

VISVESVARAYA TECHNOLOGICAL UNIVERSITY

“JnanaSangama”, Belgaum -590014, Karnataka.



LAB RECORD

Bio Inspired Systems (23CS5BSBIS)

Submitted by

SANA SUBODH (1BM23CS296)

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 **SANA SUBODH(1BM23CS296)**, 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:	04-06
2	12/09/2025	Particle Swarm Optimization for Function Optimization:	07-08
3	10/10/2025	Ant Colony Optimization for the Traveling Salesman Problem:	09-10
4	17/10/2025	Cuckoo Search (CS):	11-13
5	17/10/2025	Grey Wolf Optimizer (GWO):	14-17
6	29/08/2025	Parallel Cellular Algorithms and Programs:	18-19
7	07/11/2025	Optimization via Gene Expression Algorithms:	20-22

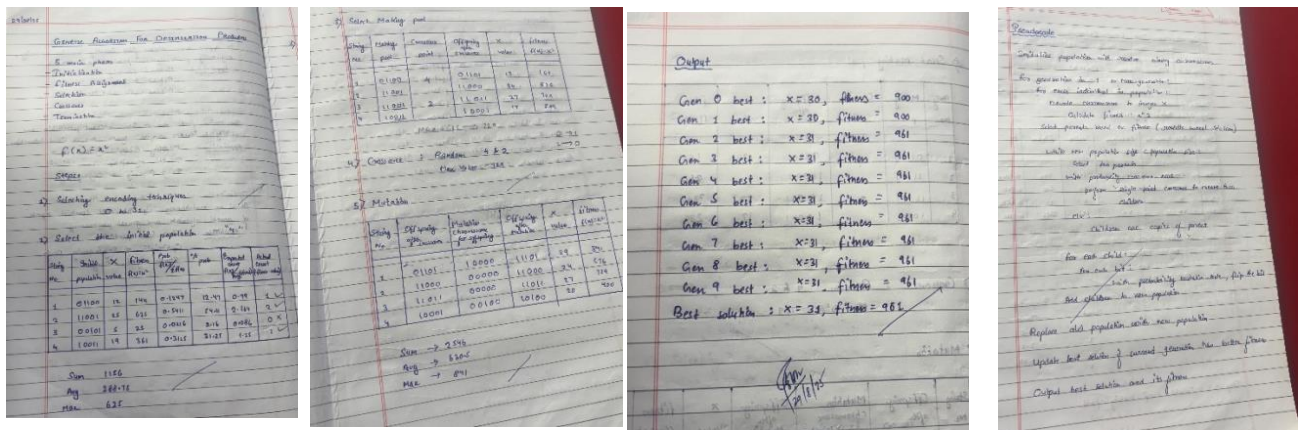
Github Link:

<https://github.com/SANASUBODH/BISLAB>

Program 1

Genetic Algorithm for Optimization Problems:

Algorithm:



Code:

```
import random

# Define the function to maximize
def fitness_function(x):
    return x ** 2

# Convert binary string to integer
def decode(chromosome):
    return int(chromosome, 2)

# Evaluate fitness for the entire population
def evaluate_population(population):
    return [fitness_function(decode(individual)) for individual in population]

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

# Crossover: Single-point crossover
def crossover(parent1, parent2):
```

```

if random.random() < CROSSOVER_RATE:
    point = random.randint(1, CHROMOSOME_LENGTH - 1)
    return (parent1[:point] + parent2[point:], parent2[:point] + parent1[point:])
return parent1, parent2

# Mutation: Flip random bit
def mutate(chromosome):
    new_chromosome = ""
    for bit in chromosome:
        if random.random() < MUTATION_RATE:
            new_chromosome += '0' if bit == '1' else '1'
        else:
            new_chromosome += bit
    return new_chromosome

# Get user input for initial population
def get_initial_population(size, length):
    population = []
    print(f"Enter {size} chromosomes (each of {length} bits, e.g., '10101':)")
    while len(population) < size:
        chrom = input(f"Chromosome {len(population)+1}: ").strip()
        if len(chrom) == length and all(bit in '01' for bit in chrom):
            population.append(chrom)
        else:
            print(f"Invalid input. Please enter a {length}-bit binary string.")
    return population

# Run the Genetic Algorithm
def genetic_algorithm():
    population = get_initial_population(POPULATION_SIZE, CHROMOSOME_LENGTH)
    best_solution = None
    best_fitness = float('-inf')

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

        # Update best solution
        for i, individual in enumerate(population):
            if fitnesses[i] > best_fitness:
                best_fitness = fitnesses[i]
                best_solution = individual

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

    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"Chromosome: {best_solution}")

