# VISVESVARAYA TECHNOLOGICAL UNIVERSITY

**"JnanaSangama", Belgaum -590014, Karnataka.**

**LAB RECORD**

# Bio Inspired Systems (23CS5BSBIS)

***Submitted by***

**Sanjana Srinivas (1BM23CS301)**

***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 **Sanjana Srinivas (1BM23CS301),** 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<br>Assistant Professor<br>Department of CSE, BMSCE | Dr. Kavitha Sooda<br>Professor & HOD<br>Department of CSE, BMSCE |

# Index

Github Link:
https://github.com/sanjanasrinivas22/BIS

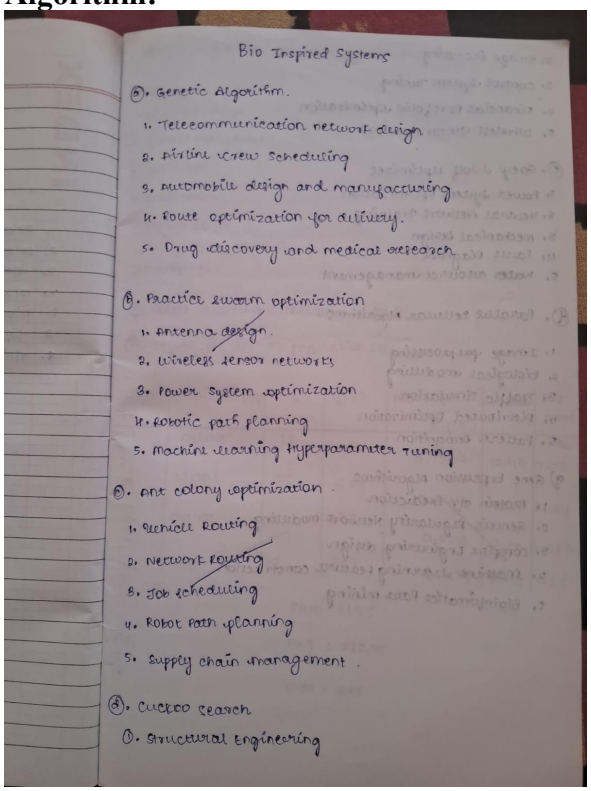
**Program 1**

Genetic Algorithm for Optimization Problems:

**Problem Statement:**

Genetic Algorithms (GA) are inspired by the principle of natural selection, where only the fittest individuals survive and reproduce. Implement a Genetic Algorithm in Python to solve an optimization problem in which the goal is to find the maximum value of a given mathematical function within a defined search space. The GA should include initialization of a population, fitness evaluation, selection, crossover, mutation, and generation-wise evolution until the optimal or near-optimal solution is found.

**Algorithm:**



2. Image Processing
3. Control System Tuning
4. Financial Portfolio Optimization
5. Wireless Sensor Network

6. Grey Wolf optimizes
4. Power System Optimization
6. Neutral Network training
8. mechanical design
10. Fault Diagnosis
12. water resource management

7. Parallel cellular Algorithms
1. Image Preprocessing
2. Biological modelling
3. Traffic simulation
4. Distributed optimization
5. Pattern Recognition

8) Gene Expression Algorithms
1. Protein structure prediction
2. Genetic Regularity Network modelling
3. complex Engineering design
4. machine learning feature construction
5. Bioinformatics Data mining

Genetic algorithm for optimization problems

Steps:-
- initialization
- fitness assignment
- selection
- cross over
- termination

$f(x) = x^2$

Iteration:-

①. select encoding technique : 0 to 31

②. select the initial population : 4

| Sn | Initial pop | x | f(x) | Prob: $\frac{f(x)}{\Sigma f(x)}$ | % Prob | Exp count |
|---|---|---|---|---|---|---|
| 1 | 01100 | 12 | 144 | 0.1247 | 12.47 | 0.499 |
| 2 | 11001 | 25 | 625 | 0.5411 | 54.11 | 2.164 |
| 3. | 00101 | 5 | 25 | 0.0216 | 2.16 | 0.086 |
| 4. | 10011 | 19 | 361 | 0.3125 | 31.25 | 1.25 |

Sum 1155

Avg : 288.75

max = 625

③. Selecting mating pool (Eliminate row 3)

| Sn | mating pool | Crossover point | offspring after crossover | x | fitness f(x) |
|---|---|---|---|---|---|
| 1 | 01\|100 | 4 | 01101 | 13 | 169 |
| 2 | 11\|001 | | 11000 | 24 | 576 |
| 3 | 11\|001 | 2 | 11011 | 27 | 729 |
| 4 | 10\|011 | | 10001 | 17 | 289 |

Let's choose 1 and 2 max → 729

④. mutation

| Sno | offspring after crossover | mutation chromosome for flipping | offspring | x |
|---|---|---|---|---|
| 1 | 01101 | 10000 | 11101 | 29 |
| 2 | 11000 | 11000 | 11000 | 24 |
| 3 | 11011 | 11011 | 11011 | 27 |
| 4 | 10001 | 10100 | 10100 | 20 |

