VISVESVARAYA TECHNOLOGICAL UNIVERSITY

“JnanaSangama”, Belgaum -590014, Karnataka.

LAB RECORD

Bio Inspired Systems (23CS5BSBIS)

Submitted by

Subramanya J (1BM23CS343)

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 **Subramanya J (1BM23CS343),** 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

Github Link:
https://github.com/SubramanyaJ/23CS5BSBIS

Program 1

Genetic Algorithm for Optimization Problems.

Algorithm:

29.08.2025 Lab-1 Page No. 3

Genetic Algorithm

Steps:

1. Define the Problem
   Create a mathematical function to optimize

2. Initialize parameters
   Population size, mutation rate, etc

3. Create initial population
   Generate initial population of potential solutions

4. Evaluate Fitness
   Evaluate for each individual

5. Selection
   Based on fitness to reproduce

6. Crossover
   Perform crossover between individuals

7. Mutation
   Repeat evaluation, selection, crossover, mutation

8. Iteration: Repeat for n generation

9. Output the Best Solution

Page No. 4

Pseudocode / Algorithm:

1. Start

2. Initialize popn, initial popn, mut rate, num_gens

3. Define functions
   sel_indiv (df):
      return list of selected individuals

   cross (mating_pool):
      shuffle mating_pool
      for each pair, perform
         crossover, return offspring

   mutate_offs (of, m_rate)
      for each offspring, apply
      mutation with predefined rate

   gen_gen (popul, mut_rate,
         initial_popn):
      select individuals with
      sel_indiv
      call cross, mutate offspring
      return population, new data

4. for gen=1 to num_gens:
   gen_gen (gen)
   find best individual,
   update best_fit

5. Output results

Output

Generation Data

No	Popn	Value	Fit	Prob	%	expect	actual
1	12	01100	144	0.124	12.4	0.49	0
2	25	00011001	625	0.549	54.1	2.16	2
3	5	00101	25	0.021	2.16	0.01	1
4	19	10011	361	0.321	31.25	1.25	1

Mating pool

No	Mating Pool	Value	Crossover	Fitness
1	11001	25	2	625
2	11001	25	2	625
3	10011	19	3	361
4	11001	25	3	625

Mutation data

No	Offspring	after mutation	x	Fitness
1	11001	-	25	625
2	11001	-	25	625
3	11001	-	25	625
4	10011	-	19	361