```

```

print(f"x = {decode(best_solution)}")
print(f"f(x) = {fitness_function(decode(best_solution))}")

# Parameters
POPULATION_SIZE = 4
CHROMOSOME_LENGTH = 5
MUTATION_RATE = 0.01
CROSSOVER_RATE = 0.8
GENERATIONS = 20

# Run it
if __name__ == "__main__":
    genetic_algorithm()

```

OUTPUT:

```

Enter 4 chromosomes (each of 5 bits, e.g., '10101'):
Chromosome 1: 01101
Chromosome 2: 11000
Chromosome 3: 11011
Chromosome 4: 10001
Generation 1: Best Fitness = 729, Best x = 27
Generation 2: Best Fitness = 841, Best x = 29
Generation 3: Best Fitness = 841, Best x = 29
Generation 4: Best Fitness = 841, Best x = 29
Generation 5: Best Fitness = 841, Best x = 29
Generation 6: Best Fitness = 841, Best x = 29
Generation 7: Best Fitness = 841, Best x = 29
Generation 8: Best Fitness = 841, Best x = 29
Generation 9: Best Fitness = 841, Best x = 29
Generation 10: Best Fitness = 841, Best x = 29
Generation 11: Best Fitness = 841, Best x = 29
Generation 12: Best Fitness = 841, Best x = 29
Generation 13: Best Fitness = 841, Best x = 29
Generation 14: Best Fitness = 841, Best x = 29
Generation 15: Best Fitness = 841, Best x = 29
Generation 16: Best Fitness = 841, Best x = 29
Generation 17: Best Fitness = 841, Best x = 29
Generation 18: Best Fitness = 841, Best x = 29
Generation 19: Best Fitness = 841, Best x = 29
Generation 20: Best Fitness = 841, Best x = 29

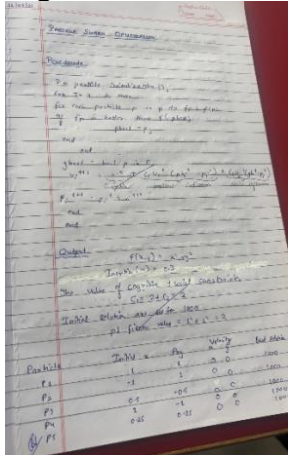
Best solution found:
Chromosome: 11101
x = 29
f(x) = 841

```

Program 2

Particle Swarm Optimization for Function Optimization:

Algorithm:



Code:

```
import random
```

```
def objective(pos):
```

```
    return pos[0]**2 + pos[1]**2
```

```
# PSO parameters
```

```
num_particles = 3
```

```
num_iterations = 2
```

```
w = 0.5 # inertia
```

```
c1 = 1.0 # cognitive
```

```
c2 = 1.0 # social
```

```
r1 = r2 = 0.5 # fixed random factors for simplicity
```

```
# Initial positions and velocities
```

```
positions = [[2, 3], [-1, 4], [0, -2]]
```

```
velocities = [[0, 0] for _ in range(num_particles)]
```

```
pbests = [pos[:] for pos in positions]
```

```
pbest_scores = [objective(p) for p in positions]
```

```
gbest = pbests[pbest_scores.index(min(pbest_scores))][:]
```

```
gbest_score = min(pbest_scores)
```

```
for it in range(num_iterations):
```

```
    for i in range(num_particles):
```

```
        for d in range(2): # for x and y
```

```
            velocities[i][d] = (
```

```
                w * velocities[i][d]
```

```
                + c1 * r1 * (pbests[i][d] - positions[i][d])
```

```

        + c2 * r2 * (gbest[d] - positions[i][d])
    )
    positions[i][d] += velocities[i][d]
    score = objective(positions[i])
    if score < pbest_scores[i]:
        pbests[i] = positions[i][:]
        pbest_scores[i] = score
    gbest = pbests[pbest_scores.index(min(pbest_scores))][:]
    gbest_score = min(pbest_scores)
    print(f"Iteration {it+1}: gbest position = {gbest}, gbest score = {gbest_score}")