f(x)
841
576
729
400

Sum → 2546
Avg → 636.5
max → 841

actual count
1
2 ✓
0
1

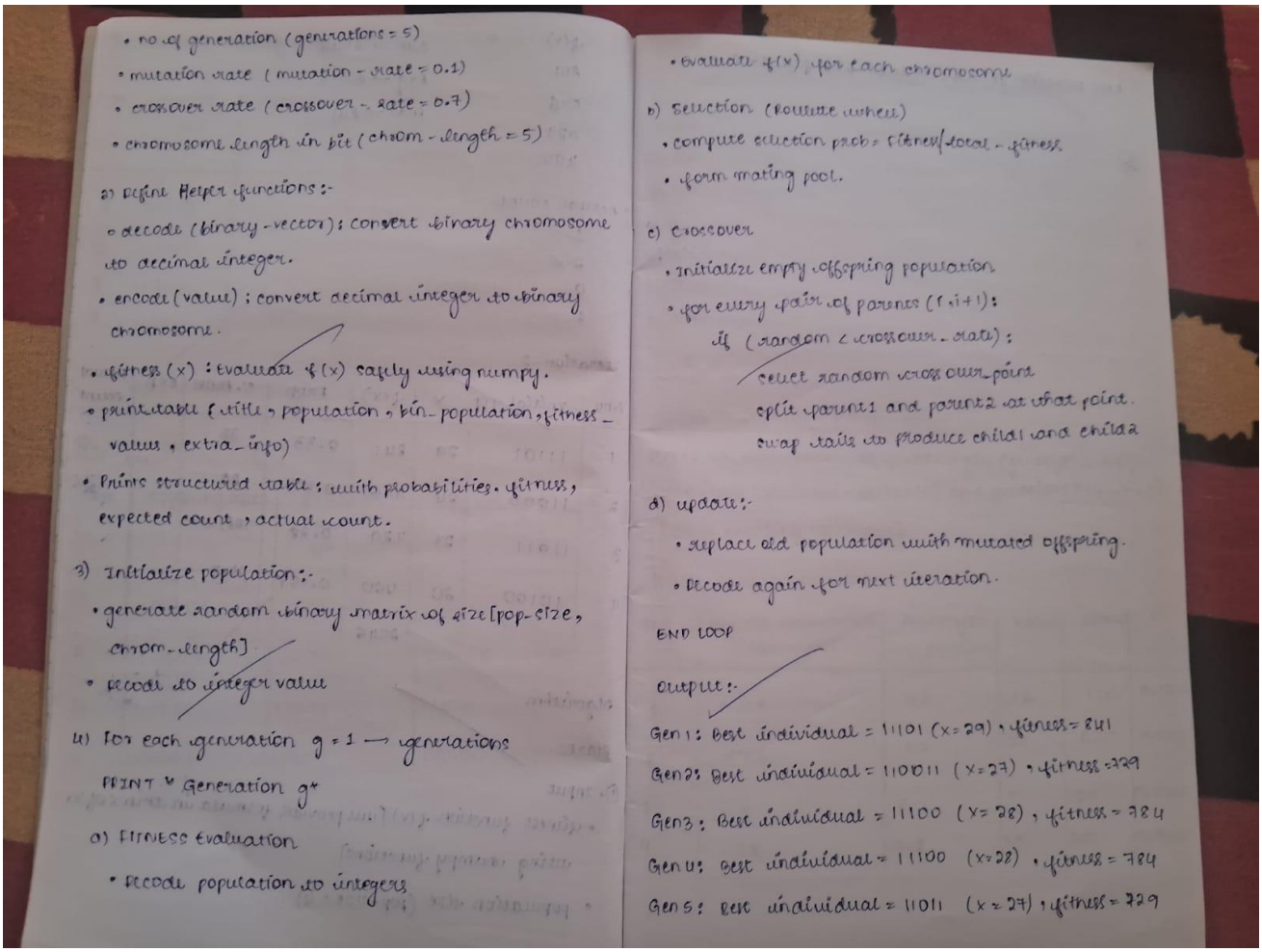
Iteration - 2

| Sno | Initial pop | x | f(x) | Prob | % Prob | Exp | Actual count |
|---|---|---|---|---|---|---|---|
| 1 | 11101 | 29 | 841 | 0.33 | 33 | | |
| 2 | 11000 | 24 | 576 | 0.22 | 22 | | |
| 3 | 11011 | 27 | 729 | 0.28 | 28 | | |
| 4 | 10100 | 20 | 400 | 0.157 | 15.7 | | |
| | | | 2546 | | | | |

Algorithm

Start

①. Input
- fitness function f(x) [user provides formula in terms of x, using numpy functions]
- population size (pop-size = 4)




**Code:**

```
import random

POP = 6
BITS = 5
GEN = 5
CROSS = 0.7
MUT = 0.1

def fit(s):
    return int(s, 2) ** 2

def init_pop():
    return [''.join(random.choice('01') for _ in range(BITS)) for _ in range(POP)]

def select(p):
    a, b = random.sample(p, 2)
    return a if fit(a) > fit(b) else b

def cross(a, b):
    if random.random() < CROSS:
```

```
        pt = random.randint(1, BITS - 1)
        return a[:pt] + b[pt:], b[:pt] + a[pt:]
    return a, b

def mutate(s):
    return ''.join('1' if c == '0' and random.random() < MUT else
                   '0' if c == '1' and random.random() < MUT else c
                   for c in s)

def run():
    pop = init_pop()
    best, best_fit = None, -1
    for g in range(1, GEN + 1):
        for x in pop:
            f = fit(x)
            if f > best_fit:
                best, best_fit = x, f
        print(f"Gen {g}: {best} (x={int(best,2)}), Fit={best_fit}")
        nxt = []
        while len(nxt) < POP:
            p1, p2 = select(pop), select(pop)
            c1, c2 = cross(p1, p2)
            nxt.extend([mutate(c1), mutate(c2)])
        pop = nxt[:POP]
    print(f"\nBest: {best} (x={int(best,2)}), Fit={best_fit}")

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

**Output:**

```
Gen 1: 11110 (x=30), Fit=900
Gen 2: 11111 (x=31), Fit=961
Gen 3: 11111 (x=31), Fit=961
Gen 4: 11111 (x=31), Fit=961
Gen 5: 11111 (x=31), Fit=961

Best: 11111 (x=31), Fit=961
```

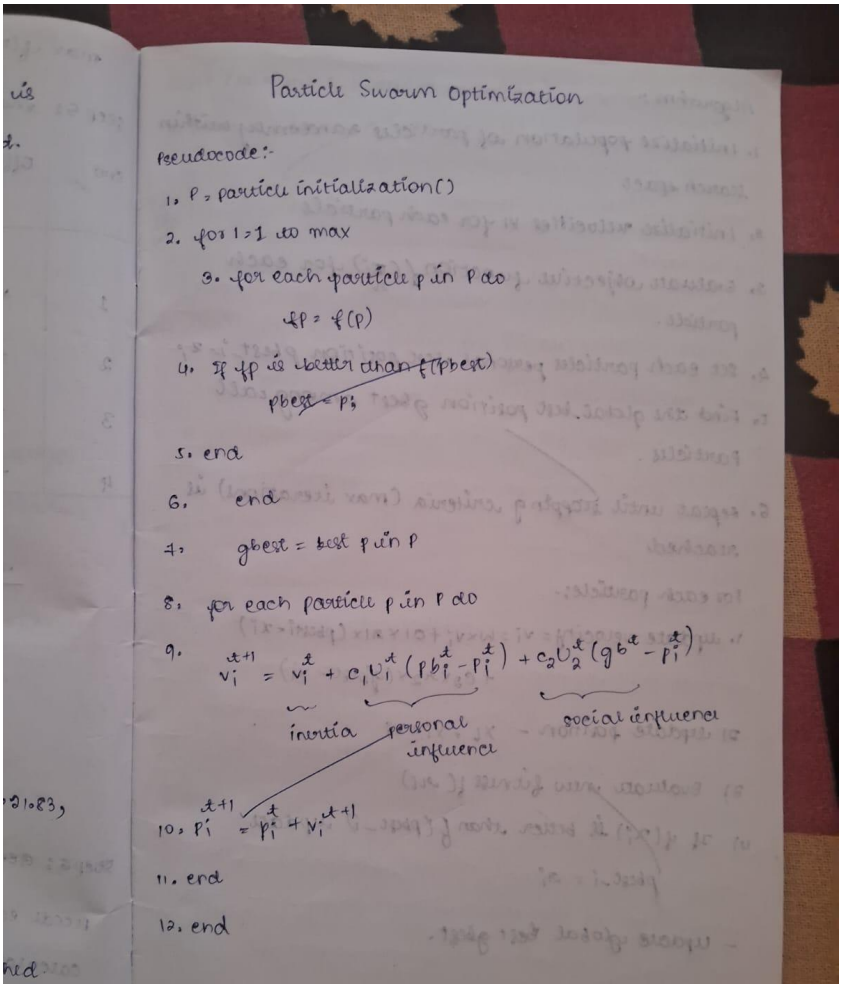
**Program 2**

Particle Swarm Optimization for Function Optimization:

**Problem Statement:**

Particle Swarm Optimization (PSO) is modeled after the collective movement of birds or fish. Implement a PSO algorithm in Python to find the minimum or maximum of a mathematical function. Each particle should update its velocity and position based on personal best and global best positions. Evaluate how swarm behavior improves the algorithm's ability to converge to optimal solutions over time.

**Algorithm:**

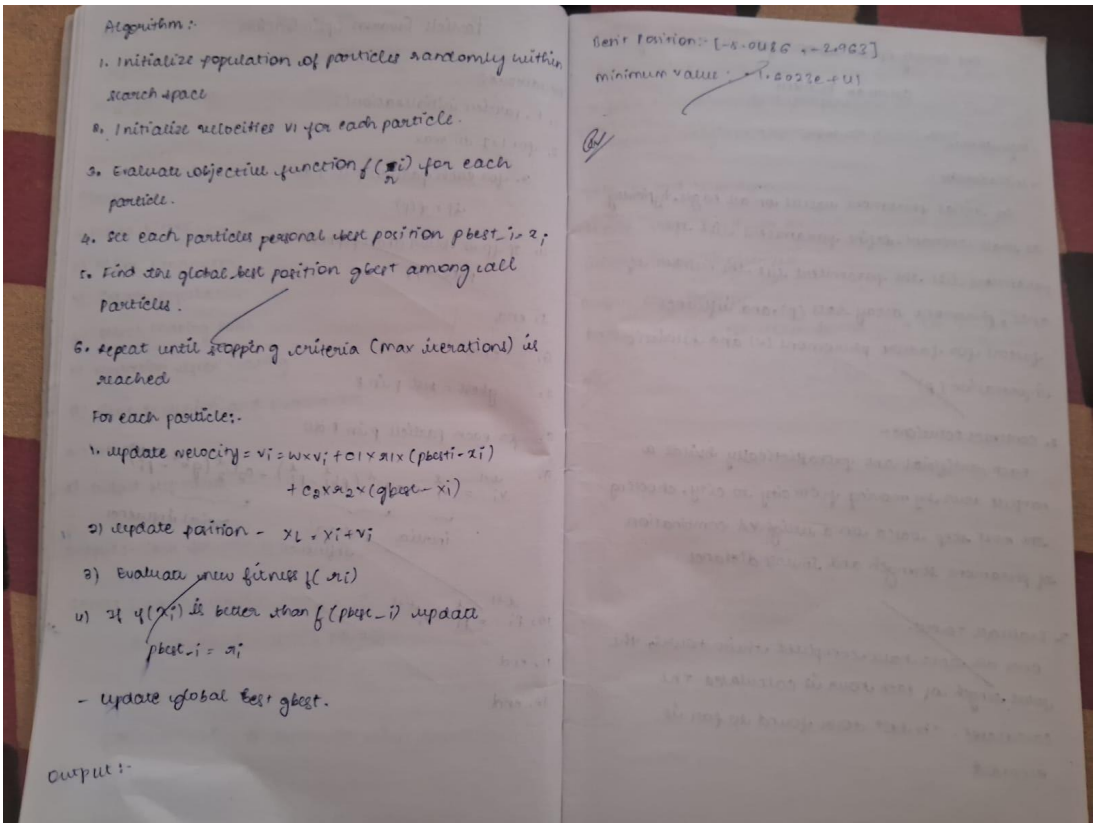


# Particle Swarm Optimization

Pseudocode:-

1. P = particle initialization()
2. for i = 1 to max
3. for each particle p in P 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. $v_i^{t+1} = v_i^t + c_1 U_1^t (pb_i^t - p_i^t) + c_2 U_2^t (gb^t - p_i^t)$

   inertia — personal influence — social influence
10. $p_i^{t+1} = p_i^t + v_i^{t+1}$
11. end
12. end




**Code:**

