):
    """Generate a new random permutation by swapping two cities"""
    new_path = path[:]
    i, j = random.sample(range(len(new_path)), 2)
    new_path[i], new_path[j] = new_path[j], new_path[i]
    return new_path

# -----
# Cuckoo Search Algorithm
# -----


def cuckoo_search_tsp(cities, n_nests=15, pa=0.25, alpha=1, Lambda=1.5, max_iter=100):
    # Initialize nests (random permutations)
    nests = [random.sample(range(len(cities)), len(cities)) for _ in range(n_nests)]
    fitness = [total_distance(nest, cities) for nest in nests]

    best_nest = nests[fitness.index(min(fitness))]
    best_fitness = min(fitness)

    for iteration in range(max_iter):
        for i in range(n_nests):
            # Lévy flight (small random move)
            step_size = levy_flight(Lambda)
            new_nest = get_new_solution(nests[i])

            new_fitness = total_distance(new_nest, cities)

```

```

# If better, replace
if new_fitness < fitness[i]:
    nests[i] = new_nest
    fitness[i] = new_fitness

# Update best
if new_fitness < best_fitness:
    best_nest = new_nest[:]
    best_fitness = new_fitness

# Abandon some nests with probability pa
for i in range(n_nests):
    if random.random() < pa:
        nests[i] = random.sample(range(len(cities)), len(cities))
        fitness[i] = total_distance(nests[i], cities)

print(f"Iteration {iteration+1}/{max_iter} - Best Distance: {best_fitness:.4f}")

return best_nest, best_fitness

# -----
# Example Run
# -----


if __name__ == "__main__":
    # Example: 10 cities with random coordinates
    cities = [(random.uniform(0, 100), random.uniform(0, 100)) for _ in range(10)]

    best_path, best_distance = cuckoo_search_tsp(cities, n_nests=20, max_iter=100)

    print("\n↙ Best Path Found:")
    print(best_path)
    print(f"Total Distance: {best_distance:.4f}")

```

**Output:**

