VISVESVARAYA TECHNOLOGICAL UNIVERSITY

"JnanaSangama", Belgaum -590014, Karnataka.

LAB RECORD

Bio Inspired Systems (23CS5BSBIS)

Submitted by

Umesha H N(1BM24CS428)

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 **Umesha H N(1BM24CS428),** who is a 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.

Dr. RAGHAVENDRA C K Associate Professor Department of CSE, BMSCE	Dr. Kavitha Sooda Professor & HOD Department of CSE, BMSCE

Index

Github Link: https://github.com/Umeshahnn/1BM24CS428_BIS

Program 1 : Genetic Algorithm for Optimization Problems:

Algorithm:

(You should provide the screen shot of your observation book of Algorithm/Logic/Solving of respective problem)

Code:

```
import random
import math

def fitness_function(x):

    return x * math.sin(10 * math.pi * x) + 1

POPULATION_SIZE = 20
MUTATION_RATE = 0.1
CROSSOVER_RATE = 0.8
GENERATIONS = 100
X_BOUND = (-1, 2)  # Search space for x

def create_individual():
    return random.uniform(*X_BOUND)

def create_population(size):
    return [create_individual() for _ in range(size)]

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

def select(population, fitnesses):
    total_fitness = sum(fitnesses)
    probs = [f / total_fitness for f in fitnesses]
    return random.choices(population, weights=probs, k=2)

def crossover(parent1, parent2):
    if random.random() < CROSSOVER_RATE:
        alpha = random.random()
        child1 = alpha * parent1 + (1 - alpha) * parent2
        child2 = alpha * parent2 + (1 - alpha) * parent1
        return child1, child2
    else:
        return parent1, parent2


def mutate(individual):
    if random.random() < MUTATION_RATE:
        mutation = random.uniform(-0.1, 0.1)
```

```
        individual += mutation
        individual = max(min(individual, X_BOUND[1]), X_BOUND[0])  # Keep in bounds
    return individual

def genetic_algorithm():
    population = create_population(POPULATION_SIZE)
    best_individual = None
    best_fitness = float('-inf')

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

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

        new_population = []
        while len(new_population) < POPULATION_SIZE:
            parent1, parent2 = select(population, fitnesses)
            child1, child2 = crossover(parent1, parent2)
            child1 = mutate(child1)
            child2 = mutate(child2)
            new_population.extend([child1, child2])

        population = new_population[:POPULATION_SIZE]

        print(f"Generation {generation+1}: Best Fitness = {best_fitness:.5f}, Best X =
{best_individual:.5f}")

    print("\nOptimization complete!")
    print(f"Best solution: x = {best_individual:.5f}, f(x) = {best_fitness:.5f}")

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

Output:

```
= RESTART: C:/Users/student/AppData/Local/Progr
Best Fitness = 16
Best Fitness = 17
Best Fitness = 17
Best Fitness = 18
Best Fitness = 19
Best Fitness = 18
Best Fitness = 18
Best Fitness = 18
Best Fitness = 19
Best Fitness = 20
Best Fitness = 20
Best Fitness = 19
Best Fitness = 20
Best Fitness = 20
Best Fitness = 20
Best Fitness = 20
Best Fitness = 20
Best Fitness = 20
Best Fitness = 20
Best Fitness = 20
Best Fitness = 20
Best Fitness = 20
Best Fitness = 19
Best Fitness = 20
Best Fitness = 20
Best Fitness = 20
Best Fitness = 19
Best Fitness = 20
Best Fitness = 20
Best Fitness = 20
Best Fitness = 20
Best Fitness = 20
Best Fitness = 20
Best Fitness = 20
Best Fitness = 20
Best Fitness = 20
Best Fitness = 20
Best Fitness = 20
Best Fitness = 20
Best Fitness = 20
Best Fitness = 19
Best Fitness = 19
Best Fitness = 19
Best Fitness = 19
Best Fitness = 20
Best Fitness = 20
Best Fitness = 20
Best Fitness = 20
Best Fitness = 20
Best Fitness = 20