```
import random

# Objective (fitness) function: De Jong function
def fitness_function(position):
    x, y = position
    return x*2 + y*2  # minimize this function

# PSO parameters
num_particles = 10
num_iterations = 50
W = 0.3       # inertia weight (from PDF)
C1 = 2        # cognitive coefficient
C2 = 2        # social coefficient

# Initialize particles and velocities
particles = [[random.uniform(-10, 10), random.uniform(-10, 10)] for _ in range(num_particles)]
velocities = [[0.0, 0.0] for _ in range(num_particles)]

# Initialize personal bests
pbest_positions = [p[:] for p in particles]
pbest_values = [fitness_function(p) for p in particles]

# Initialize global best
```

```
gbest_index = pbest_values.index(min(pbest_values))
gbest_position = pbest_positions[gbest_index][:]
gbest_value = pbest_values[gbest_index]

# PSO main loop
for iteration in range(num_iterations):
    for i in range(num_particles):
        r1, r2 = random.random(), random.random()

        # Update velocity
        velocities[i][0] = (W * velocities[i][0] +
                            C1 * r1 * (pbest_positions[i][0] - particles[i][0]) +
                            C2 * r2 * (gbest_position[0] - particles[i][0]))
        velocities[i][1] = (W * velocities[i][1] +
                            C1 * r1 * (pbest_positions[i][1] - particles[i][1]) +
                            C2 * r2 * (gbest_position[1] - particles[i][1]))

        # Update position
        particles[i][0] += velocities[i][0]
        particles[i][1] += velocities[i][1]

        # Evaluate fitness
        current_value = fitness_function(particles[i])

        # Update personal best
        if current_value < pbest_values[i]:
            pbest_positions[i] = particles[i][:]
            pbest_values[i] = current_value

            # Update global best
            if current_value < gbest_value:
                gbest_value = current_value
                gbest_position = particles[i][:]

    print(f"Iteration {iteration+1}/{num_iterations} | Best Value: {gbest_value:.6f} at
{gbest_position}")

print("\nOptimal Solution Found:")
print(f"Best Position: {gbest_position}")
print(f"Minimum Value: {gbest_value}")
```

**Output:**