Fitness of overall best fit: 625

Best solution: 25

Code:

```
import pandas as pd
import random

def sel_indiv(df):
  return [row['Popul'] for _, row in df.iterrows() for _ in
range(int(row['ActualCnt']))]

def cross(mating_pool):
  random.shuffle(mating_pool)
  if len(mating_pool) % 2 != 0: mating_pool.append(random.choice(mating_pool))
  offspring, mating_pool_data, crossover_data = [], [], []
  for i in range(0, len(mating_pool), 2):
    p1, p2 = int(mating_pool[i]), int(mating_pool[i+1])
    bp1, bp2 = bin(p1)[2:].zfill(8), bin(p2)[2:].zfill(8)
    cp = random.randint(1, len(bp1) - 1)
    o1_binary, o2_binary = bp1[:cp] + bp2[cp:], bp2[:cp] + bp1[cp:]
    o1_decimal, o2_decimal = int(o1_binary, 2), int(o2_binary, 2)
    offspring.extend([o1_decimal, o2_decimal])
    mating_pool_data.extend([{'Subject No': i + 1, 'Value': p1, 'Mating Pool
(Binary)': bp1, 'Crossover Point': cp, 'Fitness': p1**2},
                             {'Subject No': i + 2, 'Value': p2, 'Mating Pool
(Binary)': bp2, 'Crossover Point': cp, 'Fitness': p2**2}])
    crossover_data.append({'Parent 1': p1, 'Parent 2': p2, 'Binary Parent 1':
bp1, 'Binary Parent 2': bp2, 'Crossover Point': cp,
                           'Offspring 1 Binary': o1_binary, 'Offspring 2 Binary':
o2_binary, 'Offspring 1 Decimal': o1_decimal, 'Offspring 2 Decimal': o2_decimal})
  return offspring, pd.DataFrame(mating_pool_data), pd.DataFrame(crossover_data)

def mutate_offs(offspring, mut_rate):
  mut_offsp, mutation_data = [], []
  for i, indiv in enumerate(offspring):
    original_binary = bin(indiv)[2:].zfill(8)
    mutated_binary, mutated_indiv, mutation_happened = original_binary, indiv,
False
    if random.random() < mut_rate:
      bin_list = list(original_binary)
      if bin_list:
          mut_point = random.randint(0, len(bin_list) - 1)
          bin_list[mut_point] = '1' if bin_list[mut_point] == '0' else '0'
          mutated_binary = "".join(bin_list)
          mutated_indiv = int(mutated_binary, 2)
          mutation_happened = True
    mut_offsp.append(mutated_indiv)
    mutation_data.append({'Subject No': i + 1, 'Offspring Before Mutation
(Binary)': original_binary,
                           'Mutation Chromosome (Binary)': mutated_binary if
mutation_happened else original_binary,
                           'Offspring After Mutation (Binary)':
bin(mutated_indiv)[2:].zfill(8),
                           'X Value (Decimal)': mutated_indiv, 'Fitness':
mutated_indiv**2})
  return mut_offsp, pd.DataFrame(mutation_data)

def gen_gen(popul, mut_rate, initial_popn):
```

```
    df_pop = pd.DataFrame({'Subject No': range(1, len(popul) + 1), 'Popul':
popul, 'Initial Popn (Binary)': [bin(p)[2:].zfill(8) for p in initial_popn]})
    fit = [ind ** 2 for ind in popul]
    cumul = sum(fit)
    prob = [f / cumul for f in fit]
    perc_prob = [p * 100 for p in prob]
    exp = [len(popul) * p for p in prob]
    actual = [round(e) for e in exp]
    df_pop['Fit'], df_pop['Prob'], df_pop['Percentage Prob'],
df_pop['ExpectCnt'], df_pop['ActualCnt'] = fit, prob, perc_prob, exp, actual
    mating_pool = sel_indiv(df_pop)
    offspring, df_mating_pool, df_crossover = cross(mating_pool)
    new_gen, df_mutation = mutate_offs(offspring, mut_rate)
    return new_gen, df_pop, df_mating_pool, df_crossover, df_mutation

popn = [12, 25, 5, 19]
initial_popn = popn[:]
curr_popul = popn[:]
best_sol, best_fit, fit_hist = None, -float('inf'), []
num_gens, mut_rate = 3, 0.01

for gen in range(num_gens):
    curr_popul, df_gen, df_mating_pool, df_crossover, df_mutation =
gen_gen(curr_popul, mut_rate, initial_popn)
    fit_vals = [ind ** 2 for ind in curr_popul]
    best_fit_curr = max(fit_vals)
    best_ind_idx = fit_vals.index(best_fit_curr)
    best_ind_curr = curr_popul[best_ind_idx]
    if best_fit_curr > best_fit: best_fit, best_sol = best_fit_curr,
best_ind_curr
    fit_hist.append(best_fit_curr)
    print(f"Gen {gen + 1}: Best Fit = {best_fit_curr}, Best Indiv =
{best_ind_curr}, Popul = {curr_popul}")
    print("Generation Data:"); display(df_gen)
    print("Mating Pool Data:"); display(df_mating_pool)
    print("Crossover Data:"); display(df_crossover)
    print("Mutation Data:"); display(df_mutation)

print("\nGenetic Algorithm finished.")
print("Overall best solution found:", best_sol)
print("Fitness of the overall best solution:", best_fit)
```

Output :

```
Genetic Algorithm finished.
Overall best solution found: 25
Fitness of the overall best solution: 625
```

Program 2

Particle Swarm Optimization for Function Optimization.

Algorithm:

Lab - 2

Gene Expression Algorithm

- Initialization
- Fitness Assignment
- Selection
- Crossover
- Mutation
- Gene expression
- Termination

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

1) Select encoding technique.
   - Use chromosome of fixed length with terminals (variables, constants) and functions (+, -, ·)

2) Initial population

S.No	Initial	Phenotype	Value	Fitness
1	+ x x	x^2	12	144
2	+ x x	2x	25	625
3	x	x	5	25
4	$-x^2$	x - 2	19	361

$\sum P(x) = 1155$

Avg = 288.75

Actual count	Expected count
1	0.5
2	2.1
0	0.08
1	1.25

3) Selection of mating pool

SNo	Selected	Cross	Offspring	Pheno
1	+ x x	2	* x +	x^2
2	+ x x	1	+ x x	2x
3	+ x x	3	+ x -	
4	$-x^2$	1	$+ x^2$	

x value	fitness
13	169
24	576
27	729
17	289

4) Crossover

Perform crossover at randomly chosen gene locations.

