VISVESVARAYA TECHNOLOGICAL UNIVERSITY

"JnanaSangama", Belgaum -590014, Karnataka.

LAB RECORD

Bio Inspired Systems (23CS5BSBIS)

Submitted by

Shreya Sathyanarayana (1BM23CS318)

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 **Shreya Sathyanarayana (1BM23CS318),** 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

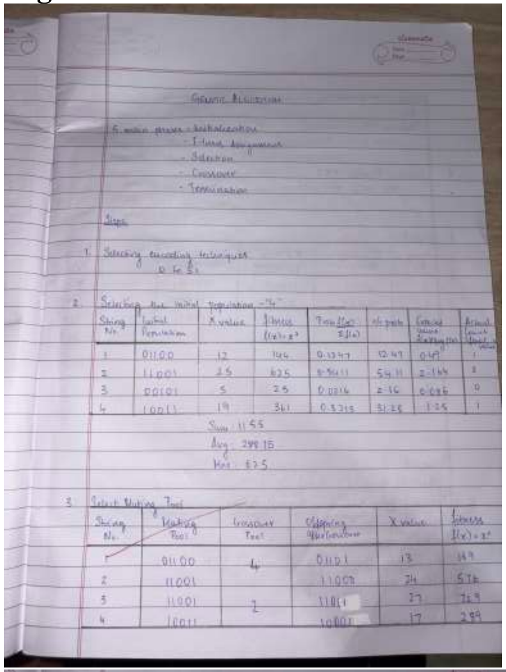
Index

Github Link:
https://github.com/shreyasathyanarayana1/BIS-Lab

Program 1

Genetic Algorithm for Optimization Problems:
Genetic Algorithms (GA) are inspired by the process of natural selection and genetics, where the fittest individuals are selected for reproduction to produce the next generation. GAs are widely used for solving optimization and search problems. Implement a Genetic Algorithm using Python to solve a basic optimization problem, such as finding the maximum value of a mathematical function.

Algorithm:



Genetic Algorithm

5 main phases - Initialization
- Fitness Assignment
- Selection
- Crossover
- Termination

Steps

1. Selecting encoding techniques
0 to 31

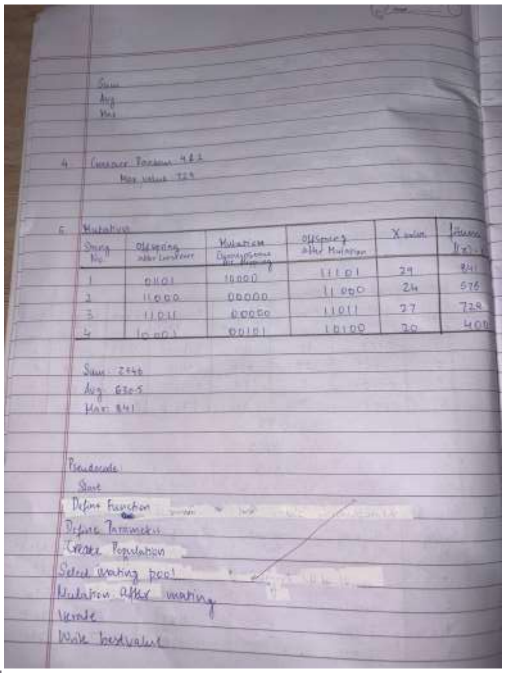
2. Selecting the initial population - "4"

String No.	Initial Population	X value	Fitness $f(x)=x^2$	Prob $\frac{f(x)}{\Sigma f(x)}$	% prob	Expected value Average f(x)	Actual count
1	01100	12	144	0.1247	12.47	0.49	1
2	11001	25	625	0.5411	54.11	2.164	2
3	00101	5	25	0.0216	2.16	0.086	0
4	10011	19	361	0.3125	31.25	1.25	1

Sum: 1155
Avg: 288.75
Max: 625

3. Select Mating Pool

String No.	Mating Pool	Crossover Point	Offspring after crossover	X value	Fitness $f(x)=x^2$
1	01100	4	01101	13	169
2	11001		11000	24	576
3	11001	2	11011	27	729
4	10011		10001	17	289



Sum
Avg
Max

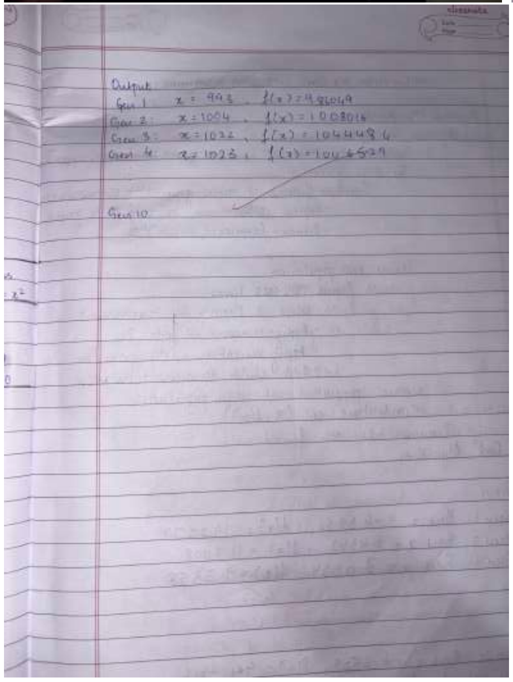
4. Crossover Random 4 & 2
Max value 729

5. Mutation

String No.	Offspring after crossover	Mutation Chromosome for flipping	Offspring after Mutation	X value	Fitness $f(x)$
1	01101	10000	11101	29	841
2	11000	00000	11000	24	576
3	11011	00000	11011	27	729
4	10001	00101	10100	20	400

