VISVESVARAYA TECHNOLOGICAL UNIVERSITY

“JnanaSangama”, Belgaum -590014, Karnataka.

LAB RECORD

Bio Inspired Systems (23CS5BSBIS)

Submitted by

Skanda Mahesh (1BM23CS332)

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 Skanda Mahesh **(1BM23CS332),** 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

Github: https://github.com/Skanda-Mahesh/Bis-lab

Index

Program 1

Genetic Algorithm for Optimization Problems.

PAGE NO.
DATE / /

BIS lab

How genetic algorithm work

1) Initiatasition :
2) Evaluation
3) Crossover
4) Termination

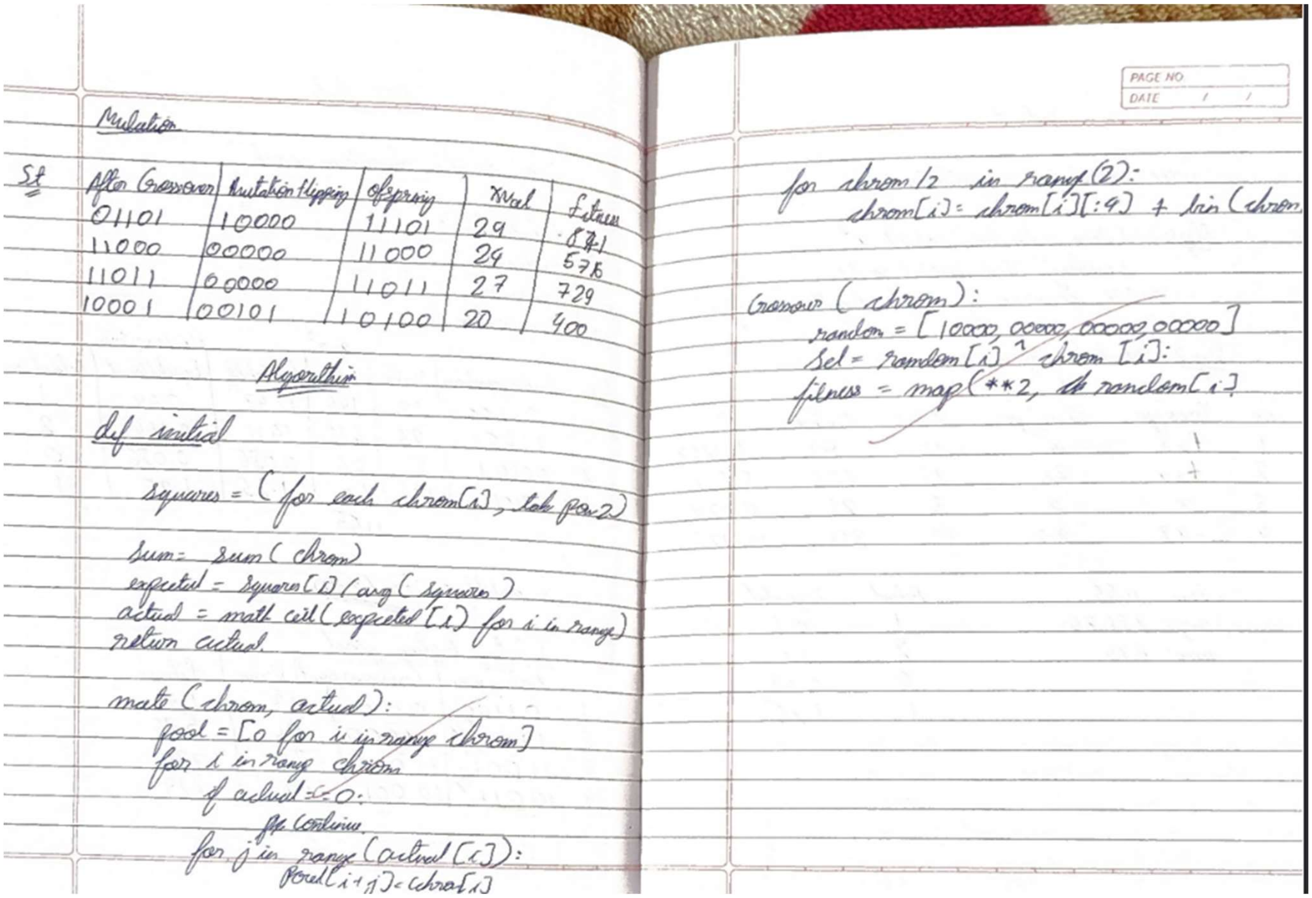
SLNO	Initial population	Value	$y = x^2$ fitness	Probability	$f(x)/\text{avg}\,\Sigma fx$ Expected Count	Actual Count
1	01100	12	144	12.47	0.49	1
2	11001	25	625	54.11	2.164	2
3	00101	5	25	0.086	0.086	0
4	10011	19	361	1.25	1.25	1
			1155			