Max fitness = 729

5) Mutation

SNo	Offspring	Mutation	Offspring	Phenotype
1	+x+	+ → −	+x−	x+(x−)
2	+xx	None	+xx	2x
3	+x−	− → +	+x+	x + x*x
4	+x2	None	+x2	x+2

x value	Fitness
29	841
24	576
27	729
20	400

6) Gene expression and evaluation

Decode each genotype → phenotype

$\Sigma P(x) = 841 + 576 + 729 + 400$
$= 2546$

Avg = 636.5

Max = 841

Page No.: 8

7) Iterate until convergence
Repeat step 3-6 until fitness improvement converges to 0.

Pseudocode:
- Define fitness function
- Define parameters
- Create population
- Select mating pool
- Mutation after mating
- Gene expression
- Iteration
- Output best value

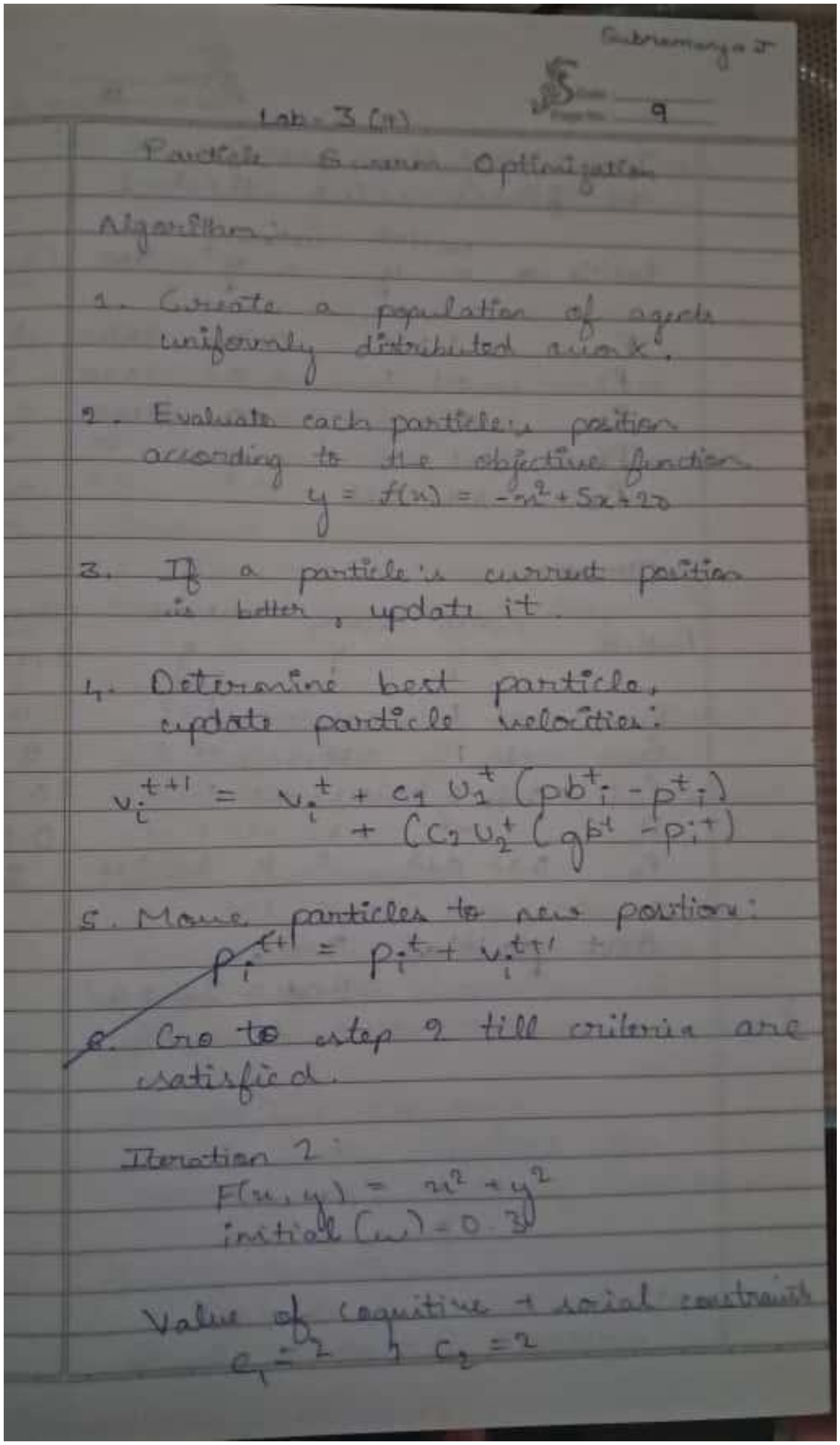
Output:
x : 26.37
$f(x)$: 695.45

Code:
```
import numpy as np

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

num_particles = 100
lower = -10
upper = 10

positions = np.random.uniform(lower, upper, num_particles)
velocities = np.random.uniform(-1, 1, num_particles)


pbestpos = np.copy(positions)
pbestval = np.array([polynomial(p) for p in positions])

gbest_position = pbestpos[np.argmin(pbestval)]
gbestval = np.min(pbestval)

w = 0.5
c1 = 2
c2 = 2

for iteration in range(1000):
  r1 = np.random.rand(num_particles)
  r2 = np.random.rand(num_particles)

  for i in range(num_particles):
    velocities[i] = w * velocities[i] + c1 * r1[i] * (pbestpos[i] - positions[i])
+ c2 * r2[i] * (gbest_position - positions[i])
    positions[i] += velocities[i]

    positions[i] = np.clip(positions[i], lower, upper)

    current = polynomial(positions[i])

    if current < pbestval[i]:
      pbestpos[i] = positions[i]
      pbestval[i] = current

    if current < gbestval:
      gbest_position = positions[i]
      gbestval = current
      #print(f"Iteration {iteration} - Best Value: {gbestval}")
print("Final solution ", gbest_position)
print("Final Best Value:", gbestval)

Output :
Final solution  -10.0
Final Best Value: -130.0
```

Program 3

Ant Colony Optimization for the Traveling Salesman Problem.

Algorithm:



Subramanya S

Lab - 3 (4) 9

Particle Swarm Optimization

Algorithm:

1. Create a population of agents uniformly distributed across X.