Sum: 2546
Avg: 636.5
Max: 841

Pseudocode:
```
 Start
Define Function
Define Parameters
Create Population
Select mating pool
Mutation after mating
Iterate
Write bestvalue
```

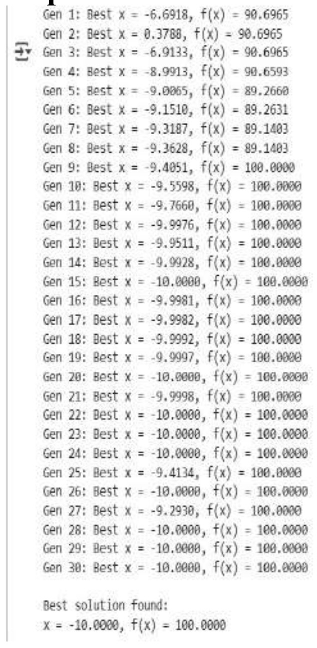


```
Output
Gen 1    x = 993 ,  f(x) = 986049
Gen 2:   x = 1004 , f(x) = 1008016
Gen 3:   x = 1022 , f(x) = 1044484
Gen 4:   x = 1023 , f(x) = 1046529

Gen 10
```

Code:
```
import random

def objective_function(x):
    return x ** 2

POP_SIZE = 20
GENS = 30
CROSSOVER_RATE = 0.8
MUTATION_RATE = 0.1
BOUNDS = [-10, 10]


def create_population(size):
    return [random.uniform(BOUNDS[0], BOUNDS[1]) for _ in range(size)]

def evaluate(population):
    return [objective_function(ind) for ind in population]

def select(population, fitness):
    i, j = random.sample(range(len(population)), 2)     return
population[i] if fitness[i] > fitness[j] else population[j]


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


def mutate(ind):
    if random.random() < MUTATION_RATE:
        ind += random.uniform(-1, 1)
        ind = max(min(ind, BOUNDS[1]), BOUNDS[0])
    return ind

def genetic_algorithm():
    population = create_population(POP_SIZE)     for
gen in range(GENS):         fitness =
evaluate(population)         new_population = []
for _ in range(POP_SIZE):
            parent1 = select(population, fitness)             parent2 = select(population,
fitness)             child = crossover(parent1, parent2)             child = mutate(child)
```

```
new_population.append(child)         population = new_population         best_idx =
fitness.index(max(fitness))         best_solution = population[best_idx]         best_fitness
= fitness[best_idx]         print(f"Gen {gen+1}: Best x = {best_solution:.4f}, f(x) =
{best_fitness:.4f}")     return best_solution, best_fitness


best_x, best_val = genetic_algorithm() print("\nBest
solution found:") print(f"x = {best_x:.4f}, f(x) =
{best_val:.4f}")
```

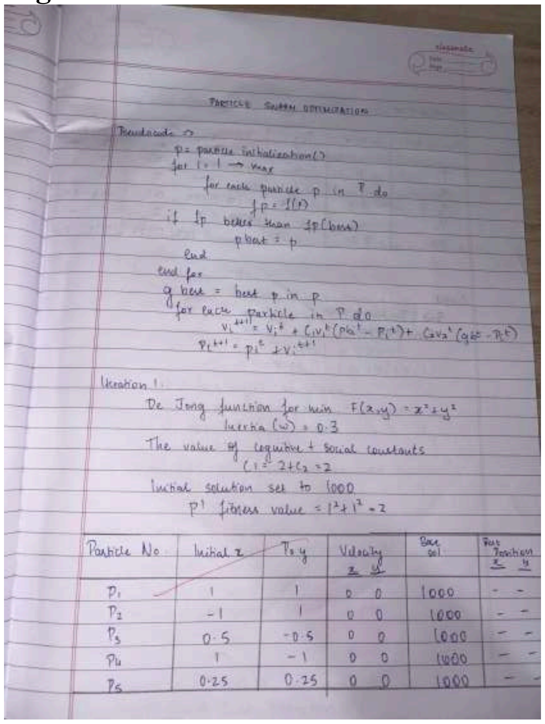
Output:



```
Gen 1: Best x = -6.6918, f(x) = 90.6965
Gen 2: Best x = 0.3788, f(x) = 90.6965
Gen 3: Best x = -6.9133, f(x) = 90.6965
Gen 4: Best x = -8.9913, f(x) = 90.6593
Gen 5: Best x = -9.0065, f(x) = 89.2660
Gen 6: Best x = -9.1510, f(x) = 89.2631
Gen 7: Best x = -9.3187, f(x) = 89.1403
Gen 8: Best x = -9.3628, f(x) = 89.1403
Gen 9: Best x = -9.4051, f(x) = 100.0000
Gen 10: Best x = -9.5598, f(x) = 100.0000
Gen 11: Best x = -9.7660, f(x) = 100.0000
Gen 12: Best x = -9.9976, f(x) = 100.0000
Gen 13: Best x = -9.9511, f(x) = 100.0000
Gen 14: Best x = -9.9928, f(x) = 100.0000
Gen 15: Best x = -10.0000, f(x) = 100.0000
Gen 16: Best x = -9.9981, f(x) = 100.0000
Gen 17: Best x = -9.9982, f(x) = 100.0000
Gen 18: Best x = -9.9992, f(x) = 100.0000
Gen 19: Best x = -9.9997, f(x) = 100.0000
Gen 20: Best x = -10.0000, f(x) = 100.0000
Gen 21: Best x = -9.9998, f(x) = 100.0000
Gen 22: Best x = -10.0000, f(x) = 100.0000
Gen 23: Best x = -10.0000, f(x) = 100.0000
Gen 24: Best x = -10.0000, f(x) = 100.0000
Gen 25: Best x = -9.4134, f(x) = 100.0000
Gen 26: Best x = -10.0000, f(x) = 100.0000
Gen 27: Best x = -9.2930, f(x) = 100.0000
Gen 28: Best x = -10.0000, f(x) = 100.0000
Gen 29: Best x = -10.0000, f(x) = 100.0000
Gen 30: Best x = -10.0000, f(x) = 100.0000

Best solution found:
x = -10.0000, f(x) = 100.0000
```