```

OUTPUT:

```

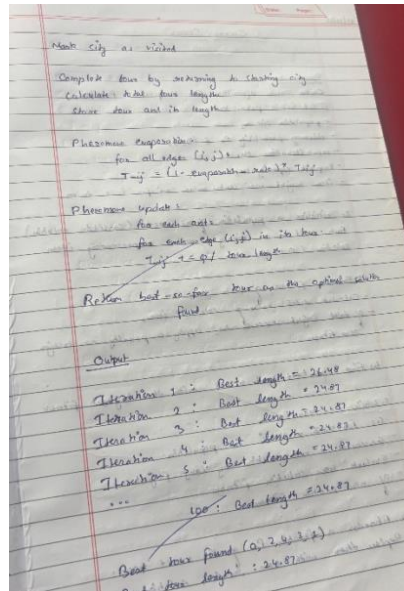
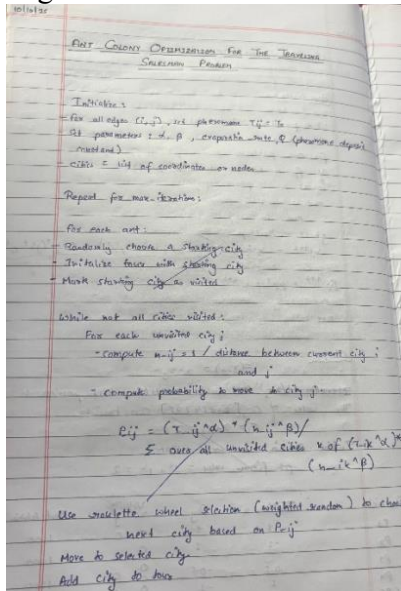
Iteration 1: gbest position = [1.0, 0.5], gbest score = 1.25
Iteration 2: gbest position = [0.5, -0.75], gbest score = 0.8125

```

Program 3

Ant Colony Optimization for the Traveling Salesman Problem:

Algorithm:



Code:

```
import random, math
```

```
n = 5
```

```
ants = 5
```

```
alpha, beta, rho, Q = 1, 5, 0.5, 100
```

```
iters = 20
```

```
dist = [
```

```
    [0, 2, 2, 5, 7],
```

```
    [2, 0, 4, 8, 2],
```

```
    [2, 4, 0, 1, 3],
```

```
    [5, 8, 1, 0, 2],
```

```
    [7, 2, 3, 2, 0]
```

```
]
```

```
pher = [[1]*n for _ in range(n)]
```

```
def prob(i,j,v):
```

```
    return 0 if j in v else (pher[i][j]**alpha)*((1/dist[i][j])**beta)
```

```
def next_city(i,v):
```

```
    p=[prob(i,j,v) for j in range(n)]
```

```
    s=sum(p)
```

```
    if s==0: return random.choice([j for j in range(n) if j not in v])
```

```
    r=random.random(); c=0
```

```

for j in range(n):
    c+=p[j]/s
    if r<=c: return j

def tour_len(t):
    return sum(dist[t[i]][t[i+1]] for i in range(n-1))+dist[t[-1]][t[0]]

best,bestL=None,math.inf

for it in range(iters):
    tours=[]
    for _ in range(ants):
        t=[random.randint(0,n-1)]
        while len(t)<n: t.append(next_city(t[-1],t))
        tours.append(t)
    for i in range(n):
        for j in range(n):
            pher[i][j]*=(1-rho)
    for t in tours:
        L=tour_len(t)
        for i in range(n):
            a,b=t[i],t[(i+1)%n]
            pher[a][b]+=Q/L; pher[b][a]+=Q/L
        if L<bestL: best,bestL=t,L
    print(f'Iteration {it+1:2d}: Best Length = {bestL}')

print("\nFinal Best Tour:", best)
print("Final Best Length:", bestL)