2. Evaluate each particle's position according to the objective function.
$y = f(x) = -x^2 + 5x + 20$

3. If a particle's current position is better, update it.

4. Determine best particle, update particle velocities:

$$v_i^{t+1} = v_i^t + c_1 U_1^t (pb_i^t - p_i^t) + (c_2 U_2^t (gb^t - p_i^t)$$

5. Move particles to new positions:

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

6. Go to step 2 till criteria are satisfied.

Iteration 2:

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

initial $(w) = 0.3$

Value of cognitive + social constraints

$c_1 = 2$, $c_2 = 2$

Initial solution are set to 10^3
P1 fitness value $1^2+1^2=2$

Particle no	Initial x	Initial y	Velocity x	Velocity y	Best	Fitness value
P1	1	1	0	0	1000	2
P2	-1	1	0	0	1000	2
P3	0.5	-0.5	0	0	100	0.5
P4	1	-1	0	0	100	2
P5	0.25	0.25	0	0	100	0.125

Iteration 2

Particle	Initial x	Initial y	Velocity x	Velocity y	Best sol x	Best sol y	Fitness value
							0.125
P1	1	1	-0.25	-0.25	1	1	0.125
P2	-1	1	1.25	-0.25	-1	1	2
P3	0.5	-0.5	-0.25	0.75	0.5	-0.5	0.5
P4	1	-1	-0.75	1.25	1	-1	0.125
P5	0.25	0.25	0	0	0.25	0.25	2

Best position $x = 5$
Best = 26.25

Code:
```
import numpy as np
import random

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

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

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

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

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

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

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

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

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

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

            if length < best_length:
                best_length = length
                best_path = path

        # Evaporate pheromone
        pheromone *= (1 - decay)

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

    return best_path, best_length

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

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


Output :
Best path: [0, 1, 4, 3, 2] with length: 9.00
```

Program 4

Cuckoo Search (CS).