Program 2
Particle Swarm Optimization for Function Optimization:
Particle Swarm Optimization (PSO) is inspired by the social behavior of birds flocking or fish schooling. PSO is
used to find optimal solutions by iteratively improving a candidate solution with regard to a given measure of quality. Implement the PSO algorithm using Python to optimize a mathematical function.

Algorithm:



PARTICLE SWARM OPTIMIZATION

Pseudocode ⇒

```
p = particle initialization()
for i = 1 → max
    for each particle p in P do
        fp = f(p)
    if fp better than fp(best)
            pbest = p
    end
end for
gbest = best p in P
for each particle in P do
    $v_i^{t+1} = v_i^t + c_1 v_1^t (pb^t - p_i^t) + c_2 v_2^t (gb^t - p_i^t)$
    $p_i^{t+1} = p_i^t + v_i^{t+1}$
```

Iteration 1:

De Jong function for min $F(x,y) = x^2 + y^2$

Inertia (w) = 0.3

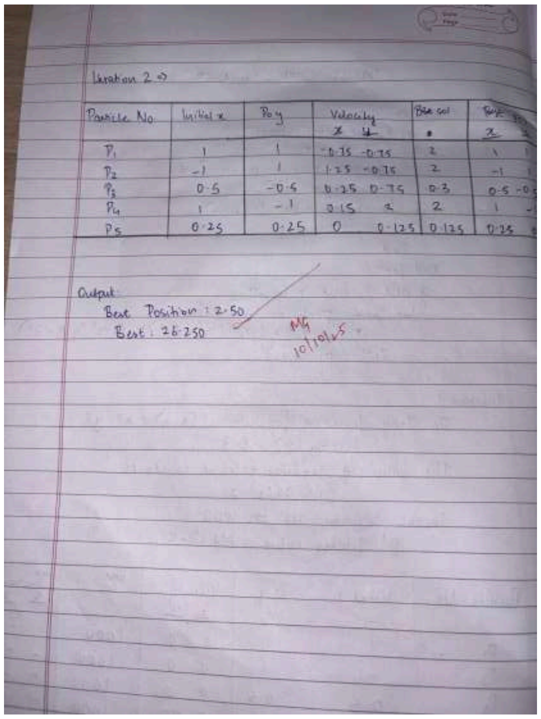
The value of cognitive + social constants

$c_1 = 2 + c_2 = 2$

Initial solution set to 1000

P^1 fitness value $= 1^2 + 1^2 = 2$

Particle No	Initial x	Po y	Velocity x	y	Best sol	Best Position x	y
P_1	1	1	0	0	1000	-	-
P_2	-1	1	0	0	1000	-	-
P_3	0.5	-0.5	0	0	1000	-	-
P_4	1	-1	0	0	1000	-	-
P_5	0.25	0.25	0	0	1000	-	-



Iteration 2 ⇒

Particle No	Initial x	Po y	Velocity x	y	Best sol	Best x	
P_1	1	1	-0.75	-0.75	2	1	
P_2	-1	1	1.25	-0.75	2	-1	
P_3	0.5	-0.5	0.25	0.75	0.3	0.5	-0
P_4	1	-1	0.15	2	2	1	-
P_5	0.25	0.25	0	0.125	0.125	0.25	

Output

Best Position : 2.50

Best : 26.250

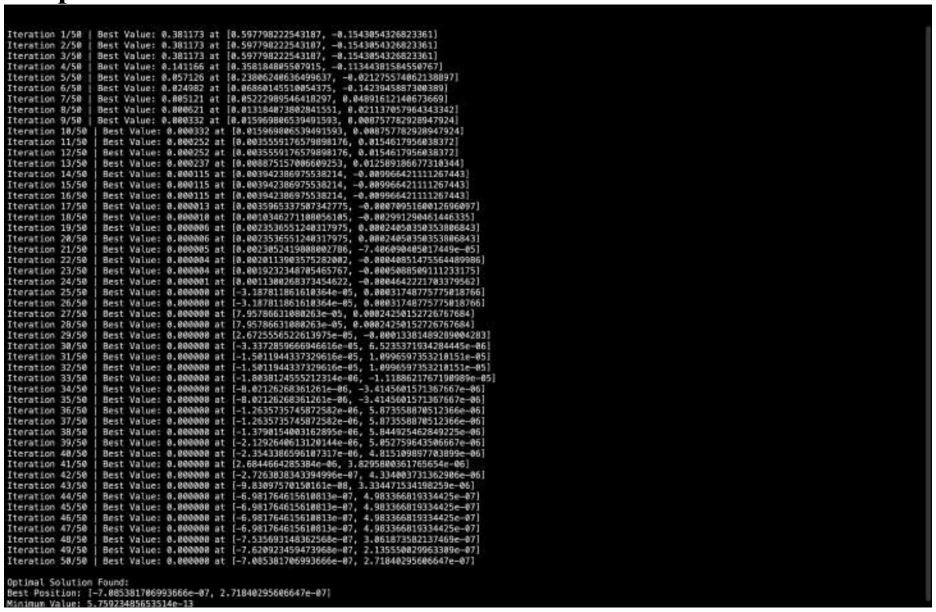
MG
10/10/25

Code:

```
import random


def fitness_function(position):

    x, y = position

    return x**2 + y**2


num_particles = 10

num_iterations = 50

W = 0.3

C1 = 2

C2 = 2
```