```

OUTPUT:

```

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

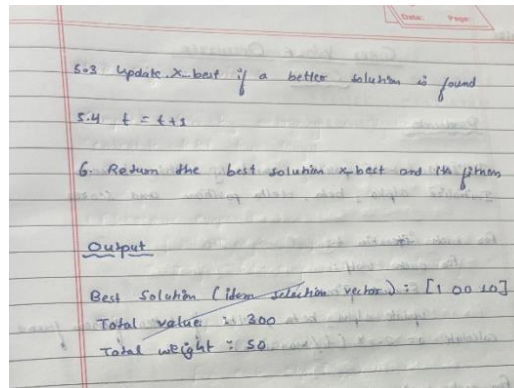
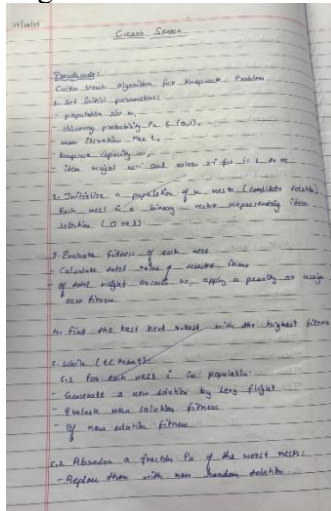
Final Best Tour: [1, 4, 3, 2, 0]
Final Best Length: 9

```

Program 4

Cuckoo Search (CS):

Algorithm:



Code:

```
import random
import math
```

```
# Fitness function (objective function to minimize)
```

```
def fitness(x):
```

```
    return x**2 #  $f(x) = x^2$ , global minimum at  $x = 0$ 
```

```
# Levy flight for generating new solutions
```

```
def levy_flight(beta=1.5):
```

```
    # Generate a random step using Levy flight (simplified version)
```

```
    u = random.gauss(0, 1) # Gaussian distribution for u
```

```
    v = random.gauss(0, 1) # Gaussian distribution for v
```

```
    step = u / abs(v) ** (1 / beta) # Levy flight step
```

```
    return step
```

```
# Cuckoo Search Algorithm
```

```
def cuckoo_search(nests=10, max_iter=100, pa=0.25):
```

```
    # Step 1: Initialize nests (solutions)
```

```
    nests_positions = [random.uniform(-10, 10) for _ in range(nests)]
```

```
    fitness_values = [fitness(x) for x in nests_positions]
```

```
    # Best solution initially
```

```
    best_nest = nests_positions[fitness_values.index(min(fitness_values))]
```

```
    best_fitness = min(fitness_values)
```

```

# Iteration loop
for t in range(max_iter):
    # Generate new solutions via Levy flight
    new_nests = [nest + levy_flight() * (nest - best_nest) for nest in nests_positions]

    # Evaluate fitness of new solutions
    new_fitness = [fitness(x) for x in new_nests]

    # Update nests with better solutions
    for i in range(nests):
        if new_fitness[i] < fitness_values[i]:
            nests_positions[i] = new_nests[i]
            fitness_values[i] = new_fitness[i]

    # Abandon worst nests with probability Pa
    for i in range(nests):
        if random.random() < pa:
            nests_positions[i] = random.uniform(-10, 10) # Generate a new random solution
            fitness_values[i] = fitness(nests_positions[i])

    # Update best solution
    current_best_nest = nests_positions[fitness_values.index(min(fitness_values))]
    current_best_fitness = min(fitness_values)

    if current_best_fitness < best_fitness:
        best_nest = current_best_nest
        best_fitness = current_best_fitness

    print(f'Iteration {t+1}, Best Fitness: {best_fitness}, Best Nest: {best_nest}')

return best_nest, best_fitness

# Run the cuckoo search algorithm
best_solution, best_value = cuckoo_search(nests=10, max_iter=100, pa=0.25)
print(f'Global minimum found at x = {best_solution}, f(x) = {best_value}')

```

[illegible][illegible]

Program 5

Grey Wolf Optimizer (GWO):

Algorithm:

GREY WOLF OPTIMIZER
(GWO)

Procedure

Initialize wolf positions randomly within bounds.
Initialize alpha, beta, delta positions and leaders.