Algorithm:

18.10.2025 Lab 4 Page No. 11

Ant Colony Optimization for the Travelling Salesman Problem.

Algorithm:

1. Initialization:
   Set pheromone values on all edges to small constant.
   Define number of ants and heuristic.

2. Construct Solutions
   Each ant builds a tour by moving from city to city.

3. Evaluate tours:
   Once all tours are complete, calculate cost of each tour.
   The best tour found is marked.

4. Update pheromones:
   All trails are reduced by evaporation factor (ρ). Then, ants deposit pheromones inversely proportional to distance covered.

5. Iterate and Converge
   Step 2-4 repeated for predetermined number of iterations.

6. Output Best Tour
   Outputs the shortest tour found across all iterations.

Output :
Best path : [0, 1, 4, 3, 2]
Optimal solution : 9.00

Pseudocode :

START
Initialize pheromone levels
for it = 1 to max_it :
for each ant :
place ant on start
initialize

while path not complete :
calculate probability :
- alpha
- beta
Choose best city
Add city to path

Calculate total length of constructed path.

Return best path

END

Code:
```
import numpy as np
import math

def knapsack_fitness(solution, values, weights, capacity):
    total_weight = np.sum(solution * weights)
    if total_weight > capacity:
        return 0   # Penalize overweight solutions
    return np.sum(solution * values)

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

def sigmoid(x):
    return 1 / (1 + np.exp(-x))

def cuckoo_search_knapsack(values, weights, capacity, n_nests=25, miter=100,
pa=0.25):

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

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

    Lambda = 1.5   # Levy flight exponent

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

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

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

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

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

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

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

    return best_solution, best_fitness

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

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

    print("\nBest solution found:")
    print(best_sol)
    print("Total value:", best_val)
    print("Total weight:", np.sum(best_sol * weights))
```

```
Output :
Iteration 1/100, Best Fitness: 280
Iteration 10/100, Best Fitness: 360
Iteration 20/100, Best Fitness: 360
Iteration 30/100, Best Fitness: 360
Iteration 40/100, Best Fitness: 360
Iteration 50/100, Best Fitness: 360
Iteration 60/100, Best Fitness: 360
Iteration 70/100, Best Fitness: 360
Iteration 80/100, Best Fitness: 360
Iteration 90/100, Best Fitness: 360
Iteration 100/100, Best Fitness: 360

Best solution found:
```

```
[1 1 1 1 0]
Total value: 360
Total weight: 100
```

Program 5

Grey Wolf Optimizer (GWO).

Algorithm:

17.10.2025 Lab- 5

Cuckoo Search Algorithm

Pseudocode

Set initial host nest size n
Set max iterations Maxt
For (i=1 : i<=n) do
    Generate initial population of n host x_i^t.
    Evaluate Fitness Function $f(x_i^t)$
End for
Generate a new solution (cuckoo) x_i^{t+1} random by Levy Flight
Evaluate Fitness function x_i^{t+1}
    ie $f(x_i^{t+1})$
Choose a nest x_j among n solutions randomly.
If $(f(x_i^{t+1}) > f(x_j^t))$ then
Replace the solution x_j with x_i^{t+1}
End if
Abandon Pa
Build new nest at new location using Levy flight a fraction Pa,
Keep best solution
Rand and find current best solution
$t = t+1$ till $t \geq$ Maxt

Return best solution

Output:

Best solution : [1 1 1 1 0]

Date:
Page No.: 12

Iteration [10/100]: 360

⋮

Iteration [100/100]: 360

Total value: 360
Total weight: 100

Code:

```
import numpy as np

def sphere(x):
    return np.sum(x**2)

class GreyWolfOptimizer:
```

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

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

        self.alpha_pos = np.zeros(self.dim)
        self.alpha_score = float('inf')