probability : $fx / \Sigma f(x)$

Selecting Mating pool

	Mating pool	Crossover point	Value	fitness
1	01100 4	01101	13	169
2	11001 4	11000	24	576
3	11001 2	11011	27	729
4	10011 2	10001	17	289

Algorithm:



Mutation

St

After Crossover	Mutation flipping	Offspring	x val	fitness
01101	10000	11101	29	841
11000	00000	11000	24	576
11011	00000	11011	27	729
10001	00101	10100	20	400

Algorithm

def initial

squares = (for each chrom[i], take pow 2)

sum = sum(chrom)
expected = squares[i]/avg(squares)
actual = math.ceil(expected[i]) for i in range)
return actual

mate(chrom, actual):
pool = [0 for i in range chrom]
for i in range chrom
if actual == 0:
continue
for j in range(actual[i]):
Pool[i+j] = chrom[i]

PAGE NO
DATE / /

for chrom i/2 in range(2):
chrom[i] = chrom[i][:4] + bin(chrom

Crossover(chrom):
random = [10000, 00000, 00000, 00000]
Sel = random[i] ^ chrom[i]:
fitness = map(**2, random[i]

Code:

```python
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.

PAGE NO.
DATE / /

Lab-2

Particle Swarm optimization

Pseudocode

1) P = particle init
2) for i=1 till max
3) for each particle in P do:
   $F_p = f(P)$
4) if Fp is better than $F(P_p \text{ best})$
   $P_{best} = P$

end if
end for

5) g_{best} = best p in P
6) for each particle p do:
   $v_i^{t+1} = v_0^t + (c_1 u_1^t)(p^t - p_0^t) + (c_2 u_2^t)(g_b^{t} - p^t)$
7) $P_i^{t+1} = p_i^t + v_i^{t+1}$

Ex: $f(x, y) = x^2 + y^2$, inertia = 0.3
   $c_1 = 2 + c_2 = 2$

Algorithm:

particle no	Initial x	Initial y	Velocity		BestSol	Best Fi
P_1	1	1	0	0	- -	1000 2
P_2	-1	1	0	0	- -	1000 2
P_3	0.5	-0.5	0	0	- -	1000 0.
P_4	1	-1	0	0	- -	1000 2
P_5	0.25	0.25	0	0	- -	1000 0.

Iteration 2

Pno	Initial x	y	Velocity ~~Initial y~~ x	y	Best Sol		Best	Fit
P_1	0.25	0.25	-0.375	0.375	1	1	2	0.12
P_2	0.25	0.25	-0.0210	0.375	-1	1	2	0.125
P_3	0.25	0.25	0.125	0.375	0.5	0.5	0.5	0.125
P_4	0.25	0.25	-0.375	0.625	1	-1	1	0.125
P_5	0.25	0.25	0	0	0.25	0.25	0.25	0.125

Output

Best position 2.5, Best = 26.2500

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:

PAGE NO.
DATE / /

lab-3

Pseudocode

1) Initialise parameters
- num (ants)
- iterations (itr)
- Pheremone expulsion rate
- Pheremone expansion factor
- α & β values between (1 to 5)

2) Calc. distance between all citys
3) Initialise ~~[illegible]~~ pheremone on all paths
4) Record best route found and length
5) for each iteration
- for each ant
- Build complete route by choosing city based on length and phremone strength
- Complete route by returning to start
- Calculate route length, update best route if improved
- Evaporate pheromone on all lengths
- Add phremone to all paths used in current route

6) Return best route found

Output:

Input: $\begin{bmatrix} \infty & 2 & 2 & 5 & 7 \\ 2 & \infty & 9 & 8 & 2 \\ 2 & 5 & \infty & 1 & 3 \\ 5 & 7 & 1 & \infty & 2 \\ 7 & 2 & 3 & 2 & \infty \end{bmatrix}$

Best path = [0,2,3,4,1] path length 9

NG
17/10/25.

Code:
```python
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
```

```python
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:

lab-4

Step 1 Initialization

n: number of host nests x_i $(i=1,2,3 \ldots n)$

P(a): probability of discovering cuckoo egg.