```
Iteration 1/50 | Best Value: -271.091815 at [-84.43244266546853, -51.11346474300606]
Iteration 2/50 | Best Value: -3147.519112 at [-990.3734271752983, -583.3861286613839]
Iteration 3/50 | Best Value: -8275.394058 at [-2605.963881015693, -1531.733147860558]
Iteration 4/50 | Best Value: -35826.154841 at [-11286.658966206636, -6626.41845422665]
Iteration 5/50 | Best Value: -69743.934634 at [-21972.964707857907, -12899.00260912271]
Iteration 6/50 | Best Value: -128492.226042 at [-40482.80512141413, -23763.307899605523]
```

Iteration 7/50 | Best Value: -2150898.767751 at [-677683.970780463, -397765.41309513245]
Iteration 8/50 | Best Value: -15310298.854777 at [-4823827.139516395, -2831322.2878723536]
Iteration 9/50 | Best Value: -49653453.407440 at [-15644353.385262355, -9182373.318457406]
Iteration 10/50 | Best Value: -269516199.282408 at [-84916691.8396995, -49841407.80150427]
Iteration 11/50 | Best Value: -2071103939.832669 at [-652544441.5664799, -383007528.3498548]
Iteration 12/50 | Best Value: -16514265252.244366 at [-5203163304.138547, -3053969321.9836354]
Iteration 13/50 | Best Value: -62522298877.396645 at [-19698952772.636093, -11562196666.06223]
Iteration 14/50 | Best Value: -303452004697.334717 at [-95608875824.94115, -56117126523.72621]
Iteration 15/50 | Best Value: -5479784825693.140625 at [-1726520368443.3489, -1013372044403.2217]
Iteration 16/50 | Best Value: -97550626289445.703125 at [-30735357062486.24, -18039956082236.613]
Iteration 17/50 | Best Value: -418935157142521.875000 at [-131994248838541.64, -77473329732719.31]
Iteration 18/50 | Best Value: -5320521856026541.000000 at [-1676341251962195.2, -983919676051075.1]
Iteration 19/50 | Best Value: -24428360527250232.000000 at [-7696663894585885.0, -4517516369039232.0]
Iteration 20/50 | Best Value: -125188235202357312.000000 at [-3.9443161518517544e+16, -2.315095608266111e+16]
Iteration 21/50 | Best Value: -667707656673289728.000000 at [-2.1037520743658112e+17, -1.2347862090006374e+17]
Iteration 22/50 | Best Value: -1030904195610660736.000000 at [-3.248078434196429e+17, -1.906442543856875e+17]
Iteration 23/50 | Best Value: -9356678590567034880.000000 at [-2.948016515513922e+18, -1.730322779769596e+18]
Iteration 24/50 | Best Value: -122704432708514906112.000000 at [-3.866058779834038e+19, -2.269162855591707e+19]
Iteration 25/50 | Best Value: -536066327314262261760.000000 at [-1.6889886416816262e+20, -9.91342994889685e+19]
Iteration 26/50 | Best Value: -2046867967951136882688.000000 at [-6.449083952375761e+20, -3.7852558873799236e+20]
Iteration 27/50 | Best Value: -1843847955187062380953 6.000000 at [-5.809427107465299e+21, -3.409812668470013e+21]
Iteration 28/50 | Best Value: -9109205869807919117107 2.000000 at [-2.870045079295888e+22, -1.684557855608071e+22]
Iteration 29/50 | Best Value: -5864383938731196725329 92.000000 at [-1.8476963312733012e+23, -1.0844956380922973e+23]
Iteration 30/50 | Best Value: -3816864039219899427454976.000000 at [-1.2025825314161544e+24, -7.058494881937953e+23]
Iteration 31/50 | Best Value: -4030267087926756938193 3056.000000 at [-1.2698196076883996e+25, -7.453139362749791e+24]
Iteration 32/50 | Best Value: -4978648926811951439617 72032.000000 at [-1.5686270634522062e+26, -9.206973999537697e+25]
Iteration 33/50 | Best Value: -8135214148890464247835787264.000000 at [-2.5631686966729733e+27, -1.5044383777722588e+27]
Iteration 34/50 | Best Value: -3979516672997569438490296 3200.000000 at [-1.253829632193114e+28, -7.359287043056707e+27]

Iteration 35/50 | Best Value: -257532895645339190122109730816.000000 at [-8.114110389727232e+28, -4.762534392539727e+28]
Iteration 36/50 | Best Value: -14730748871754494885601151549 44.000000 at [-4.641229314385222e+29, -2.7241451214920254e+29]
Iteration 37/50 | Best Value: -11593580667625126840015144353792.000000 at [-3.6527991157629925e+30, -2.143991218049571e+30]
Iteration 38/50 | Best Value: -42386731026256574797676353159168.000000 at [-1.3354822642943654e+31, -7.838542870184634e+30]
Iteration 39/50 | Best Value: -12659115795509325488873342081 4336.000000 at [-3.988518155853745e+31, -2.3410397419009176e+31]
Iteration 40/50 | Best Value: -71768775757011849213196523850 9568.000000 at [-2.2612247944819503e+32, -1.3272139933686418e+32]
Iteration 41/50 | Best Value: -33111632126200196161852166803292 16.000000 at [-1.043250950845624e+33, -6.1233065546438565e+32]
Iteration 42/50 | Best Value: -12111965937081074979620964096016384.000000 at [-3.8161271943074396e+33, -2.2398557742330984e+33]
Iteration 43/50 | Best Value: -13030577176768005038642200273590 6816.000000 at [-4.105554802589714e+34, -2.4097337857942887e+34]
Iteration 44/50 | Best Value: -73545074875879636264324252013533 5936.000000 at [-2.3171907987454107e+35, -1.360062945048571e+35]
Iteration 45/50 | Best Value: -27013953883932034780389783373789 26592.000000 at [-8.511308946686484e+35, -4.995667995279534e+35]
Iteration 46/50 | Best Value: -14748635214493649518168345315517661184.000000 at [-4.6468647792872993e+36, -2.727452827959526e+36]
Iteration 47/50 | Best Value: -43310775725245935501677583787809046528.000000 at [-1.3645962175773178e+37, -8.009425686849791e+36]
Iteration 48/50 | Best Value: -29217914173082644114569415876216776 2944.000000 at [-9.20571255038653e+37, -5.4032445361547915e+37]
Iteration 49/50 | Best Value: -15396320896619782540270998692308851884032.000000 at [-4.8509316465295593e+39, -2.8472288017803326e+39]
Iteration 50/50 | Best Value: -16023736207869237658352263206423898148 0448.000000 at [-5.0486119111390975e+40, -2.9632561927955213e+40]

Optimal Solution Found:
Best Position: [-5.0486119111390975e+40, -2.9632561927955213e+40]
Minimum Value: -1.6023736207869238e+41

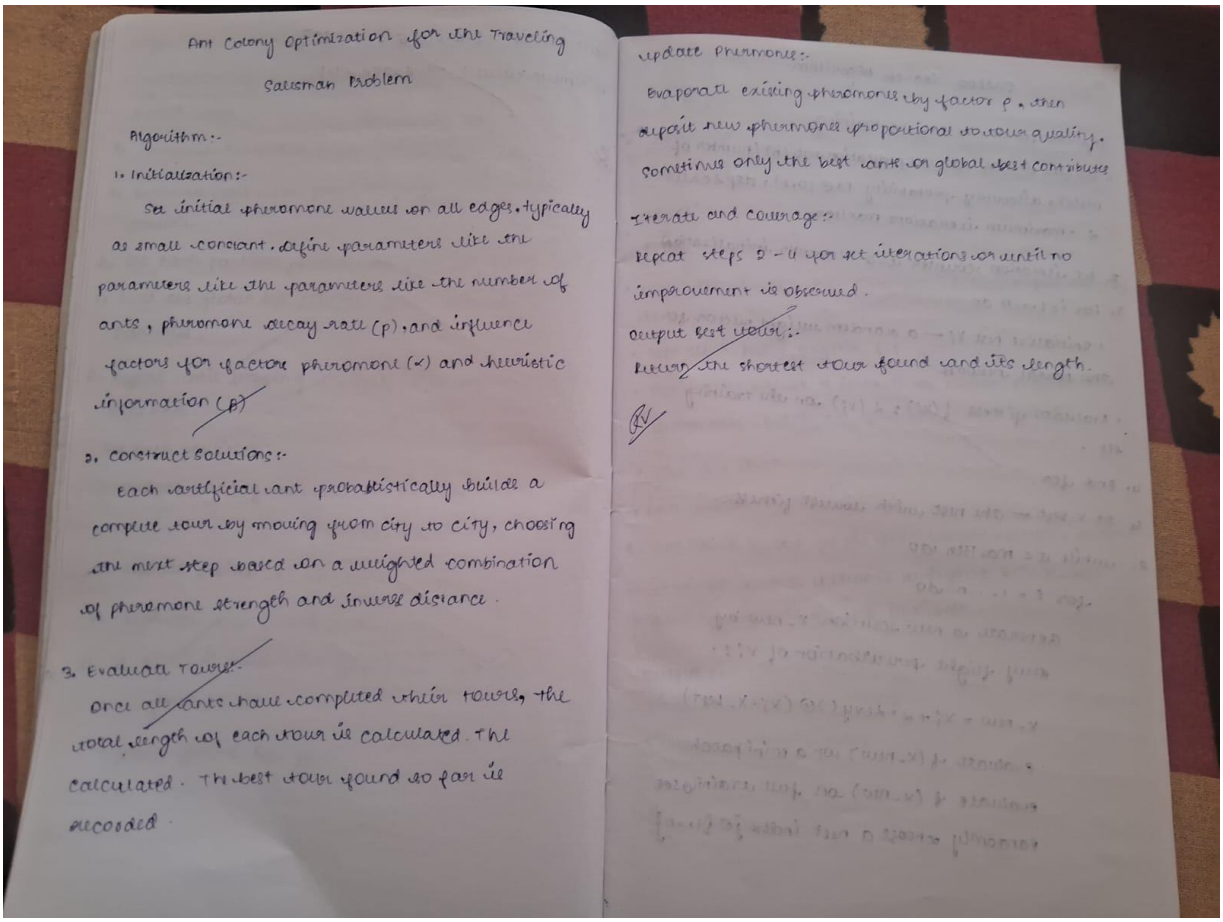
**Program 3:**

Ant Colony Optimization for the Traveling Salesman Problem:

**Problem Statement:**

Ant Colony Optimization (ACO) simulates the foraging behavior of ants to solve combinatorial optimization problems. Implement the ACO algorithm in Python to solve the Traveling Salesman Problem (TSP). Model pheromone deposition, evaporation, and probabilistic state transitions as ants traverse between cities. The objective is to identify the shortest possible Hamiltonian cycle that visits all cities exactly once and returns to the starting city.

**Algorithm:**




**Code:**

```
import random
import math

def dist(a, b):
    return math.sqrt((a[0]-b[0])**2 + (a[1]-b[1])**2)

def path_cost(route, pts):
    s = 0
    for i in range(len(route)):
        x = route[i]
        y = route[(i+1) % len(route)]
        s += dist(pts[x], pts[y])
    return s

def aco_tsp(pts, m=5, a=1, b=2, evap=0.5, it=20):
    n = len(pts)
    ph = [[1]*n for _ in range(n)]
    d = [[dist(pts[i], pts[j]) for j in range(n)] for i in range(n)]