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

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

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

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

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

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

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

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

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

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

        return self.alpha_pos, self.alpha_score


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

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

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


Output :
Best Position: [-0.84143788  0.86909036  0.62871764 -0.69388586 -0.30850344]
Best Score: 2.435273584330572
```

Program 6

Parallel Cellular Algorithms and Programs.

Algorithm:

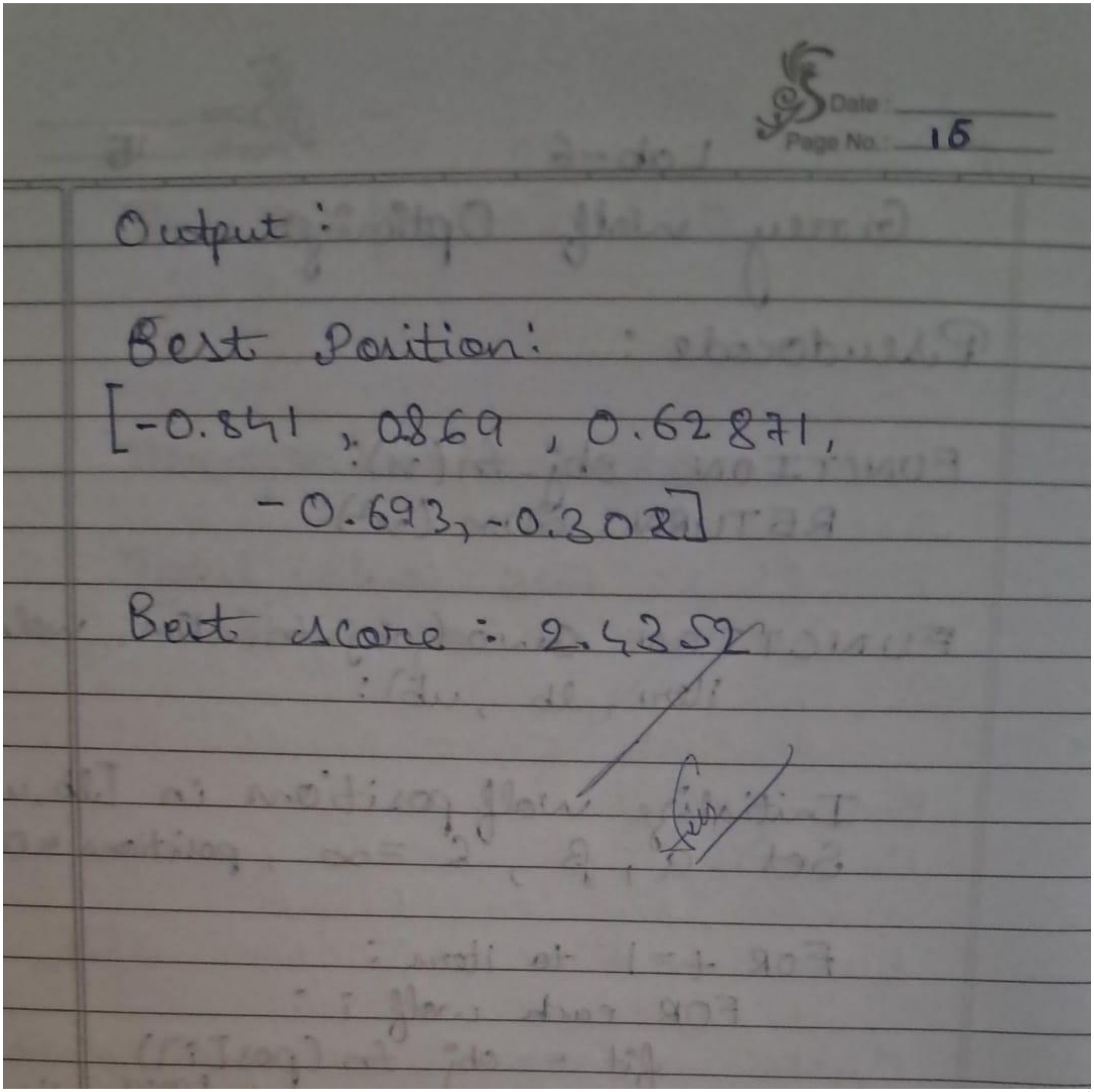
Lab-6

Page No. 15

Grey wolf Optimizer

Pseudocode :

```
FUNCTION obj_fn(x):
    RETURN SUM (x^2)

FUNCTION GWO (obj_fn, dim, wolves,
        iters, lb , ub):

    Initialize wolf positions in [lb, ub]
    Set α, β, δ = ∞ , positions = 0

    FOR t=1 to iters :
        FOR each wolf i :
            fit = obj_fn (pos[i])
            Update α, β, δ based on fit

        a = 2 - t*(2 / iters)