For each iteration:

For each wolf:

Calculate fitness.

Update alpha, beta, delta if better fitness found.

Calculate $a = 2 - \frac{2 \times \text{iteration}}{\text{max_iteration}}$.

For each wolf and each dimension:

Calculate new position using:

$$X_1 = \text{alpha_pos} - A \cdot [r_1 \cdot (\text{alpha_pos} - \text{wolf_pos})]$$

$$X_2 = \text{alpha_pos} - A \cdot [r_2 \cdot (\text{beta_pos} - \text{wolf_pos})]$$

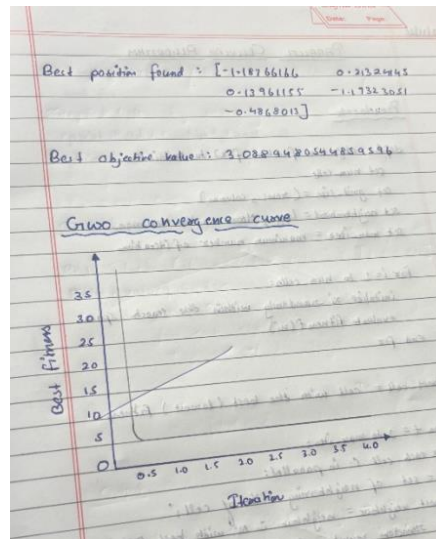
$$X_3 = \text{alpha_pos} - A \cdot [r_3 \cdot (\text{delta_pos} - \text{wolf_pos})]$$

update wolf position = $(X_1 + X_2 + X_3) / 3$

Return alpha_pos as best solution.

Output

Iteration 1000, Best fitness: 34.083101835172
Iteration 1000, Best fitness: 16.1531161711556
Iteration 1000, Best fitness: 11.75598052483987
Iteration 1000, Best fitness: 2.08294485485936
Iteration 1000, Best fitness: 2.08294485485936



Code:

```
import numpy as np
```

```
import matplotlib.pyplot as plt
```

```
def initialize_population(num_wolves, dimensions, search_space_min, search_space_max):
```

```
    """Initializes the grey wolf population within the search space."""
```

```
    population = search_space_min + (search_space_max - search_space_min) *
```

```
    np.random.rand(num_wolves, dimensions)
```

```
    return population
```

```
def calculate_distance(C, Xp, X):
```

```
    """Calculates the distance between a wolf and the prey."""
```

```
    return np.abs(C * Xp - X)
```

```
def update_position(Xp, D, A):
```

```
    """Updates the position of a wolf."""
```

```
    return Xp - A * D
```

```
def grey_wolf_optimizer(objective_function, num_wolves, dimensions, search_space_min,
search_space_max, max_iterations):
```

```
    """Implements the Grey Wolf Optimizer algorithm."""
```

```
    population = initialize_population(num_wolves, dimensions, search_space_min,
search_space_max)
```

```

alpha_position = np.zeros(dimensions)
beta_position = np.zeros(dimensions)
delta_position = np.zeros(dimensions)
alpha_score = float('inf')
beta_score = float('inf')
delta_score = float('inf')

convergence_curve = []

for iteration in range(max_iterations):
    scores = np.array([objective_function(wolf) for wolf in population])

    # Update alpha, beta, and delta
    sorted_indices = np.argsort(scores)
    alpha_score = scores[sorted_indices[0]]
    alpha_position = population[sorted_indices[0]].copy()

    if num_wolves > 1:
        beta_score = scores[sorted_indices[1]]
        beta_position = population[sorted_indices[1]].copy()
    if num_wolves > 2:
        delta_score = scores[sorted_indices[2]]
        delta_position = population[sorted_indices[2]].copy()

    # Update wolf positions
    a = 2 - iteration * (2 / max_iterations) # Decreases linearly from 2 to 0

    for i in range(num_wolves):
        r1 = np.random.rand(dimensions)
        r2 = np.random.rand(dimensions)

        A1 = 2 * a * r1 - a
        C1 = 2 * r2
        D_alpha = calculate_distance(C1, alpha_position, population[i])
        X1 = update_position(alpha_position, D_alpha, A1)

        r1 = np.random.rand(dimensions)
        r2 = np.random.rand(dimensions)
        A2 = 2 * a * r1 - a
        C2 = 2 * r2
        D_beta = calculate_distance(C2, beta_position, population[i])
        X2 = update_position(beta_position, D_beta, A2)

        r1 = np.random.rand(dimensions)
        r2 = np.random.rand(dimensions)
        A3 = 2 * a * r1 - a