```
Iteration 59/100 - Best Distance: 332.8408
Iteration 60/100 - Best Distance: 332.8408
Iteration 61/100 - Best Distance: 332.8408
Iteration 62/100 - Best Distance: 332.8408
Iteration 63/100 - Best Distance: 332.8408
Iteration 64/100 - Best Distance: 332.8408
Iteration 65/100 - Best Distance: 332.8408
Iteration 66/100 - Best Distance: 332.8408
Iteration 67/100 - Best Distance: 332.8408
Iteration 68/100 - Best Distance: 332.8408
Iteration 69/100 - Best Distance: 332.8408
Iteration 70/100 - Best Distance: 332.8408
Iteration 71/100 - Best Distance: 332.8408
Iteration 72/100 - Best Distance: 332.8408
Iteration 73/100 - Best Distance: 332.8408
Iteration 74/100 - Best Distance: 332.8408
Iteration 75/100 - Best Distance: 332.8408
Iteration 76/100 - Best Distance: 332.8408
Iteration 77/100 - Best Distance: 332.8408
Iteration 78/100 - Best Distance: 332.8408
Iteration 79/100 - Best Distance: 332.8408
Iteration 80/100 - Best Distance: 332.8408
Iteration 81/100 - Best Distance: 327.7554
Iteration 82/100 - Best Distance: 327.7554
Iteration 83/100 - Best Distance: 326.7949
Iteration 84/100 - Best Distance: 326.7949
Iteration 85/100 - Best Distance: 326.7949
Iteration 86/100 - Best Distance: 326.7949
Iteration 87/100 - Best Distance: 326.7949
Iteration 88/100 - Best Distance: 326.7949
Iteration 89/100 - Best Distance: 326.7949
Iteration 90/100 - Best Distance: 326.7949
Iteration 91/100 - Best Distance: 326.7949
Iteration 92/100 - Best Distance: 326.7949
Iteration 93/100 - Best Distance: 326.7949
Iteration 94/100 - Best Distance: 326.7949
Iteration 95/100 - Best Distance: 326.7949
Iteration 96/100 - Best Distance: 326.7949
Iteration 97/100 - Best Distance: 326.7949
Iteration 98/100 - Best Distance: 326.7949
Iteration 99/100 - Best Distance: 326.7949
Iteration 100/100 - Best Distance: 326.7949
```

Best Path Found:  
[6, 4, 2, 8, 5, 9, 7, 0, 1, 3]  
Total Distance: 326.7949

## Program 6

### **Problem Statement:**

### **Grey Wolf Optimizer (GWO) for Function Optimization:**

Use Grey Wolf Optimization to find the best solution for a mathematical function by mimicking the hunting behavior of grey wolves.

### **Algorithm:**

Date 24/10/25  
Page No.: \_\_\_\_\_

Date \_\_\_\_\_  
Page No.: \_\_\_\_\_

Week - 6  
Grey Wolf Optimisation

**Input:**  $f(x)$  → objective function to be minimized  
 $n$  → no. of wolves  
 $dim$  → dimension of the population  
 $maxIte$  → max iteration  
 $lb, ub$  → lower & upper bound

**O/P :**  $x_{\alpha}$  → best ( $\alpha$ ) solution found

1) Initialize population  $X_i$  ( $i=1$  to  $n$ )  
 $[lb, ub]$

2) evaluate fitness ( $x_i$ ) for each wolf

3) identify  
 $x_{\alpha}$  = best wolf  
 $x_{\beta}$  = second best wolf  
 $x_{\delta}$  = third best wolf

4) For  $t=1$  to MaxIter do:  
 $a = 2 - (2 * t / maxIte)$   
For each wolf  $i$  in population:  
For each dimension  $j$ :  
 $r_1 = \text{random}(0,1)$   
 $r_2 = \text{random}(0,1)$   
 $A_1 = 2 * a * r_1 - a$   
 $L_1 = 2 * r_2$   
 $D_{\alpha} = |C_j + x_{\alpha}[j] - X_i[j]|$   
 $X_i = x_{\alpha}[j] - A_1 * D_{\alpha}$

$r_1 = \text{random}(0,1)$   
 $r_2 = \text{random}(0,1)$   
 $A_2 = 2 * a * r_1 - a$   
 $L_2 = 2 * r_2$   
 $D_{\beta} = |C_j + x_{\beta}[j] - X_i[j]|$   
 $X_i = x_{\beta}[j] - A_2 * D_{\beta}$

$r_1 = \text{random}(0,1)$   
 $r_2 = \text{random}(0,1)$   
 $A_3 = 2 * a * r_1 - a$   
 $C_3 = 2 * r_2$   
 $D_{\delta} = |C_j + x_{\delta}[j] - X_i[j]|$   
 $X_i = x_{\delta}[j] - A_3 * D_{\delta}$

$x_i = \text{new}[j] = (X_1 + X_2 + X_3) / 3$

5) return  $x_{\alpha}$  as the best sol?

### **Code:**

```
import random
import math
```

```

# -----
# Objective Function (to minimize)
# -----
def objective_function(position):
    # Sphere function: sum of squares
    return sum(x**2 for x in position)

# -----
# Grey Wolf Optimizer
# -----
def grey_wolf_optimizer(num_wolves=20, dim=2, max_iter=100, lower_bound=-10,
upper_bound=10):
    # Initialize the population of wolves randomly
    wolves = [[random.uniform(lower_bound, upper_bound) for _ in range(dim)] for _ in
range(num_wolves)]
    fitness = [objective_function(w) for w in wolves]

    # Initialize alpha, beta, delta (best 3 wolves)
    alpha, beta, delta = [0]*dim, [0]*dim, [0]*dim
    alpha_score, beta_score, delta_score = float("inf"), float("inf"), float("inf")

    # Main loop
    for t in range(max_iter):
        for i in range(num_wolves):
            score = fitness[i]

            # Update alpha, beta, delta
            if score < alpha_score:
                delta_score, delta = beta_score, beta[:]
                beta_score, beta = alpha_score, alpha[:]
                alpha_score, alpha = score, wolves[i][:]
            elif score < beta_score:
                delta_score, delta = beta_score, beta[:]
                beta_score, beta = score, wolves[i][:]
            elif score < delta_score:
                delta_score, delta = score, wolves[i][:]