        FOR each wolf i :
            FOR each dimension j :
                FOR k in {α, β, δ} :
                    Generate r1, r2 ∈ [0,1]
                    A = 2* a * r1 - a
                    C = 2*r2
                    D = |C* k_pos[j] - pos[i][j]|
                    Xk = k_pos [j] - A* D
                pos [i][j] = (Xα + Xβ + Xδ)/
            Clamp pos[i] to [lb, ub]

    PRINT t, Alpha_Score, Alpha_Pos
```


Date:
Page No.: 15

Output:

Best Position:
[-0.841, 0.869, 0.62871,
-0.693, -0.308]

Best score: 2.4352

Code:

```
import numpy as np

# Initialize
grid = np.random.uniform(low=-10, high=10, size=(10, 10))
num_iterations = 100

# Define fitness function
def fitness_function(x):
    return x**2 - 4*x + 4

# Iterate
for iteration in range(num_iterations):
```

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

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

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

```
Output :
Total iterations performed: 100
Best x value found: 0.317779
Corresponding fitness (minimum f(x)): 2.829867
Algorithm converged toward x ≈ 2, where f(x) = 0 (expected optimum).
```

Program 7

Optimization via Gene Expression Algorithms.

Algorithm:

Page No: 17

17.10.2025 Lab-7

Parallel Cellular Algorithm

Pseudocode:

```
PARALLEL_CELLULAR_ALGORITHM():
    grid = initialize_grid(
                size=10,
                value_range(-10,10))
    for _ in range(100):
        fitness = compute_fitness(
                    grid,
                    f = x: x**2 - 4*x+4)
        for cell in grid:
            neighbours = get_neighbours(
                    cell, grid, wrap=True)
            best = select_best(neighbours,
                        fitness)
            cell.value = average
                (neighbours.values)

    return find_best(grid, fitness)
```

Output:

Total iterations performed: 100

Best x value: 0.317779

Corresponding values (f(x))
= 2.829867

Code:

```
import random
import math
def fitness_function(x):
    return x * math.sin(10 * math.pi * x) + 2
POPULATION_SIZE = 6
GENE_LENGTH = 10
MUTATION_RATE = 0.05
CROSSOVER_RATE = 0.8
GENERATIONS = 20
DOMAIN = (-1, 2)

def random_gene():
    return random.uniform(DOMAIN[0], DOMAIN[1])

def create_chromosome():
    return [random_gene() for _ in range(GENE_LENGTH)]

def initialize_population(size):
    return [create_chromosome() for _ in range(size)]

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

def express_gene(chromosome):
    return sum(chromosome) / len(chromosome)

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

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

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

def gene_expression_algorithm():
```

```python
    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 = 2.3125, Best x = 0.4262
Generation 2: Best Fitness = 2.3125, Best x = 0.4262
Generation 3: Best Fitness = 2.3125, Best x = 0.4262
Generation 4: Best Fitness = 2.3125, Best x = 0.4262
Generation 5: Best Fitness = 2.3125, Best x = 0.4262
Generation 6: Best Fitness = 2.3125, Best x = 0.4262
Generation 7: Best Fitness = 2.3125, Best x = 0.4262
Generation 8: Best Fitness = 2.4233, Best x = 0.6237
Generation 9: Best Fitness = 2.4233, Best x = 0.6237
Generation 10: Best Fitness = 2.4233, Best x = 0.6237
Generation 11: Best Fitness = 2.4233, Best x = 0.6237
Generation 12: Best Fitness = 2.4233, Best x = 0.6237
Generation 13: Best Fitness = 2.4233, Best x = 0.6237
Generation 14: Best Fitness = 2.4233, Best x = 0.6237
Generation 15: Best Fitness = 2.4233, Best x = 0.6237
Generation 16: Best Fitness = 2.4233, Best x = 0.6237
Generation 17: Best Fitness = 2.4395, Best x = 0.4594
Generation 18: Best Fitness = 2.4395, Best x = 0.4594
Generation 19: Best Fitness = 2.4395, Best x = 0.4594
Generation 20: Best Fitness = 2.4395, Best x = 0.4594
```

```
Best solution found:
Genes: [0.6948405045559576, -0.647173288232043, -0.3013499383055478, 1.6316275489
10124, 0.9271637073163099, 0.0324867196364278, -0.3565755055362756, 1.52263966083
97925, 1.0654293190513275, 0.024805208657060707]
x = 0.4594
f(x) = 2.4395
```