    best_r = None
```

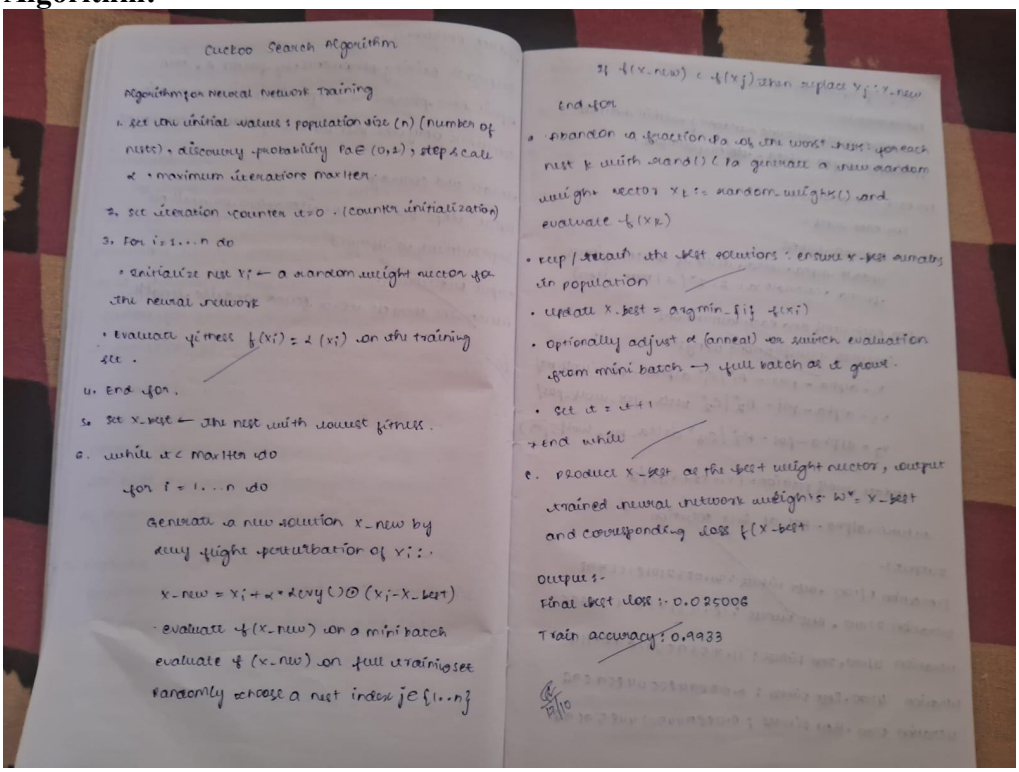
```
    best_c = float('inf')

    for _ in range(it):
        routes = []
        for _ in range(m):
            r = [random.randrange(n)]
            while len(r) < n:
                i = r[-1]
                opt = []
                for j in range(n):
                    if j not in r:
                        t = ph[i][j]**a
                        h = (1/d[i][j])**b
                        opt.append((j, t*h))
                nxt = random.choices([x for x,_ in opt], weights=[w for _,w in opt])[0]
                r.append(nxt)
            routes.append(r)

            c = path_cost(r, pts)
            if c < best_c:
                best_c = c
                best_r = r[:]

        for i in range(n):
            for j in range(n):
                ph[i][j] *= (1-evap)

        for r in routes:
            c = path_cost(r, pts)
            add = 1/c
            for i in range(n):
                x = r[i]
                y = r[(i+1) % n]
                ph[x][y] += add

    return best_r, best_c

cities = [(0,0), (1,5), (5,1), (3,4), (6,3)]
route, cost = aco_tsp(cities)
print("Best Route:", route)
print("Best Distance:", cost)
```

**Output:**
Best Route: [4, 2, 0, 1, 3]
Best Distance: 17.832452642353527

**Program 4:**
Cuckoo Search:

**Problem Statement:**
Cuckoo Search (CS) is inspired by the brood parasitism behavior of cuckoo birds. Implement the Cuckoo Search algorithm in Python using Lévy flight–based random walks to explore the global search space. Apply discovery probability to replace inferior solutions with new randomly generated nests. The objective is to apply CS to a continuous optimization problem and analyze its capability in avoiding local minima and locating optimal solutions.

**Algorithm:**




**Code:**

```
import numpy as np
import math

weights = np.array([12, 7, 11, 8, 9])
values = np.array([24, 13, 23, 15, 16])
capacity = 26
n = len(weights)

def fitness(sol):
    w = np.sum(sol * weights)
    v = np.sum(sol * values)
    return v if w <= capacity else 0

def levy_flight(l=1.5, d=n):
    a = (math.gamma(1+l) * math.sin(math.pi*l/2) /
         (math.gamma((1+l)/2) * 1 * 2**((l-1)/2))) ** (1/l)
    u = np.random.normal(0, a, d)
    v = np.random.normal(0, 1, d)
    return u / (np.abs(v) ** (1/l))
```

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

def cuckoo(pop=20, pa=0.25, it=300):
    nests = np.random.randint(0, 2, (pop, n))
    fit_vals = np.array([fitness(s) for s in nests])

    best_idx = np.argmax(fit_vals)
    best_sol = nests[best_idx].copy()
    best_fit = fit_vals[best_idx]

    for _ in range(it):
        for i in range(pop):
            step = levy_flight()
            y = nests[i] + step
            p = sig(y)
            new = (np.random.rand(n) < p).astype(int)

            f = fitness(new)
            if f > fit_vals[i]:
                nests[i] = new
                fit_vals[i] = f
                if f > best_fit:
                    best_fit = f
                    best_sol = new.copy()

        idx = np.where(np.random.rand(pop) < pa)[0]
        for j in idx:
            nests[j] = np.random.randint(0, 2, n)
            fit_vals[j] = fitness(nests[j])
            if fit_vals[j] > best_fit:
                best_fit = fit_vals[j]
                best_sol = nests[j].copy()

    return best_sol, best_fit

sol, val = cuckoo()
print("Best solution:", sol)
print("Total value:", val)
print("Total weight:", np.sum(sol * weights))
```

**Output:**

```
Best solution: [0 1 1 1 0]
Total value: 51
Total weight: 26
```

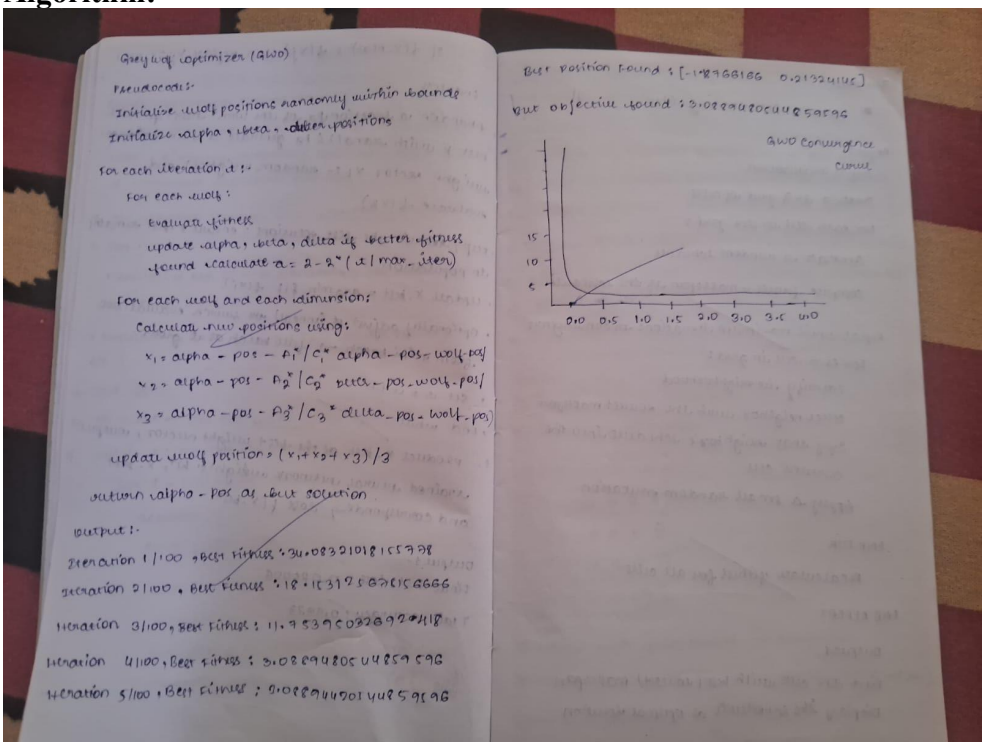
**Program 5:**

Grey Wolf Optimizer:

**Problem Statement:**

The Grey Wolf Optimizer (GWO) is inspired by the leadership hierarchy and hunting strategies of grey wolves. Implement the GWO algorithm in Python to solve a continuous optimization problem. Simulate the behaviors of alpha, beta, delta, and omega wolves during the encircling, hunting, and attacking phases. Evaluate the algorithm's ability to converge to the global minimum of a benchmark mathematical function.