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

pbest_positions = [p[:] for p in particles]
pbest_values = [fitness_function(p) for p in particles]

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

for iteration in range(num_iterations):
    for i in range(num_particles):
        r1, r2 = random.random(), random.random()
        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]))
        particles[i][0] += velocities[i][0]
        particles[i][1] += velocities[i][1]
        current_value = fitness_function(particles[i])
        if current_value < pbest_values[i]:
            pbest_positions[i] = particles[i][:]
            pbest_values[i] = current_value
        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:

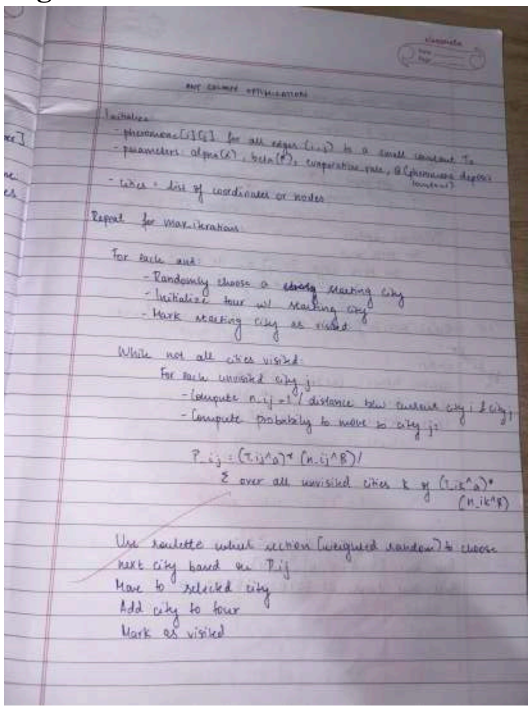


```
Iteration 1/50 | Best Value: 0.381173 at [0.597798222543187, -0.1543054326823361]
Iteration 2/50 | Best Value: 0.381173 at [0.597798222543187, -0.1543054326823361]
Iteration 3/50 | Best Value: 0.381173 at [0.597798222543187, -0.1543054326823361]
Iteration 4/50 | Best Value: 0.141166 at [0.358184805507915, -0.1134438158455076 7]
Iteration 5/50 | Best Value: 0.057126 at [0.2380624063649963 7, -0.021275574062138897]
Iteration 6/50 | Best Value: 0.024982 at [0.06860145510054375, -0.1423945887300389]
Iteration 7/50 | Best Value: 0.005121 at [0.05222989546410297, 0.04891612140673669]
Iteration 8/50 | Best Value: 0.000621 at [0.013184873802841551, 0.021137057964343342]
Iteration 9/50 | Best Value: 0.000332 at [0.015969806539491593, 0.008757782928947924]
Iteration 10/50 | Best Value: 0.000332 at [0.015969806539491593, 0.008757782928947924]
Iteration 11/50 | Best Value: 0.000252 at [0.003555917657989 8176, 0.015461795603 8372]
Iteration 12/50 | Best Value: 0.000252 at [0.0035559176579898176, 0.0154617956038372]
Iteration 13/50 | Best Value: 0.000237 at [0.0008875157006609253, 0.012589186677310344]
Iteration 14/50 | Best Value: 0.000115 at [0.003942386975538214, -0.009966421111267443]
Iteration 15/50 | Best Value: 0.000115 at [0.003942386975538214, -0.009966421111267443]
Iteration 16/50 | Best Value: 0.000115 at [0.003942386975538214, -0.009966421111267443]
Iteration 17/50 | Best Value: 0.000013 at [0.0035965337507342775, -0.0007095160012696097]
Iteration 18/50 | Best Value: 0.000010 at [0.0010346271100056105, -0.002991290461446335]
Iteration 19/50 | Best Value: 0.000006 at [0.002353655124031 7975, 0.0002405035035380 6843]
Iteration 20/50 | Best Value: 0.000006 at [0.0023536551240317975, 0.00024050350353806843]
Iteration 21/50 | Best Value: 0.000005 at [0.0023052419008002786, -7.406090405017449e-05]
Iteration 22/50 | Best Value: 0.000004 at [0.0020113903575282082, -0.00040851475564489986]
Iteration 23/50 | Best Value: 0.000004 at [0.0019232348705465767, -0.0005088509111233175]
Iteration 24/50 | Best Value: 0.000001 at [0.0011300268373454622, -0.0004642221703379562]
Iteration 25/50 | Best Value: 0.000000 at [-3.187811861610364e-05, 0.00031748775775018766]
Iteration 26/50 | Best Value: 0.000000 at [-3.187811861610364e-05, 0.00031748775775018766]
Iteration 27/50 | Best Value: 0.000000 at [7.95786631080263e-05, 0.0002425015272676 7684]
Iteration 28/50 | Best Value: 0.000000 at [7.95786631080263e-05, 0.00024250152726767684]
Iteration 29/50 | Best Value: 0.000000 at [2.6725556522613975e-05, -0.00013381489209004283]
Iteration 30/50 | Best Value: 0.000000 at [-3.3372859666946616e-05, 6.5235371934284445e-06]
Iteration 31/50 | Best Value: 0.000000 at [-1.5011944337329616e-05, 1.0996597353210151e-05]
Iteration 32/50 | Best Value: 0.000000 at [-1.5011944337329616e-05, 1.0996597353210151e-05]
Iteration 33/50 | Best Value: 0.000000 at [-1.8038124555212314e-06, -1.1188621767190989e-05]
Iteration 34/50 | Best Value: 0.000000 at [-8.02126268361261e-06, -3.4145601571367667e-06]
Iteration 35/50 | Best Value: 0.000000 at [-8.02126268361261e-06, -3.4145601571367667e-06]
Iteration 36/50 | Best Value: 0.000000 at [-1.2635735745872582e-06, 5.873558870512366e-06]
Iteration 37/50 | Best Value: 0.000000 at [-1.2635735745872582e-06, 5.873558870512366e-06]
Iteration 38/50 | Best Value: 0.000000 at [-1.3790154003162895e-06, 5.844925462849225e-06]
Iteration 39/50 | Best Value: 0.000000 at [-2.1292640613120144e-06, 5.052759643506667e-06]
Iteration 40/50 | Best Value: 0.000000 at [-2.3543386596107317e-06, 4.815109897703899e-06]
Iteration 41/50 | Best Value: 0.000000 at [2.68446642853384e-06, 3.829580036176565 4e-06]
Iteration 42/50 | Best Value: 0.000000 at [-2.7263038343394996e-07, 4.334003731362906e-06]
Iteration 43/50 | Best Value: 0.000000 at [-9.83097570150161e-08, 3.334471534198259e-06]
Iteration 44/50 | Best Value: 0.000000 at [-6.981764615610813e-07, 4.983366819334425e-07]
Iteration 45/50 | Best Value: 0.000000 at [-6.981764615610813e-07, 4.983366819334425e-07]
Iteration 46/50 | Best Value: 0.000000 at [-6.981764615610813e-07, 4.983366819334425e-07]
Iteration 47/50 | Best Value: 0.000000 at [-6.981764615610813e-07, 4.983366819334425e-07]
Iteration 48/50 | Best Value: 0.000000 at [-7.535693148362568e-07, 3.061873582137469e-07]
Iteration 49/50 | Best Value: 0.000000 at [-7.620923459473968e-07, 2.135550029963389e-07]
Iteration 50/50 | Best Value: 0.000000 at [-7.085381706993666e-07, 2.71840295606647e-07]

Optimal Solution Found:
Best Position: [-7.085381706993666e-07, 2.71840295606647e-07]
Minimum Value: 5.75923485653514e-13
```