        # Linearly decreasing a from 2 to 0
        a = 2 - t * (2 / max_iter)

        # Update each wolf's position
        for i in range(num_wolves):
            new_position = []
            for j in range(dim):
                r1, r2 = random.random(), random.random()
                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 = random.random(), random.random()
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 = random.random(), random.random()
A3 = 2 * a * r1 - a
C3 = 2 * r2
D_delta = abs(C3 * delta[j] - wolves[i][j])
X3 = delta[j] - A3 * D_delta

new_x = (X1 + X2 + X3) / 3
# Keep within bounds
new_x = max(lower_bound, min(upper_bound, new_x))
new_position.append(new_x)

wolves[i] = new_position
fitness[i] = objective_function(new_position)

print(f"Iteration {t+1}/{max_iter} - Best fitness: {alpha_score:.6f}")

return alpha, alpha_score

# -----
# Example Run
# -----
if __name__ == "__main__":
    best_position, best_score = grey_wolf_optimizer(num_wolves=20, dim=5, max_iter=100)
    print("\n⚡ Optimization complete!")
    print(f"Best position found: {best_position}")
    print(f"Minimum value of function: {best_score:.6f}")

```

### **Output:**

```
Iteration 59/100 - Best fitness: 0.000000
Iteration 60/100 - Best fitness: 0.000000
Iteration 61/100 - Best fitness: 0.000000
Iteration 62/100 - Best fitness: 0.000000
Iteration 63/100 - Best fitness: 0.000000
Iteration 64/100 - Best fitness: 0.000000
Iteration 65/100 - Best fitness: 0.000000
Iteration 66/100 - Best fitness: 0.000000
Iteration 67/100 - Best fitness: 0.000000
Iteration 68/100 - Best fitness: 0.000000
Iteration 69/100 - Best fitness: 0.000000
Iteration 70/100 - Best fitness: 0.000000
Iteration 71/100 - Best fitness: 0.000000
Iteration 72/100 - Best fitness: 0.000000
Iteration 73/100 - Best fitness: 0.000000
Iteration 74/100 - Best fitness: 0.000000
Iteration 75/100 - Best fitness: 0.000000
Iteration 76/100 - Best fitness: 0.000000
Iteration 77/100 - Best fitness: 0.000000
Iteration 78/100 - Best fitness: 0.000000
Iteration 79/100 - Best fitness: 0.000000
Iteration 80/100 - Best fitness: 0.000000
Iteration 81/100 - Best fitness: 0.000000
Iteration 82/100 - Best fitness: 0.000000
Iteration 83/100 - Best fitness: 0.000000
Iteration 84/100 - Best fitness: 0.000000
Iteration 85/100 - Best fitness: 0.000000
Iteration 86/100 - Best fitness: 0.000000
Iteration 87/100 - Best fitness: 0.000000
Iteration 88/100 - Best fitness: 0.000000
Iteration 89/100 - Best fitness: 0.000000
Iteration 90/100 - Best fitness: 0.000000
Iteration 91/100 - Best fitness: 0.000000
Iteration 92/100 - Best fitness: 0.000000
Iteration 93/100 - Best fitness: 0.000000
Iteration 94/100 - Best fitness: 0.000000
Iteration 95/100 - Best fitness: 0.000000
Iteration 96/100 - Best fitness: 0.000000
Iteration 97/100 - Best fitness: 0.000000
Iteration 98/100 - Best fitness: 0.000000
Iteration 99/100 - Best fitness: 0.000000
Iteration 100/100 - Best fitness: 0.000000
```

Optimization complete!

Best position found: [-1.36314311659068e-11, -1.4462989227279875e-11, -1.381407197291686e-11, -1.2215446596611536e-11, 1.1986042041444537e-11]

Minimum value of function: 0.000000

## Program 7

### **Problem Statement:**

### **Parallel Cellular Algorithm for Function Optimization:**

Use a Parallel Cellular Algorithm to optimize a function efficiently by evolving solutions in a distributed and parallel manner.

### **Algorithm:**

Date : 29/10/15  
Page No.:

Date : \_\_\_\_\_  
Page No.:

Week 7

parallel cellular Algorithm

I/P:  $f(x)$  → objective function to optimize  
grid\_size → size of grid  
num\_cells → cell number  
max\_iterations → max no of iterations  
neighbourhood → structure  
lb, ub

O/P: best solution → best solution found

1) Initialise : Create 2D grid  
For each cell  $i$  in grid:  
Assign random  $x_i$  within  $[lb, ub]$

2) evaluate fitness  
For each cell  $i$   
 $\text{fitness}[i] = f(x_i)$

3) Identify best solution  
 $\text{best\_sol}^*$  = cell with min fitness

4) For  $ite = 1$  to  $\text{max\_iter}$  do  
For each cell  $i$  in grid (in parallel):  
neighbours = cells in neighbourhood of  $i$   
best\_neighbour = neighbour with best fitness  
 $x_{i\_new} = (x_i + \text{best\_neighbour} \cdot \text{value}) / 2$

*Name: [redacted]  
Date: 29/10/15*

### **Code:**