**Algorithm:**




**Code:**

```
import numpy as np

def gwo(obj, dim, lb, ub, wolves=25, iters=80):
    lb = np.array(lb)
    ub = np.array(ub)

    X = np.random.uniform(lb, ub, (wolves, dim))

    a_pos = np.zeros(dim)
    b_pos = np.zeros(dim)
    d_pos = np.zeros(dim)

    a_score = float('inf')
    b_score = float('inf')
    d_score = float('inf')

    curve = []
```

```
    for t in range(iters):
        for i in range(wolves):
            X[i] = np.clip(X[i], lb, ub)
            f = obj(X[i])

            if f < a_score:
                d_score, d_pos = b_score, b_pos.copy()
                b_score, b_pos = a_score, a_pos.copy()
                a_score, a_pos = f, X[i].copy()
            elif f < b_score:
                d_score, d_pos = b_score, b_pos.copy()
                b_score, b_pos = f, X[i].copy()
            elif f < d_score:
                d_score, d_pos = f, X[i].copy()

        A = 2 - 2 * (t / iters)

        for i in range(wolves):
            r1, r2 = np.random.rand(), np.random.rand()
            A1 = 2 * A * r1 - A
            C1 = 2 * r2

            r1, r2 = np.random.rand(), np.random.rand()
            A2 = 2 * A * r1 - A
            C2 = 2 * r2

            r1, r2 = np.random.rand(), np.random.rand()
            A3 = 2 * A * r1 - A
            C3 = 2 * r2

            D1 = abs(C1 * a_pos - X[i])
            D2 = abs(C2 * b_pos - X[i])
            D3 = abs(C3 * d_pos - X[i])

            X1 = a_pos - A1 * D1
            X2 = b_pos - A2 * D2
            X3 = d_pos - A3 * D3

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

        curve.append(a_score)
        print(f"Iter {t+1}/{iters} | Best = {a_score}")

    return a_pos, a_score, curve


def sphere(x):
```

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


lb = [-10]*5
ub = [10]*5

best_x, best_f, curve = gwo(sphere, dim=5, lb=lb, ub=ub, wolves=20, iters=60)

print("\nBest position:", best_x)
print("Best fitness:", best_f)
```

**Output:**

```
Iter 1/60 | Best = 19.38453426344699
Iter 2/60 | Best = 12.847689101876695
Iter 3/60 | Best = 8.420193613082814
Iter 4/60 | Best = 1.3302224649909478
Iter 5/60 | Best = 1.3302224649909478
Iter 6/60 | Best = 0.8615491177497469
Iter 7/60 | Best = 0.4246542774007143
Iter 8/60 | Best = 0.3388244854477642
Iter 9/60 | Best = 0.18083224476768842
Iter 10/60 | Best = 0.07812973052449758
Iter 11/60 | Best = 0.03522460557660338
Iter 12/60 | Best = 0.03522460557660338
Iter 13/60 | Best = 0.012944899570136364
Iter 14/60 | Best = 0.012944899570136364
Iter 15/60 | Best = 0.006152950614212774
Iter 16/60 | Best = 0.0037310759775999102
Iter 17/60 | Best = 0.002735506845768878
Iter 18/60 | Best = 0.0011640701103155938
Iter 19/60 | Best = 0.0009104050542873242
Iter 20/60 | Best = 0.0006602536321871713
Iter 21/60 | Best = 0.00040667508175590506
Iter 22/60 | Best = 0.0002605017849749814
Iter 23/60 | Best = 0.00014268935347434156
Iter 24/60 | Best = 7.488484272466265e-05
Iter 25/60 | Best = 7.247094535557312e-05
Iter 26/60 | Best = 3.104607941020437e-05
Iter 27/60 | Best = 2.6829884971963454e-05
Iter 28/60 | Best = 1.1882729888824425e-05
Iter 29/60 | Best = 1.1229480621360995e-05
Iter 30/60 | Best = 4.845358846713108e-06
Iter 31/60 | Best = 4.65233486230689e-06
Iter 32/60 | Best = 3.3228272561699215e-06
Iter 33/60 | Best = 3.3181126472929255e-06
Iter 34/60 | Best = 2.455945937500936e-06
Iter 35/60 | Best = 1.7479393145140238e-06
Iter 36/60 | Best = 1.5948468972464486e-06
```

Iter 37/60 | Best = 1.3760549267731323e-06
Iter 38/60 | Best = 1.106879352926668e-06
Iter 39/60 | Best = 1.0341200387126218e-06
Iter 40/60 | Best = 9.679835810536737e-07
Iter 41/60 | Best = 7.270257902916992e-07
Iter 42/60 | Best = 7.270257902916992e-07
Iter 43/60 | Best = 6.234784775579161e-07
Iter 44/60 | Best = 5.832397928347616e-07
Iter 45/60 | Best = 5.584070849595501e-07
Iter 46/60 | Best = 4.911566777040607e-07
Iter 47/60 | Best = 4.533506668531795e-07
Iter 48/60 | Best = 4.0440339346248435e-07
Iter 49/60 | Best = 3.942344547235292e-07
Iter 50/60 | Best = 3.7470697488341257e-07
Iter 51/60 | Best = 3.5470071374662986e-07
Iter 52/60 | Best = 3.4764315127985757e-07
Iter 53/60 | Best = 3.3889763569838644e-07
Iter 54/60 | Best = 3.2487447793050615e-07
Iter 55/60 | Best = 3.205353914539447e-07
Iter 56/60 | Best = 3.189124599696688e-07
Iter 57/60 | Best = 3.171865169769369e-07
Iter 58/60 | Best = 3.1551459750944405e-07
Iter 59/60 | Best = 3.1361824113999946e-07
Iter 60/60 | Best = 3.1361824113999946e-07

Best position: [-0.000383  0.00029216  0.00010707  0.00023974 -0.00011242]
Best fitness: 3.1361824113999946e-07

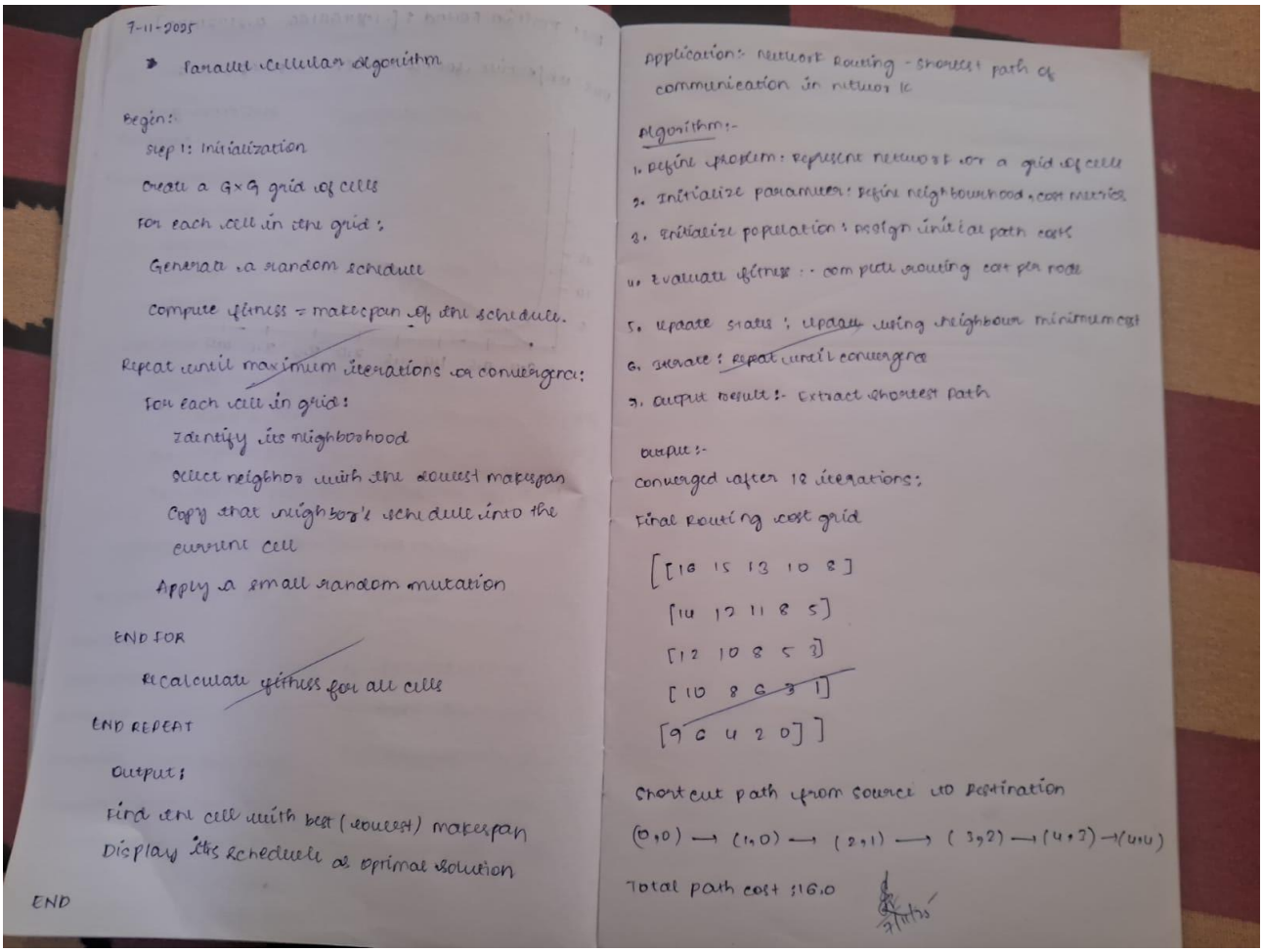
**Program 6:**

Parallel Cellular Algorithms

**Problem Statement:**

Parallel Cellular Algorithms mimic the distributed behavior of biological cells, where each cell updates its state based on interactions with neighboring cells. Implement a Parallel Cellular Optimization Algorithm in Python using a 2D grid of cells, where each cell represents a candidate solution. Define neighbor interactions and update rules that allow the algorithm to iteratively improve solutions across the grid. The objective is to evaluate how local interactions support global optimization performance, particularly for solving a numerical optimization problem.

**Algorithm:**




**Code:**