Maxt: Maximum number of iterations to reach optimized solution.

Step 2 Generate Solutions

→ Apply levy flight to introduce randomness

$$x_i = x_i^t + \alpha \oplus \text{levy}(\lambda)$$

Step 3 Check Fitness

→ If the Fitness (cuckoo) > Fitness (Host)
Then next generation

Algorithm for min stairs problem

1) Set the initial value of host to P(0,1)
2) for i till n:
    generate population of n hosts
    randomly apply levi flight

    choose nest = min (Step+1) or (Step+2) cost
    place egg at said nest

    if Fitness(Step+1) > Fitness (Step+2):
        Solution = Step +1.

3) Iterate for n times till the solution converges
4) Return solution.

Output Input cost = [1,2,1,5,2,1]

Output: Minimum cost = 4.

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:
```

```python
            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:

PAGE NO.
DATE / /

Greywolf optimization

→ Representation: wolves are candidate solutions
→ Wolves model the search space, as some hone into a solution, they become alpha, Beta, delta and omega.

→ After α, β, gamma, δ are identified their position are used to identify and circulate amongst others.

→ They simulate encircling prey to get solution

→ Check the optimality of the solution, return the alpha position.

Pseudocode

1) Generate random position in a position array
2) best cost = infinite
3) for i in maximum iterations:
    for wolf in each position:
        cost = step(stairs, wolf)
        if cost < best cost:
            best cost = cost

Code:

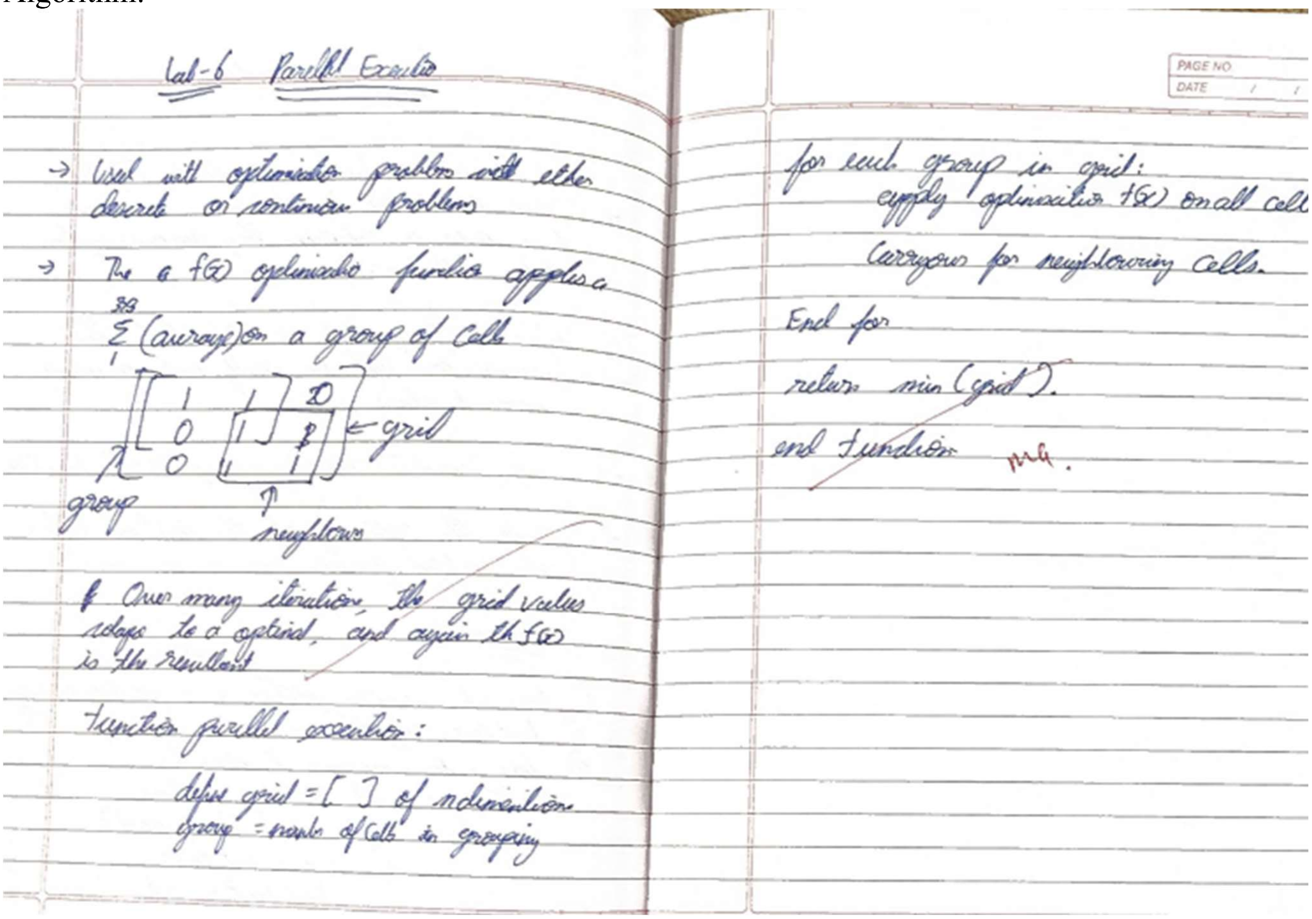
```
import numpy as np

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 Parallel Execution

→ Used with optimization problem with either discrete or continuous problem

→ The a f(x) optimization function applies a Σ (average) on a group of cells

→ Over many iterations, the grid values adapt to a optimal, and again the f(x) is the resultant

function parallel execution:

define grid = [ ] of n dimension

group = number of cells in grouping

for each group in grid:

apply optimization f(x) on all cells

carryover for neighbouring cells.

End for

return min(grid).

end function

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):
    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]
```