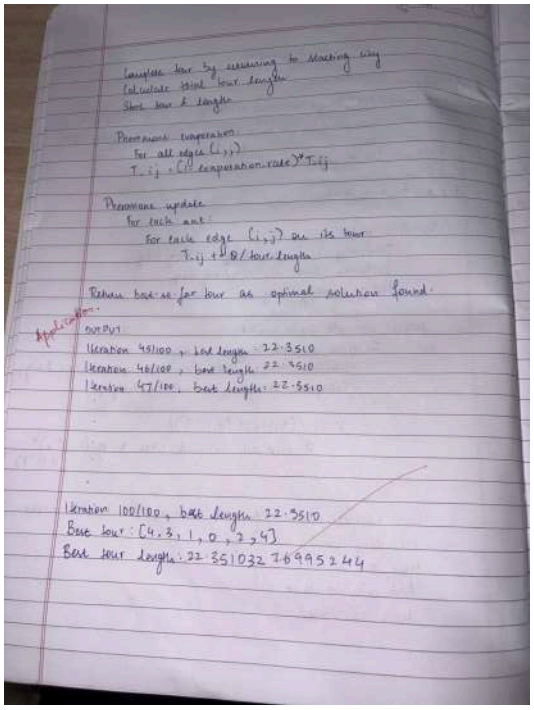
Program 3

Ant Colony Optimization for the Traveling Salesman Problem:
The foraging behavior of ants has inspired the development of optimization algorithms that can solve complex problems such as the Traveling Salesman Problem (TSP). Ant Colony Optimization (ACO) simulates the way ants
find the shortest path between food sources and their nest. Implement the ACO algorithm using Python to solve
the TSP, where the objective is to find the shortest possible route that visits a list of cities and returns to the origin city.

Algorithm:



```
ANT COLONY OPTIMIZATION

Initialize
 - pheromone[i][j] for all edges (i,j) to a small constant To
 - parameters alpha(α), beta(β), evaporation rate, Q (pheromone deposit constant)
 - cities = list of coordinates or nodes

Repeat for max_iterations

  For each ant:
     - Randomly choose a ~~[illegible]~~ starting city
     - Initialize tour w/ starting city
     - Mark starting city as visited

  While not all cities visited:
      For each unvisited city j:
         - Compute n_ij = 1 / distance b/w current city i & city j
         - Compute probability to move to city j:

           P_ij = (τ_ij^α) * (n_ij^β) /
               Σ over all unvisited cities k of (τ_ik^α) *
                                                  (n_ik^β)

  Use roulette wheel selection (weighted random) to choose
  next city based on P_ij
  Move to selected city
  Add city to tour
  Mark as visited
```



```
  Complete tour by returning to starting city
  Calculate total tour length
  Store tour & length

  Pheromone evaporation:
    For all edges (i,j)
    T_ij = (1 - evaporation rate) * T_ij

  Pheromone update
    For each ant:
      For each edge (i,j) in its tour
         T_ij += Q / tour length

Return best-so-far tour as optimal solution found.
```