```
import numpy as np

GRID_SIZE = 5 # 5x5 network grid
MAX_ITER = 100
INF = 1e9

source = (0, 0)
destination = (4, 4)

np.random.seed(42)
cost_matrix = np.random.randint(1, 10, size=(GRID_SIZE, GRID_SIZE))

state = np.full((GRID_SIZE, GRID_SIZE), INF)
state[destination] = 0 # destination cost = 0

# Define 4-neighborhood (up, down, left, right)
neighbors = [(-1, 0), (1, 0), (0, -1), (0, 1)]


def get_neighbors(i, j):
```

```
    """Return valid neighboring cells"""
    valid_neighbors = []
    for dx, dy in neighbors:
        ni, nj = i + dx, j + dy
        if 0 <= ni < GRID_SIZE and 0 <= nj < GRID_SIZE:
            valid_neighbors.append((ni, nj))
    return valid_neighbors


for iteration in range(MAX_ITER):
    new_state = state.copy()
    for i in range(GRID_SIZE):
        for j in range(GRID_SIZE):
            if (i, j) == destination:
                continue
            neighbor_costs = []
            for ni, nj in get_neighbors(i, j):
                # Update rule: min(cost to neighbor + neighbor's state)
                total_cost = cost_matrix[ni, nj] + state[ni, nj]
                neighbor_costs.append(total_cost)
            if neighbor_costs:
                new_state[i, j] = min(neighbor_costs)
    # Check for convergence
    if np.allclose(new_state, state):
        print(f"Converged after {iteration} iterations.")
        break
    state = new_state

path = [source]
current = source
while current != destination:
    i, j = current
    nbs = get_neighbors(i, j)
    next_cell = min(nbs, key=lambda n: state[n])
    path.append(next_cell)
    current = next_cell

print("Final Routing Cost Grid:")
print(np.round(state, 2))
print("\nShortest Path from Source to Destination:")
print(" → ".join([str(p) for p in path]))
print(f"\nTotal Path Cost: {state[source]}")
```

**Output:**