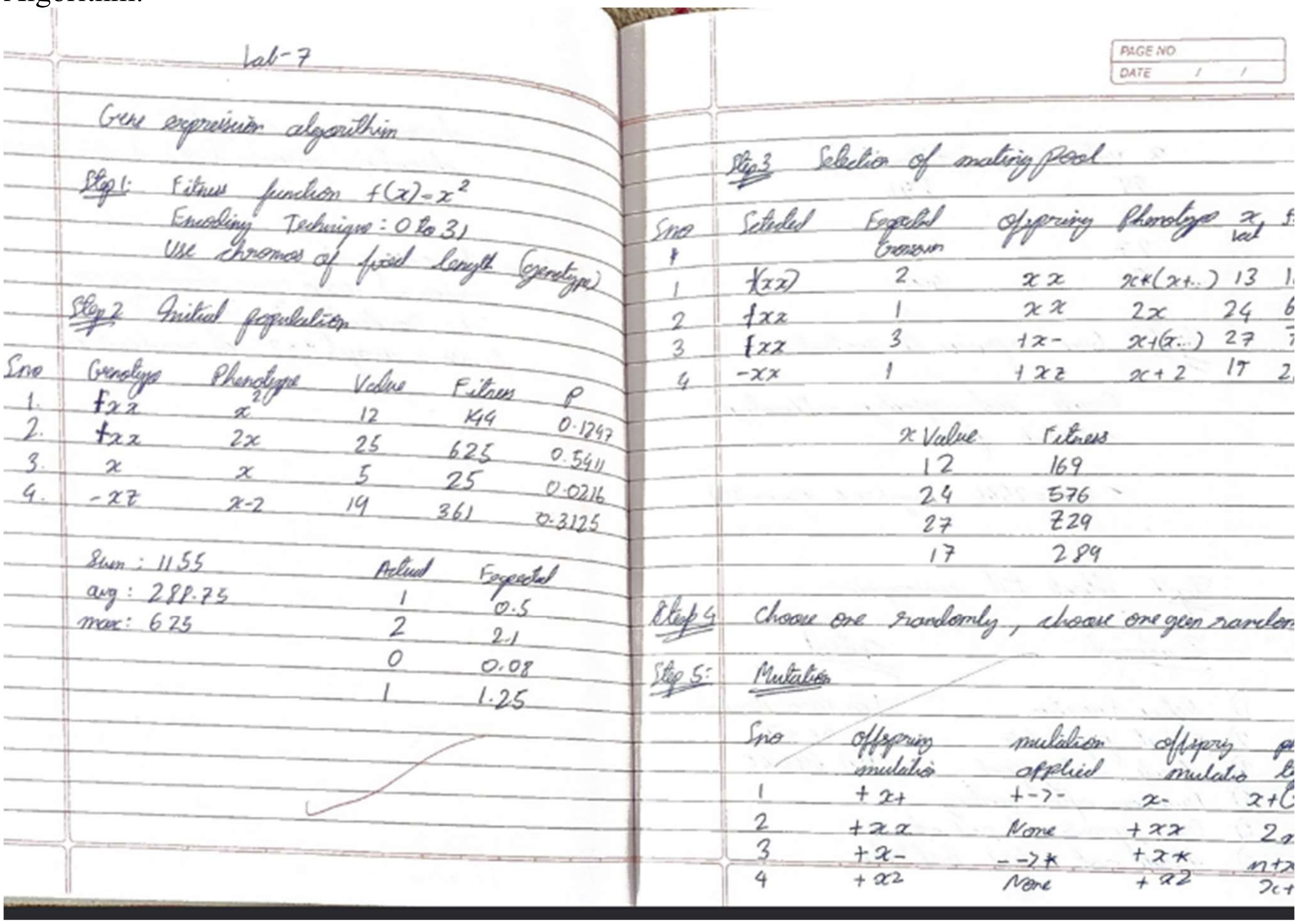
```
                    nc = (c + dc) % grid.shape[1]
                    neighbor_values.append(grid[nr, nc])
            # Update to average of neighbor values (per algorithm spec)
            new_grid[r, c] = np.mean(neighbor_values)
    grid = new_grid.copy()

# 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:



Lab-7

Gene expression algorithm

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

Encoding Technique: 0 to 31

Use chromosomes of fixed length (genotype)

Step 2: Initial population

Sno	Genotype	Phenotype	Value	Fitness	P
1.	+xx	x^2	12	144	0.1247
2.	+xx	2x	25	625	0.5411
3.	x	x	5	25	0.0216
4.	-x2	x-2	19	361	0.3125

Sum: 1155

avg: 288.75

max: 625

Actual	Expected
1	0.5
2	2.1
0	0.08
1	1.25

PAGE NO

DATE / /

Step 3 Selection of mating pool

Sno	Selected	Expected crossover	Offspring	Phenotype	x val	f
1	+(xx)	2	x x	x*(x+..)	13	1
2	+xx	1	x x	2x	24	6
3	+xx	3	+x-	x+(x..)	27	7
4	-xx	1	+x2	x+2	17	2

x Value	Fitness
12	169
24	576
27	729
17	289

Step 4 Choose one randomly, choose one gen random

Step 5: Mutation

Sno	offspring mutation	mutation applied	offspring mutation	p
1	+x+	+->-	x-	x+(
2	+xx	None	+xx	2x
3	+x-	-->*	+x*	x+x
4	+x2	None	+x2	x+

x value	fitness (x)
29	841
24	576
27	929
20	400

Step 6: Gene expression & evaluation.

Decode each genotype → Phenotype
calculate fitness

$\Sigma f(x) = 2546$ Avg: 636.5 Max = 341

Step 7 Iterate till convergence

Pseudocode

1) Define function
2) Create population
3) Create mating pool
4) Mutation after mating
5) Gene expression and evaluate
6) Iterate and output best value

Output

For 1000 iterations,
$x: 26.37$
$f(x) = 695.45$

Code:

```
import random
import math
def fitness_function(x):
    return x * math.sin(10 * math.pi * x) + 2
POPULATION_SIZE = 6
```

```python
GENE_LENGTH = 10
MUTATION_RATE = 0.05
CROSSOVER_RATE = 0.8
GENERATIONS = 20
DOMAIN = (-1, 2)

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

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

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

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

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

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

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

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

def gene_expression_algorithm():
    population = initialize_population(POPULATION_SIZE)
    best_solution = None
    best_fitness = float("-inf")
    for generation in range(GENERATIONS):
        fitnesses = evaluate_population(population)

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

```
                best_solution = chrom[:]

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

        new_population = []
        while len(new_population) < POPULATION_SIZE:
            parent1 = select(population, fitnesses)
            parent2 = select(population, fitnesses)
            offspring1, offspring2 = crossover(parent1, parent2)
            offspring1 = mutate(offspring1)
            offspring2 = mutate(offspring2)
            new_population.extend([offspring1, offspring2])

        population = new_population[:POPULATION_SIZE]

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

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

```
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
```