```

```

C3 = 2 * r2
D_delta = calculate_distance(C3, delta_position, population[i])
X3 = update_position(delta_position, D_delta, A3)

# Update position based on alpha, beta, and delta
population[i] = (X1 + X2 + X3) / 3

# Handle boundaries
population[i] = np.clip(population[i], search_space_min, search_space_max)

convergence_curve.append(alpha_score)

return alpha_position, alpha_score, convergence_curve

# Example objective function (Sphere function)
def sphere_function(x):
    return np.sum(x**2)

# GWO parameters
num_wolves = 30
dimensions = 5
search_space_min = -100
search_space_max = 100
max_iterations = 100

# Run GWO
best_position, best_score, convergence_curve = grey_wolf_optimizer(
    sphere_function, num_wolves, dimensions, search_space_min, search_space_max, max_iterations
)

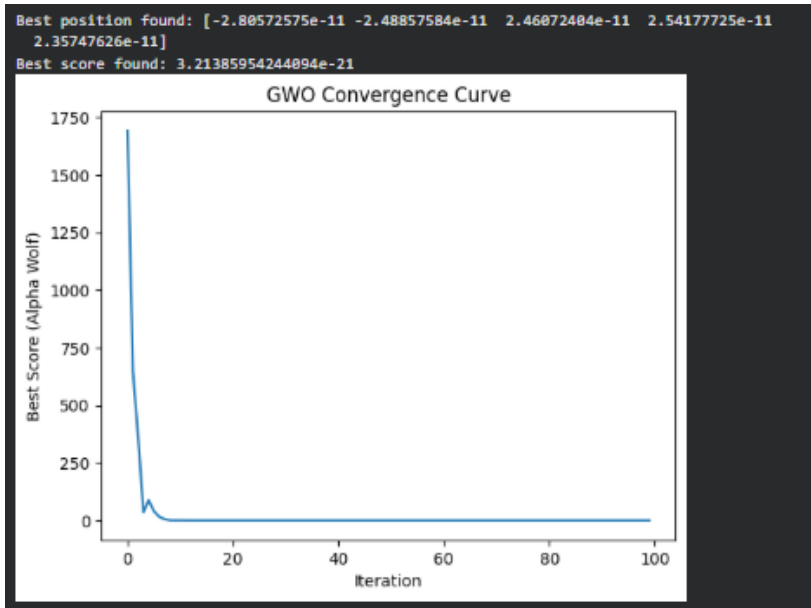
print("Best position found:", best_position)
print("Best score found:", best_score)

# You can also plot the convergence curve to visualize the optimization process

plt.plot(convergence_curve)
plt.xlabel("Iteration")
plt.ylabel("Best Score (Alpha Wolf)")
plt.title("GWO Convergence Curve")
plt.show()

```

OUTPUT:



Program 6

Parallel Cellular Algorithms and Programs:

Algorithm:

Parallel Cellular Algorithm

Pseudocode

```

define objective function  $f(x)$  to minimize
set  $max\_cells$ 
set  $grid\_size = (rows, columns)$ 
set  $neighborhood = \{eq, Ham, 8\}$ 
set  $max\_iter = maximum\_number\_of\_iterations$ 

for  $i = 1$  to  $max\_cells$ :
    initialize  $x_i$  randomly within the search space
    evaluate  $f(x_i)$ 
end for

best_cell = cell with the best (lowest) fitness

for  $t = 1$  to  $max\_iter$ :
    for each cell  $i$  in parallel:
         $n_i$  = set of neighboring cells of cell  $i$ 
        select neighbor  $n_j$  according to  $n_j$ 's best fitness
         $x_i$  = random number in  $[a, b]$ 
         $x_i$  = new =  $x_i + x^* (n_{best\_neighbor} - x_i)$ 
         $f_i$  = new =  $f(x_i)$ 
        if  $f_i$  <  $f(x_i)$ :
             $x_i$  =  $x_i$  new
        end if

        if  $f(x_i) < f(best\_cell)$ :
            best_cell =  $x_i$ 
        end if
    end for
end for
  
```

end for
end for

print "Best Solution", best_cell
print "Best fitness", $f(best_cell)$

Output

Best solution: $[x_1, x_2, \dots, x_n]$
Best fitness: $f(best_cell)$

Code:

```
import numpy as np
```

```
from multiprocessing import Pool
```

```
def objective(x):
```

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

```
def update(cell, neighbor, bounds):
```

```
    new_cell = cell + 0.1 * (neighbor - cell)
```

```
    new_cell = np.clip(new_cell, bounds[0], bounds[1])
```

```
    return new_cell
```

```
def parallel_cellular(dim=3, bounds=(-5,5), grid_shape=(4,4), max_iters=50):
```

```
    np.random.seed(0)
```

```
    grid = np.random.uniform(bounds[0], bounds[1], size=(grid_shape[0], grid_shape[1], dim))
```

```
    pool = Pool()
```

```
    for t in range(1, max_iters + 1):
```

```
        tasks = []
```

```
        for i in range(grid_shape[0]):
```

```
            for j in range(grid_shape[1]):
```

```
                ni, nj = np.random.randint(0, grid_shape[0]), np.random.randint(0, grid_shape[1])
```

```
                tasks.append((grid[i, j], grid[ni, nj], bounds))
```

```
            results = pool.starmap(update, tasks)
```

```
            grid = np.array(results).reshape(grid_shape[0], grid_shape[1], dim)
```

```

fits = np.array([[objective(grid[i,j]) for j in range(grid_shape[1])] for i in range(grid_shape[0])])
best_idx = np.unravel_index(np.argmin(fits), fits.shape)
best_fit = fits[best_idx]
print(f'Iteration {t:3d}/{max_iters} | Best fitness = {best_fit:.6f}')

pool.close(); pool.join()
best_sol = grid[best_idx]
return best_sol, best_fit

if __name__ == "__main__":
    best_sol, best_fit = parallel_cellular(dim=3, bounds=(-5,5), grid_shape=(4,4), max_iters=50)
    print("\nBest solution found:", best_sol)
    print("Best fitness:", best_fit)