Best solution found:
Fitness (X value): 20
```

~~Algorithm in BIS explanation~~ 29/8/2025

1. Genetic Algorithm

As an example we take $f(n)=n^2$

5 main phases:
- → Initialization
- → Fitness Assignment
- → Selection
- → Cross over
- → Terminator

steps

1) Selecting encoding technique 0 to 31
A population of potential solutions often represented as chromosomes are generated randomly or with a specific initialization

2) Select initial population - "4"

String no	Initial population	x value	fitness $f(n)=n^2$	% Prob	Expected count $f(n)/Avg(f(x))$
1	01100	12	144	0.1247	0.49
2	11001	25	625	0.5411	2.164
3	00101	5	25	0.0216	0.086
4	10011	19	361	0.3125	1.25

Sum 1155

Avg 288.75

MAX 625 → 729 → 841

3) Select Matching pool

String no	Matching pool	Crossover point	Offstring after crossover	x value	fitness $f(n)=n^2$
1	01100	4	01101	13	169
2	11001		11000	24	576
3	11001	2	11011	27	729
4	10011		10001	17	289

4) crossover: Random 472 0 → 1
MAX value -729 1 → 0

5) Mutation

String No	offString after crossover	Mutation chromosome for flipping	offString after Mutation	x value	fitness $f(x)=x^2$
1	0 1 1 0 1	1 0 0 0 0	1 1 1 0 1	29	841
2	1 1 0 0 0	0 0 0 0 0	1 1 0 0 0	24	576
3	1 1 0 1 1	0 0 0 0 0	1 1 0 1 1	27	729
4	1 0 0 0 1	0 0 1 0 1	1 0 1 0 0	20	400

BS code

```
import random
population_size = 100

Genes = ''' abcdefghijklmnopqrstuvwxyz ABCDEFGH
            IJKLMNOPQRSTUVWXYZ 1234567890,.
            ;:-!'#%&/() = ?@${[]}"

target = "BTSLAB"

class Indivial (object):
        def __init__(self, chromosome):
            self.chromosome = chromosome
            #self. chromosome = chromosome
            self.fitness = self.cal_fitness()

    @classmethod
    def mutated_genes(self):
        global genes
        gene = random.choice(Genes)
        return gene
    @classmethod
    def create_gnome(self):
        global target
```

02: **Particle Swarm Optimization for Function Optimization:**

Algorithm :

Code:

```
import random
import math
def fitness_function(x):
    return x * math.sin(10 * math.pi * x) + 1

# 2. Initialize parameters
NUM_PARTICLES = 30
MAX_ITER = 100
W = 0.7
C1 = 1.5
C2 = 1.5
X_BOUND = (-1, 2)
V_MAX = 0.1

class Particle:
    def __init__(self):
        self.position = random.uniform(*X_BOUND)
        self.velocity = random.uniform(-V_MAX, V_MAX)
        self.best_position = self.position
        self.best_fitness = fitness_function(self.position)

    def update_velocity(self, global_best_position):
        r1 = random.random()
        r2 = random.random()
        cognitive = C1 * r1 * (self.best_position - self.position)
        social = C2 * r2 * (global_best_position - self.position)
        self.velocity = W * self.velocity + cognitive + social
        # Clamp velocity within limits
        self.velocity = max(min(self.velocity, V_MAX), -V_MAX)

    def update_position(self):
        self.position += self.velocity
        # Keep particle within bounds
        self.position = max(min(self.position, X_BOUND[1]), X_BOUND[0])

def particle_swarm_optimization():

    swarm = [Particle() for _ in range(NUM_PARTICLES)]

    global_best_particle = max(swarm, key=lambda p: p.best_fitness)
    global_best_position = global_best_particle.position
    global_best_fitness = global_best_particle.best_fitness

    for iteration in range(MAX_ITER):
        for particle in swarm:
```

```
            current_fitness = fitness_function(particle.position)

            if current_fitness > particle.best_fitness:
                particle.best_fitness = current_fitness
                particle.best_position = particle.position

            # Update global best
            if current_fitness > global_best_fitness:
                global_best_fitness = current_fitness
                global_best_position = particle.position

        for particle in swarm:
            particle.update_velocity(global_best_position)
            particle.update_position()


        print(f"Iteration {iteration+1:03d} | Best Fitness = {global_best_fitness:.5f} | Best X = {global_best_position:.5f}")

    print("\nOptimization complete!")
    print(f"Best solution: x = {global_best_position:.5f}, f(x) = {global_best_fitness:.5f}")

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

Output:

OUTPUT:

```
Iteration 1/20 - Best Fitness: 0.081095
Iteration 2/20 - Best Fitness: 0.081095
Iteration 3/20 - Best Fitness: 0.006725
Iteration 4/20 - Best Fitness: 0.003298
Iteration 5/20 - Best Fitness: 0.003298
Iteration 6/20 - Best Fitness: 0.003298
Iteration 7/20 - Best Fitness: 0.002956
Iteration 8/20 - Best Fitness: 0.002956
Iteration 9/20 - Best Fitness: 0.002956
Iteration 10/20 - Best Fitness: 0.002956
Iteration 11/20 - Best Fitness: 0.002956
Iteration 12/20 - Best Fitness: 0.001661
Iteration 13/20 - Best Fitness: 0.001066
Iteration 14/20 - Best Fitness: 0.001066
Iteration 15/20 - Best Fitness: 0.001066
Iteration 16/20 - Best Fitness: 0.000587
Iteration 17/20 - Best Fitness: 0.000587
Iteration 18/20 - Best Fitness: 0.000165
Iteration 19/20 - Best Fitness: 0.000095
Iteration 20/20 - Best Fitness: 0.000018

Best solution found:
Position: [ 0.00321441 -0.00268418]
Fitness: 1.7537291281392393e-05
```

Pseudocode.

9/09/25

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

Example: $f(x, y) = x^2 + y^2$

~~Fne~~ initial = 0.3

value of cognitive + social constant

$c_1 = 2 + c_2 = 2$

initial sol^n all set to 1000

Iteration 1 initial ~~a~~ P_1 Fitness value $= 1^2 + 1^2 = 2$.

Particle No	initial x	Pos y	velocity x	velocity y	Best x	Best y	Fitness value
P_1	1	1	0	0	–	–	2
P_2	-1	1	0	0	–	–	2
P_3	0.5	-0.5	0	0	–	–	0.5
P_4	1	-1	0	0	–	–	2
P_5	0.25	0.25	0	0	–	–	0.125

Iteration' 2

Pno	Initial x	Initial y	velocity x	velocity y	Best sol	Best pos x	Best pos y	Fitness
P1	1	1	-0.75	-0.75	2	1	-1	2
P2	-1	1	1.25	-0.75	2	-1	1	2
P3	0.5	-0.5	-0.25	0.75	0.5	0.5	0.5	0.5
P4	1	-1	-0.75	1.25	2	1	-1	2
P5	0.25	0.25	0	0	0.125	0.25	0.8	0.125

Iteration 3

Pno	Initial x	Initial y	velocity x	velocity y	Best sol	Best pos x	Best pos y	Fitness
P1	0.25	0.25	-0.375	-0.375	2	-1	1	0.125
P2	0.25	0.75	-0.6210	-0.375	2	-1	1	0.125
P3	0.25	0.25	-0.125	-0.375	0.5	0.5	0.5	0.125
P4	0.25	0.25	-0.375	-0.615	2	1	-1	0.125
P5	0.25	0.25	0	0	0.125	0.25	0.75	0.125

output

Best Position : 2.5 , Best = 26.2500

Program 03: **Ant Colony Optimization for the Traveling Salesman Problem:**

Algorithm:

Code:

```
import numpy as np

def distance(city1, city2):
    return np.linalg.norm(city1 - city2)

def route_length(route, cities):
    dist = 0.0
    for i in range(len(route) - 1):
        dist += distance(cities[route[i]], cities[route[i+1]])
    dist += distance(cities[route[-1]], cities[route[0]])
    return dist

def select_next_city(current_city, unvisited, pheromone, heuristic, alpha, beta):
    pheromone_vals = pheromone[current_city, unvisited] ** alpha
    heuristic_vals = heuristic[current_city, unvisited] ** beta
    probs = pheromone_vals * heuristic_vals
    probs /= probs.sum()
    return np.random.choice(unvisited, p=probs)


def aco_tsp(cities, num_ants=20, num_iterations=20, alpha=1.0, beta=5.0, rho=0.5,
initial_pheromone=1.0):
    num_cities = len(cities)


    pheromone = np.full((num_cities, num_cities), initial_pheromone)


    heuristic = np.zeros((num_cities, num_cities))
    for i in range(num_cities):
        for j in range(num_cities):
            if i != j:
                heuristic[i, j] = 1.0 / (distance(cities[i], cities[j]) + 1e-10)

    best_route = None
    best_length = float('inf')

    for iteration in range(num_iterations):
        all_routes = []
        all_lengths = []

        for ant in range(num_ants):
            route = []
            unvisited = list(range(num_cities))


            current_city = np.random.choice(unvisited)
            route.append(current_city)
```

```
            unvisited.remove(current_city)


            while unvisited:
                next_city = select_next_city(current_city, unvisited, pheromone,
heuristic, alpha, beta)
                route.append(next_city)
                unvisited.remove(next_city)
                current_city = next_city

            length = route_length(route, cities)
            all_routes.append(route)
            all_lengths.append(length)


            if length < best_length:
                best_length = length
                best_route = route

        pheromone *= (1 - rho)


        for route, length in zip(all_routes, all_lengths):
            for i in range(num_cities - 1):
                pheromone[route[i], route[i+1]] += 1.0 / length
                pheromone[route[i+1], route[i]] += 1.0 / length

            pheromone[route[-1], route[0]] += 1.0 / length
            pheromone[route[0], route[-1]] += 1.0 / length

        print(f"Iteration {iteration+1}/{num_iterations} - Best Length:
{best_length:.4f}")

    return best_route, best_length


if __name__ == "__main__":

    cities = np.array([
        [0, 0],
        [1, 5],
        [5, 2],
        [6, 6],
        [8, 3],
        [7, 9],
        [2, 8],
        [3, 3]
    ])

    best_route, best_length = aco_tsp(cities)

    print("\nBest route found:")
    print(best_route)
    print("Route length:", best_length)
```