```
Converged after 8 iterations.
Final Routing Cost Grid:
[[33. 30. 22. 17. 17.]
 [30. 23. 15. 12. 13.]
```

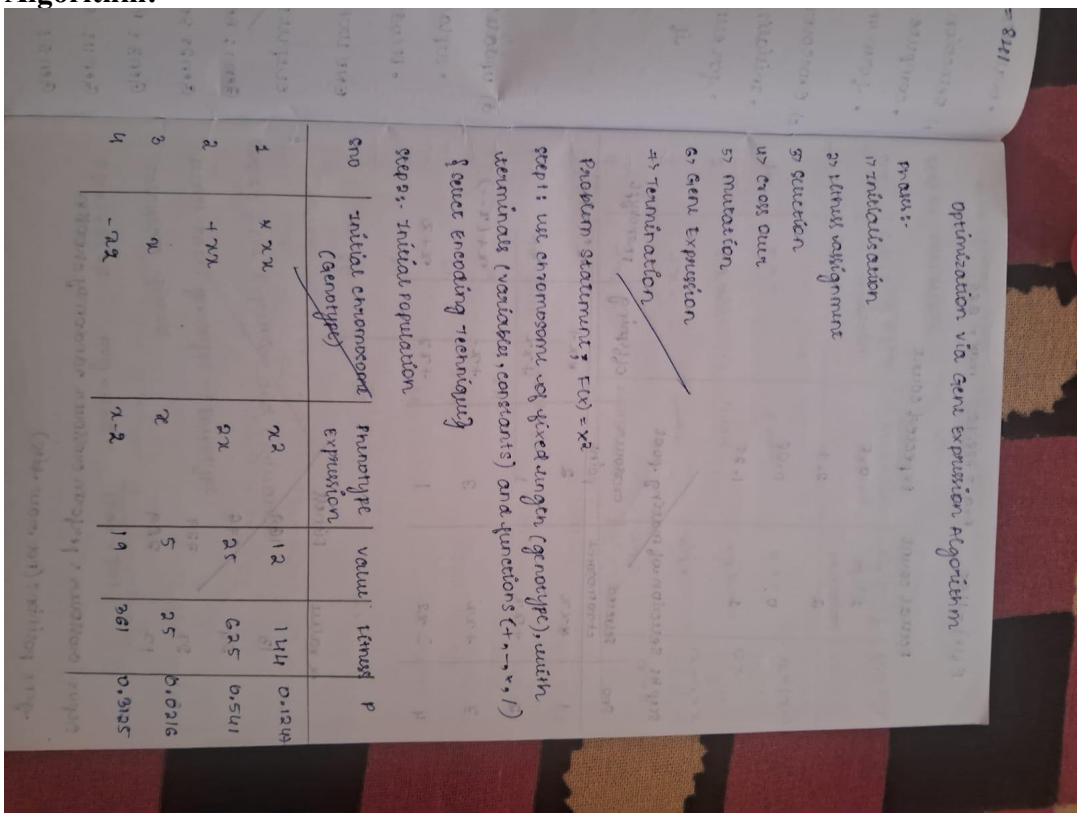
[22. 15. 12.  6.  8.]
[20. 12.  6.  4.  3.]
[19. 13.  4.  3.  0.]]

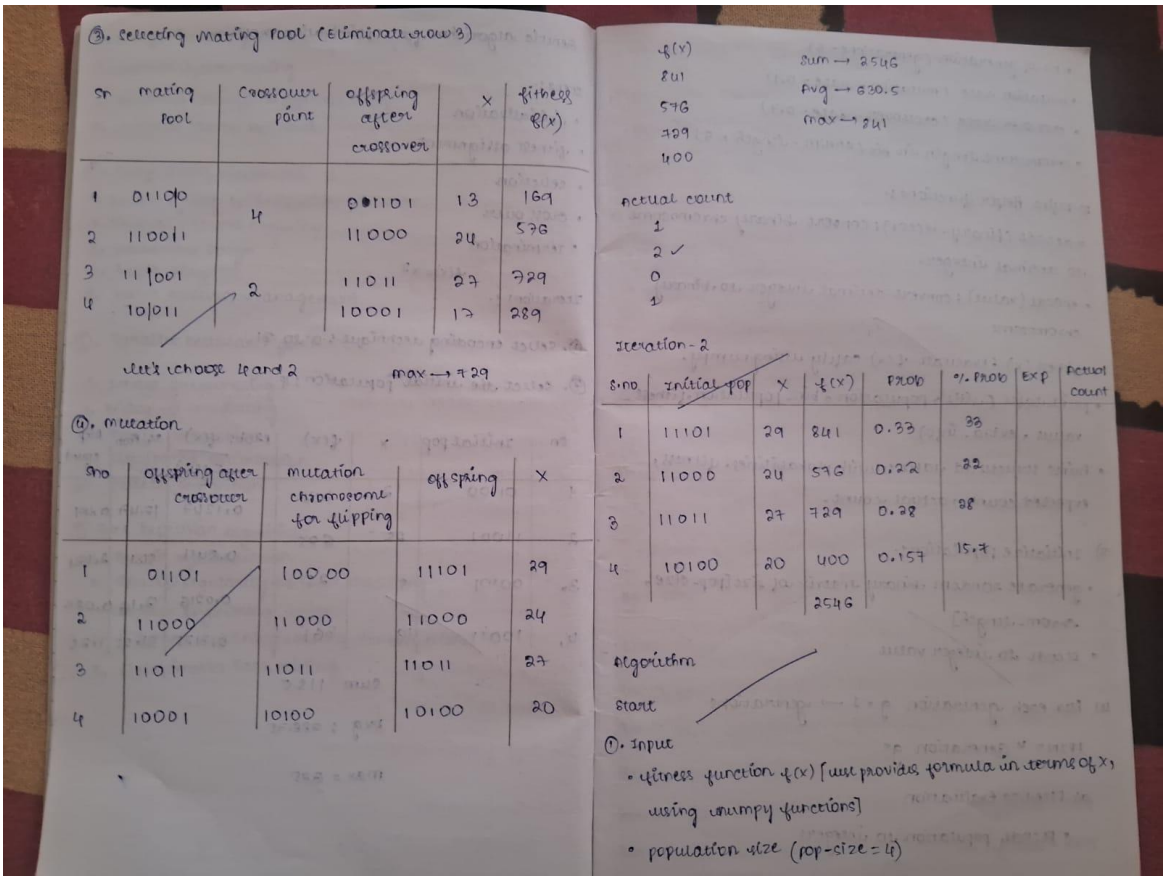
Shortest Path from Source to Destination:
(0, 0) → (1, 0) → (2, 0) → (2, 1) → (3, 1) → (3, 2) → (4, 2) → (4, 3) → (4, 4)

Total Path Cost: 33.0

**Program 7:**
Optimization via Gene Expression Algorithms
**Problem Statement:**
Gene Expression Algorithms (GEA) mimic the biological mechanism of gene expression, where DNA sequences are translated into functional proteins. Implement a GEA in Python to solve a continuous optimization problem by encoding candidate solutions as gene-like sequences. Apply selection, crossover, mutation, and expression operators to evolve the population across generations. The objective is to determine the algorithm's effectiveness in discovering optimal or near-optimal solutions.Algorithm

**Algorithm:**





③. selecting mating pool (Eliminate row 3)

| Sr | mating pool | Crossover point | offspring after crossover | x | fitness f(x) |
|---|---|---|---|---|---|
| 1 | 01\|01 | 4 | 01101 | 13 | 169 |
| 2 | 11001 | | 11000 | 24 | 576 |
| 3 | 11\|001 | 2 | 11011 | 27 | 729 |
| 4 | 10\|011 | | 10001 | 17 | 289 |

let's choose 4 and 2 max → 729

④. mutation

| Sno | offspring after crossover | mutation chromosome for flipping | offspring | x |
|---|---|---|---|---|
| 1 | 01101 | 10000 | 11101 | 29 |
| 2 | 11000 | 11000 | 11000 | 24 |
| 3 | 11011 | 11011 | 11011 | 27 |
| 4 | 10001 | 10100 | 10100 | 20 |

f(x)
841
576
729
400

sum → 2546
Avg → 636.5
max → 841

actual count
1
2 ✓
0
1

Iteration - 2

| S.no | Initial pop | x | f(x) | Prob | % Prob | Exp | Actual count |
|---|---|---|---|---|---|---|---|
| 1 | 11101 | 29 | 841 | 0.33 | 33 | | |
| 2 | 11000 | 24 | 576 | 0.22 | 22 | | |
| 3 | 11011 | 27 | 729 | 0.28 | 28 | | |
| 4 | 10100 | 20 | 400 | 0.157 | 15.7 | | |
| | | | 2546 | | | | |

algorithm

start

①. Input
- fitness function f(x) [user provides formula in terms of x, using numpy functions]
- population size (pop-size = 4)

Step 4:- Iterate until convergence

Repeat step 3 to 6 until fitness improvement is negligible or generation limited has reached.

Pseudocode

Start

1) Defini Fitness function

2) Defini Parameter

3) Create Population

4) select mating pool

5) mutation after mating

6) Gene expression and Evaluation

7) Herate

8) output best value

Output:- (Ran for 1000 generations)

Genes: [29.53, 29.82, 29.34, 28.57, 15.09, 21.83, 23.53, 30.81, 28.51, 26.23]

$x = 26.37$

$f(x) = 695.45$ # Generation limit reached

**Code:**

```
import random, math

def fitness(x):
    return x * math.sin(10 * math.pi * x) + 2

def init_pop(size):
    return [random.uniform(-1, 2) for _ in range(size)]

def select(pop, fits):
    total = sum(fits)
    r = random.uniform(0, total)
    cum = 0
    for p, f in zip(pop, fits):
        cum += f
        if cum >= r:
            return p
    return random.choice(pop)

def crossover(a, b, rate):
    if random.random() < rate:
        alpha = random.random()
        return alpha*a + (1-alpha)*b, alpha*b + (1-alpha)*a
    return a, b

def mutate(x, rate):
    if random.random() < rate:
        return random.uniform(-1, 2)
    return x

def run():
    POP, GEN = 6, 20
    CROSS, MUT = 0.8, 0.05
    pop = init_pop(POP)
    best, best_fit = None, float("-inf")

    for g in range(1, GEN+1):
        fits = [fitness(x) for x in pop]
        for x, f in zip(pop, fits):
            if f > best_fit:
                best, best_fit = x, f
        print(f"Gen {g}: Best Fitness = {best_fit:.4f}, x = {best:.4f}")

        new_pop = []
        while len(new_pop) < POP:
            p1, p2 = select(pop, fits), select(pop, fits)
            c1, c2 = crossover(p1, p2, CROSS)
```

```
            new_pop += [mutate(c1, MUT), mutate(c2, MUT)]
        pop = new_pop[:POP]

    print("\nBest solution found:")
    print(f"x = {best:.4f}")
    print(f"f(x) = {best_fit:.4f}")

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

**Output:**

```
Gen 1: Best Fitness = 3.0193, x = 1.0432
Gen 2: Best Fitness = 3.0193, x = 1.0432
Gen 3: Best Fitness = 3.0193, x = 1.0432
Gen 4: Best Fitness = 3.1817, x = 1.2402
Gen 5: Best Fitness = 3.1817, x = 1.2402
Gen 6: Best Fitness = 3.1817, x = 1.2402
Gen 7: Best Fitness = 3.1817, x = 1.2402
Gen 8: Best Fitness = 3.1817, x = 1.2402
Gen 9: Best Fitness = 3.1817, x = 1.2402
Gen 10: Best Fitness = 3.1817, x = 1.2402
Gen 11: Best Fitness = 3.1817, x = 1.2402
Gen 12: Best Fitness = 3.1817, x = 1.2402
Gen 13: Best Fitness = 3.1817, x = 1.2402
Gen 14: Best Fitness = 3.1817, x = 1.2402
Gen 15: Best Fitness = 3.1817, x = 1.2402
Gen 16: Best Fitness = 3.1817, x = 1.2402
Gen 17: Best Fitness = 3.1817, x = 1.2402
Gen 18: Best Fitness = 3.1817, x = 1.2402
Gen 19: Best Fitness = 3.1817, x = 1.2402
Gen 20: Best Fitness = 3.1817, x = 1.2402

Best solution found:
x = 1.2402
f(x) = 3.1817
```