```
import numpy as np
```

```
def parallel_cellular_algorithm(f, grid_size=(10, 10), max_iterations=100, lb=-10, ub=10):  
    rows, cols = grid_size
```

```

num_cells = rows * cols

# 1. Initialize grid with random values
X = np.random.uniform(lb, ub, size=(rows, cols))

# Evaluate fitness
fitness = f(X)

# Track best global solution
best_value = np.min(fitness)
best_pos = np.unravel_index(np.argmin(fitness), fitness.shape)
best_solution = X[best_pos]

# Neighborhood offsets for 3x3 neighborhood
neighbors_offset = [(-1,-1), (-1,0), (-1,1),
                     (0,-1), (0,0), (0,1),
                     (1,-1), (1,0), (1,1)]

for iteration in range(max_iterations):
    X_new = np.copy(X)

    # 4. Update each cell in parallel logic
    for i in range(rows):
        for j in range(cols):
            # Collect neighbors
            neighbor_values = []
            for dx, dy in neighbors_offset:
                ni, nj = i + dx, j + dy
                if 0 <= ni < rows and 0 <= nj < cols:
                    neighbor_values.append((X[ni, nj], fitness[ni, nj]))

            # Get best neighbor (minimum fitness)
            best_neighbor_value, _ = min(neighbor_values, key=lambda x: x[1])

            # Update rule: average
            X_new[i, j] = (X[i, j] + best_neighbor_value) / 2

    # Boundary enforcement
    X_new = np.clip(X_new, lb, ub)

    # Replace old values
    X = X_new

    # Recalculate fitness
    fitness = f(X)

    # Update global best
    if fitness < best_value:
        best_value = fitness
        best_pos = np.unravel_index(np.argmin(fitness), fitness.shape)
        best_solution = X[best_pos]

```

```

current_best = np.min(fitness)
if current_best < best_value:
    best_value = current_best
    best_pos = np.unravel_index(np.argmin(fitness), fitness.shape)
    best_solution = X[best_pos]

return best_solution, best_value

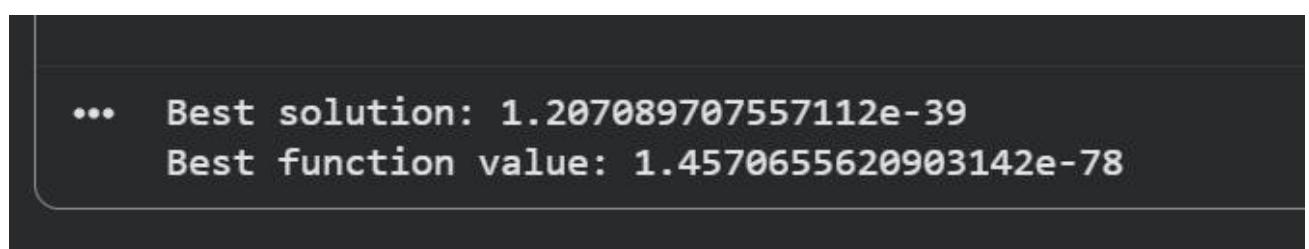
# -----
# Example: Optimize Sphere Function f(x) = x^2
# -----


def sphere_function(x):
    return x**2 # works element-wise

best_x, best_val = parallel_cellular_algorithm(
    f=sphere_function,
    grid_size=(10, 10),
    max_iterations=100,
    lb=-5,
    ub=5
)
print("Best solution:", best_x)
print("Best function value:", best_val)

```

*Output:*



```

...
... Best solution: 1.207089707557112e-39
... Best function value: 1.4570655620903142e-78

```