```

OUTPUT:

```

Iteration 1/50 | Best fitness = 1.072768
Iteration 2/50 | Best fitness = 0.778101
Iteration 3/50 | Best fitness = 0.949416
Iteration 4/50 | Best fitness = 0.938134
Iteration 5/50 | Best fitness = 1.467251
Iteration 6/50 | Best fitness = 1.557592
Iteration 7/50 | Best fitness = 1.053744
Iteration 8/50 | Best fitness = 1.114765
Iteration 9/50 | Best fitness = 1.356454
Iteration 10/50 | Best fitness = 1.268463
Iteration 11/50 | Best fitness = 1.114063
Iteration 12/50 | Best fitness = 0.785405
Iteration 13/50 | Best fitness = 0.534989
Iteration 14/50 | Best fitness = 0.335928
Iteration 15/50 | Best fitness = 0.310278
Iteration 16/50 | Best fitness = 0.211640
Iteration 17/50 | Best fitness = 0.133326
Iteration 18/50 | Best fitness = 0.133326
Iteration 19/50 | Best fitness = 0.058898
Iteration 20/50 | Best fitness = 0.095787
Iteration 21/50 | Best fitness = 0.093319
Iteration 22/50 | Best fitness = 0.130149
Iteration 23/50 | Best fitness = 0.195106
Iteration 24/50 | Best fitness = 0.204868
Iteration 25/50 | Best fitness = 0.204868
Iteration 26/50 | Best fitness = 0.250055
Iteration 27/50 | Best fitness = 0.296530
Iteration 28/50 | Best fitness = 0.354994
Iteration 29/50 | Best fitness = 0.393302
Iteration 30/50 | Best fitness = 0.436567
Iteration 31/50 | Best fitness = 0.465526
Iteration 32/50 | Best fitness = 0.507777
Iteration 33/50 | Best fitness = 0.546291
Iteration 34/50 | Best fitness = 0.588597
Iteration 35/50 | Best fitness = 0.614305
Iteration 36/50 | Best fitness = 0.646638
Iteration 37/50 | Best fitness = 0.668270
Iteration 38/50 | Best fitness = 0.693019
Iteration 39/50 | Best fitness = 0.707178
Iteration 40/50 | Best fitness = 0.730492
Iteration 41/50 | Best fitness = 0.754197
Iteration 42/50 | Best fitness = 0.767602
Iteration 43/50 | Best fitness = 0.780511
Iteration 44/50 | Best fitness = 0.793749
Iteration 45/50 | Best fitness = 0.813550
Iteration 46/50 | Best fitness = 0.828427
Iteration 47/50 | Best fitness = 0.835176
Iteration 48/50 | Best fitness = 0.843817
Iteration 49/50 | Best fitness = 0.857765
Iteration 50/50 | Best fitness = 0.865320

Best solution found: [0.44644079 0.22715409 0.78384392]
Best fitness: 0.8653196576734627

```



```

    return sum(chromosome) / len(chromosome)

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

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

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

def gene_expression_algorithm():
    population = initialize_population(POPULATION_SIZE)
    best_solution = None
    best_fitness = float("-inf")

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

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

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

        new_population = []
        while len(new_population) < POPULATION_SIZE:

```

```

    parent1 = select(population, fitnesses)
    parent2 = select(population, fitnesses)
    offspring1, offspring2 = crossover(parent1, parent2)
    offspring1 = mutate(offspring1)
    offspring2 = mutate(offspring2)
    new_population.extend([offspring1, offspring2])

    population = new_population[:POPULATION_SIZE]

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

if __name__ == "__main__":
    gene_expression_algorithm()

```

OUTPUT:

```

Generation 1: Best Fitness = 2.8721, Best x = 1.0697
Generation 2: Best Fitness = 2.8721, Best x = 1.0697
Generation 3: Best Fitness = 2.8721, Best x = 1.0697
Generation 4: Best Fitness = 2.8721, Best x = 1.0697
Generation 5: Best Fitness = 2.8721, Best x = 1.0697
Generation 6: Best Fitness = 2.8721, Best x = 1.0697
Generation 7: Best Fitness = 2.8721, Best x = 1.0697
Generation 8: Best Fitness = 2.8721, Best x = 1.0697
Generation 9: Best Fitness = 2.8721, Best x = 1.0697
Generation 10: Best Fitness = 2.8721, Best x = 1.0697
Generation 11: Best Fitness = 2.8721, Best x = 1.0697
Generation 12: Best Fitness = 2.8721, Best x = 1.0697
Generation 13: Best Fitness = 2.8721, Best x = 1.0697
Generation 14: Best Fitness = 2.8721, Best x = 1.0697
Generation 15: Best Fitness = 2.8721, Best x = 1.0697
Generation 16: Best Fitness = 2.8721, Best x = 1.0697
Generation 17: Best Fitness = 3.0504, Best x = 1.0505
Generation 18: Best Fitness = 3.0504, Best x = 1.0505
Generation 19: Best Fitness = 3.0504, Best x = 1.0505
Generation 20: Best Fitness = 3.0504, Best x = 1.0505

Best solution found:
Genes: [1.7474487834851282, 1.448835705315127, 1.288354202186989, 0.5356338186642926, 1.9313294626811004, 0.4938398895126963, 1.1917819583142881, 0.49568705659264634, 1.646857105140671, -0.2751214395471905]
x = 1.0505
f(x) = 3.0504

```