Application:-

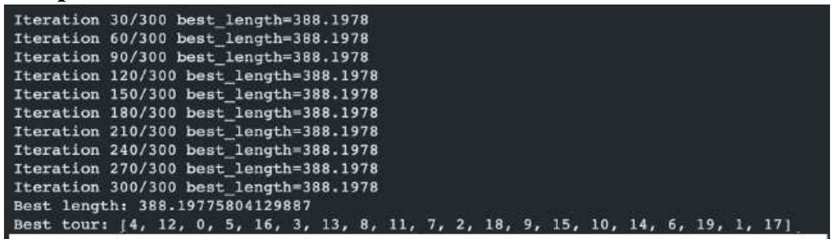
```
OUTPUT
Iteration 45/100, best length = 22.3510
Iteration 46/100, best length 22.3510
Iteration 47/100, best length: 22.3510

Iteration 100/100, best length 22.3510
Best tour: [4, 3, 1, 0, 2, 4]
Best tour length: 22.35103276995244
```

Code:

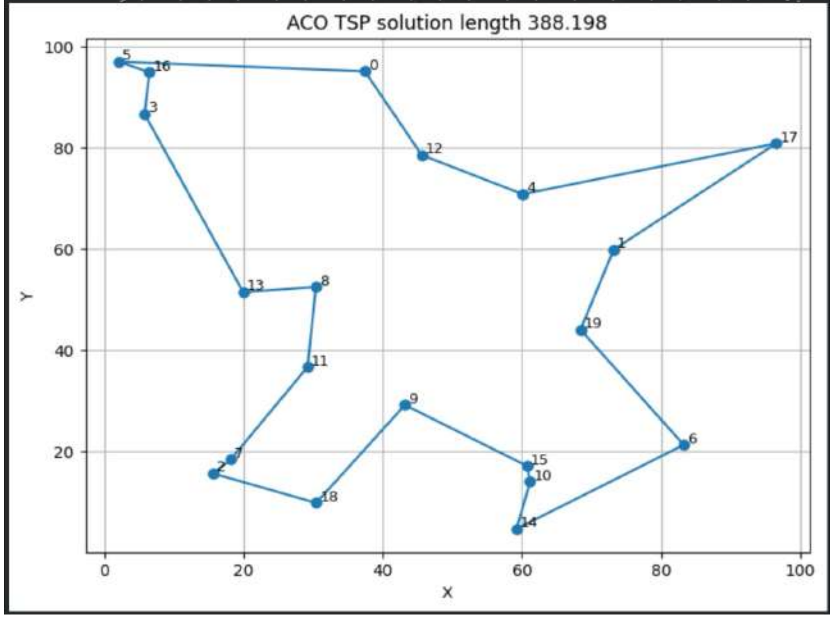
```
import random
import numpy as np

class AntColonyTSP:
    def __init__(
        self,
        coords,
        n_ants=20,
        n_iterations=200,
        alpha=1.0,
        beta=5.0,
        rho=0.5,
        initial_pheromone=1.0,
        q=100.0,
        seed=None
    ):
        if seed is not None:
            random.seed(seed)
            np.random.seed(seed)

        self.coords = np.asarray(coords)
        self.n_cities = len(self.coords)
        self.dist = self._distance_matrix(self.coords)
        self.heuristic = 1.0 / (self.dist + 1e-12)
        np.fill_diagonal(self.heuristic, 0.0)

        self.n_ants = n_ants
        self.n_iterations = n_iterations
        self.alpha = alpha
        self.beta = beta
        self.rho = rho
        self.q = q

        self.pheromone = np.full((self.n_cities, self.n_cities), initial_pheromone, dtype=float)
        self.best_tour = None
        self.best_length = float("inf")

    @staticmethod
    def _distance_matrix(coords):
        n = len(coords)
        D = np.zeros((n, n))
        for i in range(n):
            for j in range(i + 1, n):
                d = np.linalg.norm(coords[i] - coords[j])
                D[i, j] = d
                D[j, i] = d
        return D

    def _tour_length(self, tour):
        L = 0.0
        for i in range(len(tour) - 1):
            L += self.dist[tour[i], tour[i + 1]]
        L += self.dist[tour[-1], tour[0]]
        return L

    def _transition_probabilities(self, current, unvisited):
        pher = self.pheromone[current, unvisited] ** self.alpha
        heur = self.heuristic[current, unvisited] ** self.beta
        num = pher * heur
        s = num.sum()
        if s == 0:
            return np.ones(len(unvisited)) / len(unvisited)
        return num / s

    def _construct_solutions(self):
        tours = []
        lengths = []
        for _ in range(self.n_ants):
            start = random.randrange(self.n_cities)
            tour = [start]

            unvisited = list(range(self.n_cities))
            unvisited.remove(start)

            while unvisited:
                current = tour[-1]
                unvisited_arr = np.array(unvisited, dtype=int)
                probs = self._transition_probabilities(current, unvisited_arr)
                chosen_idx = np.random.choice(len(unvisited), p=probs)
                next_city = unvisited.pop(chosen_idx)
                tour.append(next_city)

            L = self._tour_length(tour)
            tours.append(tour)
            lengths.append(L)

            if L < self.best_length:
                self.best_length = L
                self.best_tour = tour.copy()

        return tours, lengths

    def _update_pheromones(self, tours, lengths):
        self.pheromone *= (1.0 - self.rho)

        for tour, L in zip(tours, lengths):
            deposit = self.q / (L + 1e-12)
            for i in range(len(tour)):
                a = tour[i]
                b = tour[(i + 1) % self.n_cities]
                self.pheromone[a, b] += deposit
                self.pheromone[b, a] += deposit

    def run(self, verbose=False):
        for it in range(1, self.n_iterations + 1):
            tours, lengths = self._construct_solutions()
            self._update_pheromones(tours, lengths)

            if verbose and (it % max(1, self.n_iterations // 10) == 0):
                print(f"Iteration {it}/{self.n_iterations} best_length={self.best_length:.4f}")

        return self.best_tour, self.best_length


if __name__ == "__main__":
```

```
import matplotlib.pyplot as plt

n_cities = 20
seed = 42
np.random.seed(seed)
coords = np.random.rand(n_cities, 2) * 100

aco = AntColonyTSP(
    coords,
    n_ants=40,
    n_iterations=300,
    alpha=1.0,
    beta=5.0,
    rho=0.4,
    initial_pheromone=1.0,
    q=100.0,
    seed=seed
)

best_tour, best_len = aco.run(verbose=True)
print("Best length:", best_len)
print("Best tour:", best_tour)

tour_coords = coords[[*best_tour, best_tour[0]]]

plt.figure(figsize=(8, 6))
plt.scatter(coords[:, 0], coords[:, 1])

for i, (x, y) in enumerate(coords):
    plt.text(x + 0.5, y + 0.5, str(i), fontsize=9)

plt.plot(tour_coords[:, 0], tour_coords[:, 1], marker='o', linestyle='-')
plt.title(f"ACO TSP solution length {best_len:.3f}")
plt.xlabel("X")
plt.ylabel("Y")
plt.grid(True)
plt.show()
```