Output:

```
Iteration 1/20 - Best Length: 31.6195
Iteration 2/20 - Best Length: 29.7691
Iteration 3/20 - Best Length: 29.7691
Iteration 4/20 - Best Length: 29.7691
Iteration 5/20 - Best Length: 29.7691
Iteration 6/20 - Best Length: 29.7691
Iteration 7/20 - Best Length: 29.7691
Iteration 8/20 - Best Length: 29.7691
Iteration 9/20 - Best Length: 29.7691
Iteration 10/20 - Best Length: 29.7691
Iteration 11/20 - Best Length: 29.7691
Iteration 12/20 - Best Length: 29.7691
Iteration 13/20 - Best Length: 29.7691
Iteration 14/20 - Best Length: 29.7691
Iteration 15/20 - Best Length: 29.7691
Iteration 16/20 - Best Length: 29.7691
Iteration 17/20 - Best Length: 29.7691
Iteration 18/20 - Best Length: 29.7691
Iteration 19/20 - Best Length: 29.7691
Iteration 20/20 - Best Length: 29.7691

Best route found:
[np.int64(3), np.int64(5), np.int64(6), np.int64(1), np.int64(0), np.int64(7), np.int64(2), np.int64(4)]
Route length: 29.76913194777377
```

Lab - 4 ~~Week 4~~ 10/10/25

Ant Colony Optimization (ACO) for the Travelling Salesman Problem

Pseudo code:

① Initialize Pheromone levels τ on all edges

② Set parameters: ants, alpha (pheromone influ), beta, evaporation_rate, iterations

③ for iter = 1 to iterations do
    for each ant in number_of_ants do
        Initialize starting city randomly
        while tour not complete do
            Choose next city based on probability
            probability = (pheromone() * (heuristic ^ beta) / sum_over_all_possible_cities
            Move to chosen city and add it to the tour
        end while
        calculate total length of the tour
    end for

    update pheromone on all edges:
        Evaporate pheromone. pheromone = * (1 - evaporation_rate)
    • for each ant:
            Deposit pheromone proportional to quality on edges visited

    • optionally, reinforce pheromone on the globally best tour found so far

end for.

Return the best tour found

O/P 14/10/24

Best Solution : [3, 4, 2, 0, 1]

Best length : 14

Program 04: **Cuckoo Search (CS):**

Algorithm;