Output:

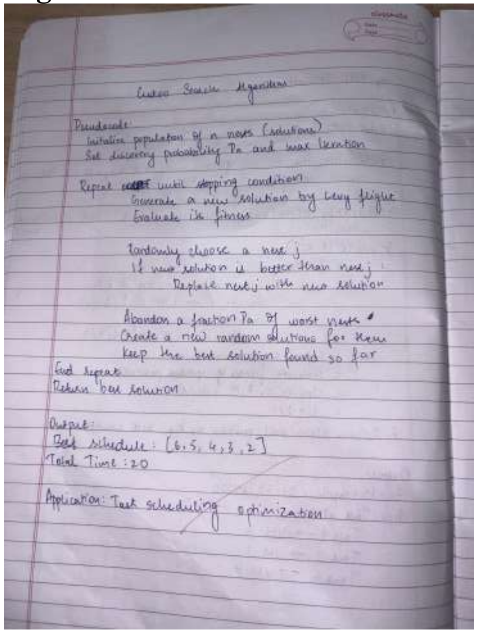


```
Iteration 30/300 best_length=388.1978
Iteration 60/300 best_length=388.1978
Iteration 90/300 best_length=388.1978
Iteration 120/300 best_length=388.1978
Iteration 150/300 best_length=388.1978
Iteration 180/300 best_length=388.1978
Iteration 210/300 best_length=388.1978
Iteration 240/300 best_length=388.1978
Iteration 270/300 best_length=388.1978
Iteration 300/300 best_length=388.1978
Best length: 388.19775804129887
Best tour: [4, 12, 0, 5, 16, 3, 13, 8, 11, 7, 2, 18, 9, 15, 10, 14, 6, 19, 1, 17]
```



Program 4
Cuckoo Search (CS):
Cuckoo Search (CS) is a nature-inspired optimization algorithm based on the brood parasitism of some cuckoo species. This behavior involves laying eggs in the nests of other birds, leading to the optimization of survival strategies. CS uses Lévy flights to generate new solutions, promoting global search capabilities and avoiding local
minima. The algorithm is widely used for solving continuous optimization problems and has applications in various domains, including engineering design, machine learning, and data mining.

Algorithm:



Cuckoo Search Algorithm

```
Pseudocode:
Initialize population of n nests (solutions)
Set discovery probability Pa and max iteration

Repeat until stopping condition
    Generate a new solution by Levy flight
    Evaluate its fitness

    randomly choose a nest j
    If new solution is better than nest j
        Replace nest j with new solution

    Abandon a fraction Pa of worst nests
    Create a new random solutions for them
    Keep the best solution found so far
End repeat
Return best solution
```

Output:
Best schedule: [6, 5, 4, 3, 2]
Total Time: 20

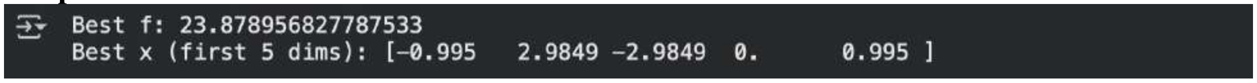
Application: Task scheduling optimization