Code:
```
import numpy as np
import math


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

def levy_flight(Lambda=1.5):
    sigma_u = (math.gamma(1 + Lambda) * math.sin(math.pi * Lambda / 2) /
               (math.gamma((1 + Lambda) / 2) * Lambda * 2 ** ((Lambda - 1) / 2)))
** (1 / Lambda)
    sigma_v = 1
    u = np.random.normal(0, sigma_u)
    v = np.random.normal(0, sigma_v)
    step = u / (abs(v) ** (1 / Lambda))
    return step

def cuckoo_search_nest_location(n=10, iterations=200, pa=0.25, lower_bound=0,
upper_bound=10):
    nests = np.random.uniform(lower_bound, upper_bound, n)
    fitness = np.array([food_availability(x) for x in nests])

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

    Lambda = 1.5

    for t in range(iterations):
        for i in range(n):
            step_size = 0.1 * levy_flight(Lambda)
            new_nest = nests[i] + step_size * (nests[i] - best_nest)
            new_nest = np.clip(new_nest, lower_bound, upper_bound)

            new_fitness = food_availability(new_nest)
            if new_fitness > fitness[i]:
                nests[i] = new_nest
                fitness[i] = new_fitness

                if new_fitness > best_fitness:
                    best_fitness = new_fitness
                    best_nest = new_nest

        num_abandon = int(pa * n)
        worst_indices = np.argsort(fitness)[:num_abandon]
        nests[worst_indices] = np.random.uniform(lower_bound, upper_bound,
num_abandon)
        fitness[worst_indices] = [food_availability(x) for x in
nests[worst_indices]]

        current_best_idx = np.argmax(fitness)
```

```python
        if fitness[current_best_idx] > best_fitness:
            best_fitness = fitness[current_best_idx]
            best_nest = nests[current_best_idx]

        if (t+1) % 20 == 0 or t == 0:
            print(f"Iteration {t+1}: Best nest location = {best_nest:.4f}, Max food = {best_fitness:.6f}")

    return best_nest, best_fitness

if __name__ == "__main__":
    best_location, max_food = cuckoo_search_nest_location()
    print("\nFinal Results:")
    print(f"Best nest location: {best_location:.4f}")
    print(f"Maximum food collected: {max_food:.6f}")
```

Output:

```
Iteration 1: Best nest location = 5.3102, Max food = 19.903783
Iteration 20: Best nest location = 4.9646, Max food = 19.998744
Iteration 40: Best nest location = 4.9646, Max food = 19.998744
Iteration 60: Best nest location = 5.0038, Max food = 19.999985
Iteration 80: Best nest location = 4.9992, Max food = 19.999999
Iteration 100: Best nest location = 5.0000, Max food = 20.000000
Iteration 120: Best nest location = 5.0000, Max food = 20.000000
Iteration 140: Best nest location = 5.0000, Max food = 20.000000
Iteration 160: Best nest location = 5.0000, Max food = 20.000000
Iteration 180: Best nest location = 5.0000, Max food = 20.000000
Iteration 200: Best nest location = 5.0000, Max food = 20.000000

Final Results:
Best nest location: 5.0000
Maximum food collected: 20.000000
```

Lab-5

17/10/25

Cuckoo search Algorithm

Psuedocode

Start Cuckoo Search Algorithm

① Set parameters

- n = numbe of seen (solutions)

- pa = chance of discovering a bad egg (e.g, 0.25)

- MAxGen = number of iterations (generation)

② create n random solutions (nests)

③ Repeat for MaxGen times:

a. for each cuckoo (solution):

i. make a new solution using a random jump

ii) If the new solution is better than the old one:

keep the new one (replace the old)

b. for each nest:

i. with a small chance (pa), forget the bad nests

ii) replace them with new random solutions

c. remember the best solution found so far

④ After All generations, return the best Solution

End Algorithm

output

Gen 1: Best x = -0.43366, f(x) = 11.78999

Gen 5: Best x = 5.50779, f(x) = 6.28899

Gen 10: Best x = 6.63823, f(x) = 13.23670

Gen 15: Best x = 9.26885, f(x) = 39.29847.

Gen 20: Best x = 6.49781, f(x) = 12.23468.

Gen 25: Best x = -1.44701, f(x) = 19.77586

Gen 30: Best x = -0.21684, f(x) = 10.34808.

Gen 35: Best x = 9.11414, f(x) = 37.3872

Gen 40: Best x = 0.76955, f(x) = 4.97492

Gen 45: Best x = -0.5.08760, f(x) = 65.40931

Gen 50: Best x = -3.3412, f(x) = 40.12114

Optimization finished!

Best solution found: x = -3.33412, f(x) = 40.12114

Program 05: Grey Wolf Optimizer (GWO):

Algorithm:

Code:
```
import numpy as np

def nectar_availability(position):
    x, y = position
    term1 = 20 * np.exp(-((x - 3) ** 2 + (y - 3) ** 2) / 4)
    term2 = 15 * np.exp(-((x - 7) ** 2 + (y - 7) ** 2) / 3)
    term3 = 10 * np.exp(-((x - 5) ** 2 + (y - 8) ** 2) / 2)
    return term1 + term2 + term3

def gwo_beehive(n_wolves=30, iterations=20, lower_bound=0, upper_bound=10):
    dim = 2

    wolves = np.random.uniform(lower_bound, upper_bound, (n_wolves, dim))

    alpha_pos = np.zeros(dim)
    alpha_score = -np.inf
    beta_pos = np.zeros(dim)
    beta_score = -np.inf
    delta_pos = np.zeros(dim)
    delta_score = -np.inf

    for t in range(iterations):
        for i in range(n_wolves):
            fitness = nectar_availability(wolves[i])

            if fitness > alpha_score:
                alpha_score = fitness
                alpha_pos = wolves[i].copy()
            elif fitness > beta_score:
                beta_score = fitness
                beta_pos = wolves[i].copy()
            elif fitness > delta_score:
                delta_score = fitness
                delta_pos = wolves[i].copy()

        a = 2 - t * (2 / iterations)

        for i in range(n_wolves):
            for j in range(dim):
                r1 = np.random.rand()
                r2 = np.random.rand()
                A1 = 2 * a * r1 - a
                C1 = 2 * r2
                D_alpha = abs(C1 * alpha_pos[j] - wolves[i][j])
                X1 = alpha_pos[j] - A1 * D_alpha

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

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

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

            wolves[i] = np.clip(wolves[i], lower_bound, upper_bound)

        print(f"Iteration {t+1:2d}: Best nectar = {alpha_score:.6f} at location
x={alpha_pos[0]:.4f}, y={alpha_pos[1]:.4f}")

    return alpha_pos, alpha_score

if __name__ == "__main__":
    best_hive_location, max_nectar = gwo_beehive()
    print("\nFinal optimal beehive location:")
    print(f"x = {best_hive_location[0]:.4f}, y = {best_hive_location[1]:.4f}")
    print(f"Maximum nectar availability: {max_nectar:.6f}")
```

Output:

```
Iteration  1: Best nectar = 14.314430 at location x=5.9729, y=6.9847
Iteration  2: Best nectar = 14.314430 at location x=5.9729, y=6.9847
Iteration  3: Best nectar = 18.833972 at location x=3.4183, y=2.7443
Iteration  4: Best nectar = 18.833972 at location x=3.4183, y=2.7443
Iteration  5: Best nectar = 19.918544 at location x=2.8770, y=3.0355
Iteration  6: Best nectar = 19.918544 at location x=2.8770, y=3.0355
Iteration  7: Best nectar = 19.918544 at location x=2.8770, y=3.0355
Iteration  8: Best nectar = 19.918544 at location x=2.8770, y=3.0355
Iteration  9: Best nectar = 19.926097 at location x=3.1064, y=2.9403
Iteration 10: Best nectar = 19.976975 at location x=3.0522, y=3.0444
Iteration 11: Best nectar = 19.976975 at location x=3.0522, y=3.0444
Iteration 12: Best nectar = 19.976975 at location x=3.0522, y=3.0444
Iteration 13: Best nectar = 19.976975 at location x=3.0522, y=3.0444
Iteration 14: Best nectar = 19.976975 at location x=3.0522, y=3.0444
Iteration 15: Best nectar = 19.976975 at location x=3.0522, y=3.0444
Iteration 16: Best nectar = 19.976975 at location x=3.0522, y=3.0444
Iteration 17: Best nectar = 19.976975 at location x=3.0522, y=3.0444
Iteration 18: Best nectar = 19.976975 at location x=3.0522, y=3.0444
Iteration 19: Best nectar = 19.990873 at location x=2.9582, y=3.0120
Iteration 20: Best nectar = 19.999439 at location x=3.0134, y=2.9978

Final optimal beehive location:
x = 3.0134, y = 2.9978
Maximum nectar availability: 19.999439
```

17/10/20

Grey wolf optimization

~~Pseudocode~~

Applications

* Engineering design and optimization
* machine learning hyper parameter tunning

Pseudocode

- Define nectar function (x, y) as sum of Gaussian clusters.
- initialize wolf population with random positions
- Evaluate fitness of wolves (nectar availability)
- Identify alpha, beta, delta wolves by best fitness

for each iteration:
    update parameter 'a' decreasing from 2 to 10
        for each wolf:
            update position influenced by alpha, beta, delta wolves
            clamp positions within field boundaries

* Evaluate fitness and update alpha, beta, delta wolves
* Return alpha wolf position and fitness as best hive location and nectar amount

Output

Optimal beehive location : x = 3.0015 , y = 2.9970

Maximum nectar availability : 20.00295

RCD 17/10

Program 6: Parallel Cellular Algorithms and Programs:

Algorithm:

Code:

```
import numpy as np

def rosenbrock(x):
    return (1 - x[0])**2 + 100 * (x[1] - x[0]**2)**2

grid_size = (10, 10)
num_dimensions = 2
num_iterations = 200
beta0 = 1.0       # Base attractiveness
gamma = 1.0       # Light absorption coefficient
alpha = 0.2       # Randomness factor
radius = 1
search_bounds = (-2, 2)

cells = np.random.uniform(search_bounds[0], search_bounds[1],
                          size=(grid_size[0], grid_size[1], num_dimensions))

get neighbors (Moore neighborhood)
def get_neighbors(cells, i, j):
    neighbors = []
    for di in range(-radius, radius + 1):
        for dj in range(-radius, radius + 1):
            if di == 0 and dj == 0:
                continue
            ni, nj = (i + di) % grid_size[0], (j + dj) % grid_size[1]
            neighbors.append(cells[ni, nj])
    return np.array(neighbors)

best_solution = None
best_fitness = float('inf')

for t in range(num_iterations):
    new_cells = np.copy(cells)

    for i in range(grid_size[0]):
        for j in range(grid_size[1]):
            firefly = cells[i, j]
            fit_i = rosenbrock(firefly)
            neighbors = get_neighbors(cells, i, j)

            for n in neighbors:
                fit_n = rosenbrock(n)
                if fit_n < fit_i:  # brighter (better)
                    distance = np.linalg.norm(firefly - n)
                    beta = beta0 * np.exp(-gamma * distance**2)
                    step = beta * (n - firefly) + alpha * np.random.uniform(-1, 1,
num_dimensions)
                    firefly = firefly + step
                    fit_i = rosenbrock(firefly)

            new_cells[i, j] = firefly

            if fit_i < best_fitness:
```

```python
                best_fitness = fit_i
                best_solution = firefly

    cells = new_cells

    if t % 20 == 0:
        print(f"Iteration {t}: Best Fitness = {best_fitness:.6f}")

print("\n============================")
print(" Parallel Cellular Firefly Algorithm Results")
print("============================")
print("Best Solution:", best_solution)
print("Best Fitness:", best_fitness)
```

Output:

```
Iteration 0: Best Fitness = 0.120322
Iteration 20: Best Fitness = 0.000162
Iteration 40: Best Fitness = 0.000114
Iteration 60: Best Fitness = 0.000049
Iteration 80: Best Fitness = 0.000049
Iteration 100: Best Fitness = 0.000049
Iteration 120: Best Fitness = 0.000049
Iteration 140: Best Fitness = 0.000049
Iteration 160: Best Fitness = 0.000009
Iteration 180: Best Fitness = 0.000009


============================
🔥 Parallel Cellular Firefly Algorithm Results
============================
Best Solution: [0.99738409 0.99491349]
Best Fitness: 8.760504339778715e-06
```

07/11/25

Parallel Cellular Algorithm

initialization

input: Gridsize (MxN)

input: max Iterators

Initialize Cell [M][N] with initial states

Define neighborhood

Define Transition Rule (cell, neighbors)

for i=1 to maxIterations do

Parallel for each cell (i,j) in cell do

neighbours ← get neighbors (cell, i, j)

newcell[i][j] ← Transition Rule (cell [i][j], neighbours)

end parallel for.

Synchronise()

Parallel for each cell (i,j) in cell do

cell [i][j] ← newcell [i][j]

end parallel fo

Synchronise()

end for

output: Final State of cell [M][N]

O/P

Iteration 0: Best fitness = 0.113938

Iteration 20: Best fitness = 0.000406

Iteration 40: Best fitness = 0.000145.

Iteration 80: Best fitness = 0.00008

Best solution: [1.00253, 1.00519]

Best fitness: 7.95067

Program 7: Optimization via Gene Expression Algorithms:

Algorithm:

Code:

```
import random
import math

def fitness_function(x):
    return x * math.sin(10 * math.pi * x) + 1  # Objective: maximize this

POPULATION_SIZE = 30
NUM_GENES = 10         # Number of genes per chromosome
MUTATION_RATE = 0.1
CROSSOVER_RATE = 0.8
GENERATIONS = 100
X_BOUND = (-1, 2)

def create_individual():
    return [random.uniform(X_BOUND[0], X_BOUND[1]) for _ in range(NUM_GENES)]

def create_population(size):
    return [create_individual() for _ in range(size)]

def express_genes(individual):
    # Expression step: translate gene sequence into a single functional value (x)
    expressed_x = sum(individual) / len(individual)
    return expressed_x

def evaluate_fitness(individual):
    x = express_genes(individual)
    return fitness_function(x)

def tournament_selection(population, k=3):
    selected = random.sample(population, k)
    selected.sort(key=lambda ind: evaluate_fitness(ind), reverse=True)
    return selected[0]

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

def mutate(individual):
    for i in range(NUM_GENES):
        if random.random() < MUTATION_RATE:
            individual[i] += random.uniform(-0.1, 0.1)
            individual[i] = max(min(individual[i], X_BOUND[1]), X_BOUND[0])
    return individual
```

```
def gene_expression_algorithm():
    population = create_population(POPULATION_SIZE)
    best_individual = None
    best_fitness = float('-inf')

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


        for ind in population:
            fit = evaluate_fitness(ind)
            if fit > best_fitness:
                best_fitness = fit
                best_individual = ind

        while len(new_population) < POPULATION_SIZE:
            parent1 = tournament_selection(population)
            parent2 = tournament_selection(population)
            child1, child2 = crossover(parent1, parent2)
            child1 = mutate(child1)
            child2 = mutate(child2)
            new_population.extend([child1, child2])

        population = new_population[:POPULATION_SIZE]

        expressed_x = express_genes(best_individual)
        print(f"Generation {generation+1:03d} | Best Fitness = {best_fitness:.5f} | Best X = {expressed_x:.5f}")

    final_x = express_genes(best_individual)
    print("\nOptimization complete!")
    print(f"Best solution: x = {final_x:.5f}, f(x) = {best_fitness:.5f}")

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

Output:

```
Output
Generation 1: Best Value = 6.363236
Generation 2: Best Value = 6.363236
Generation 3: Best Value = 6.363236
Generation 4: Best Value = 6.363236
Generation 5: Best Value = 4.087452
Generation 6: Best Value = 3.420658
Generation 7: Best Value = 3.420658
Generation 8: Best Value = 3.420658
Generation 9: Best Value = 3.420658
Generation 10: Best Value = 3.420658

Best Solution Found: [0.07026351598497271, -0.36887618913241305, 0
    .16261776749364998, 1.729095856868689, 0.513258486361484]
Best Value: 3.4206578989545884
```

LAB - 02

Gene Expression Algorithm (GEA) : 6

6 Main Phases

-> Initialization
-> Fitness Assignment
-> Selection
-> Cross over
-> Mutation
-> Gene Expression
-> Termination,

Steps :- Fitness$(x) = x^2$

① Select encoding technique 0 to 31
Use Chromosome is fixed length with terminals (variable, constraint & functions)

② Initialize Population,

	Initial Chromosome	Phenotype	value	fitness	Probability
1	+xa	x2	12	44	0.1247
2	+x2	2x	25	625	0.541
3	-x	x	5	25	0.0216
4	-x2	x-2	19	361	0.3125

$\Sigma P(x) = 1155$

Avg = 288.75

Actual count	Expected count
1	0.5
2	9.1
0	0.08
1	1.21

3. selection of matching pool.

	Selected Chromosome	Crossover pool	Offspring	Phenotype	value
				$x(x+1)$	
1	+xx	2	+xx		13
2	+xx	1	4xx	4xx	24
3	+xx	3	+x-	+x-	27
4	$-x\overset{2}{Z}$	1	+xz	+xz	17

fitness
169
576
729
289

4. Crossover: Perform crossover randomly chosen gene position max fitness after crossover = 729

5. Mutation:

	Offspring before mutation	Mutation Applies	Offspring after mutation	Phenotype
1	x+1	→ -	*x-	$x^{+}(x---)$
2	~~+~~xx	None	+xx	zx
3	~~+~~x-	-->	+xx	x-x
4	+xz	None	+xz	x+z

x value	Fitness
29	841
24	576
27	729
20	400

6) Gene expression & Evaluation

Decode each genotype -> phenotype

calculated Fitness

$\Sigma f(x) = 2546$

Avg = 636.5

MAX = 341

7) Iterate until convergence

repeat step 3-6, until fitness improvement is negligible / generation limit ~~has~~ has reached

Pseudocode

Start

-> Define Fitness Evaluation

-> Create population

-> Define parameters

-> select mating pool

-> Mutation after mating

-> gene expression and Evolution

-> Iterate

-> output Best value.

Output

Genes :- [29.53, 29.82, 29.34, 28.57, 15.05, 21.8 23.13, 30.81, 22.51, 26.22]

x = 26.37

f(x) = 695.45