Code:

```
import numpy as np
import math


def _levy_flight(shape, beta=1.5, rng=None):
    """
    Generate Lévy flight steps using Mantegna's algorithm.
    Returns an array with given shape.
    """
    if rng is None:
        rng = np.random.default_rng()

    # Mantegna parameters
    sigma_u = (
        math.gamma(1 + beta) * math.sin(math.pi * beta / 2)
        / (math.gamma((1 + beta) / 2) * beta * 2 ** ((beta - 1) / 2))
    ) ** (1 / beta)

    u = rng.normal(0, sigma_u, size=shape)
    v = rng.normal(0, 1, size=shape)
    step = u / (np.abs(v) ** (1 / beta))
    return step


def cuckoo_search(
    objective,
    bounds,
    n_nests=25,
    n_iter=250,
    pa=0.25,            # discovery/abandonment probability
    alpha=0.01,         # step size coefficient for Lévy flights
    beta=1.5,           # Lévy distribution exponent
    seed=None,
    return_history=True,
):
    """
    Cuckoo Search (CS) metaheuristic for bounded minimization.
    """
    rng = np.random.default_rng(seed)

    bounds = np.array(bounds, dtype=float)
    lb, ub = bounds[:, 0], bounds[:, 1]
    assert np.all(ub > lb), "Each upper bound must be greater than lower bound."
    d = len(bounds)

    def clip(X):
        return np.clip(X, lb, ub)

    # Initialize nests uniformly
    nests = rng.uniform(lb, ub, size=(n_nests, d))
    fitness = np.apply_along_axis(objective, 1, nests)

    best_idx = int(np.argmin(fitness))
    best = nests[best_idx].copy()
    best_f = float(fitness[best_idx])

    history = np.empty(n_iter, dtype=float) if return_history else None
    scale = (ub - lb)
    step_scale = alpha * scale

    for t in range(n_iter):
        # Lévy flight step
        steps = _levy_flight((n_nests, d), beta=beta, rng=rng) * step_scale
        cuckoos = nests + steps * (nests - best)
        cuckoos = clip(cuckoos)

        cuckoos_fit = np.apply_along_axis(objective, 1, cuckoos)

        # Replace some nests
        rand_idx = rng.integers(0, n_nests, size=n_nests)
        replace_mask = cuckoos_fit < fitness[rand_idx]
        nests[rand_idx[replace_mask]] = cuckoos[replace_mask]
        fitness[rand_idx[replace_mask]] = cuckoos_fit[replace_mask]
        # Abandon worst nests
        n_abandon = max(1, int(pa * n_nests))
        worst_idx = np.argsort(fitness)[-n_abandon:]
        i_idx = rng.integers(0, n_nests, size=n_abandon)
        j_idx = rng.integers(0, n_nests, size=n_abandon)
        eps = rng.random((n_abandon, d))
        new_nests = nests[worst_idx] + eps * (nests[i_idx] - nests[j_idx])
        new_nests += 0.001 * rng.normal(size=new_nests.shape) * scale
        new_nests = clip(new_nests)

        new_fit = np.apply_along_axis(objective, 1, new_nests)
        better_mask = new_fit < fitness[worst_idx]
        nests[worst_idx[better_mask]] = new_nests[better_mask]
        fitness[worst_idx[better_mask]] = new_fit[better_mask]

        # Update best
        curr_idx = int(np.argmin(fitness))
        curr_best_f = float(fitness[curr_idx])
        if curr_best_f < best_f:
            best_f = curr_best_f
            best = nests[curr_idx].copy()

        if return_history:
            history[t] = best_f

    return (best, best_f, history) if return_history else (best, best_f)


# ------------------------- #
# Example usage
# ------------------------- #
if __name__ == "__main__":
    # Rastrigin test function
    def rastrigin(x):
        A = 10.0
        return A * x.size + np.sum(x**2 - A * np.cos(2 * np.pi * x))

    dim = 10
    bounds = [(-5.12, 5.12)] * dim

    best_x, best_f, hist = cuckoo_search(
        rastrigin,
        bounds,
        n_nests=30,
        n_iter=500,
        pa=0.25,
        alpha=0.05,
        beta=1.5,
        seed=42,
        return_history=True,
    )
```

```
print("Best f:", best_f)
print("Best x (first 5 dims):", np.round(best_x[:5], 4))
```

Output:

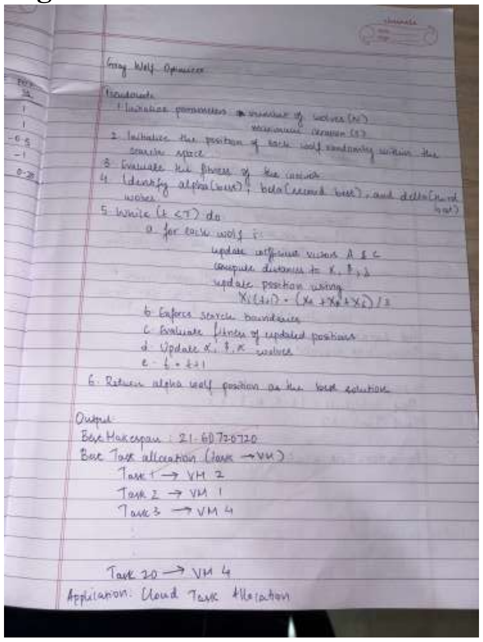


```
Best f: 23.878956827787533
Best x (first 5 dims): [-0.995   2.9849 -2.9849  0.      0.995 ]
```

Program 5

Grey Wolf Optimizer (GWO):
The Grey Wolf Optimizer (GWO) algorithm is a swarm intelligence algorithm inspired by the social hierarchy and
hunting behavior of grey wolves. It mimics the leadership structure of alpha, beta, delta, and omega wolves and
their collaborative hunting strategies. The GWO algorithm uses these social hierarchies to model the optimization process, where the alpha wolves guide the search process while beta and delta wolves assist in refining the search direction. This algorithm is effective for continuous optimization problems and has applications in engineering, data analysis, and machine learning.

Algorithm:



Grey Wolf Optimizer

Pseudocode

```
1 Initialize parameters: number of wolves (N)
                         maximum iteration (T)
2 Initialize the position of each wolf randomly within the
  search space
3 Evaluate the fitness of the wolves
4 Identify alpha(best), beta(second best), and delta(third
  wolves                                             best)
5 While (t < T) do
    a for each wolf i:
          update coefficient vectors A & C
          compute distances to α, β, δ
          update position using
              $X_i(t+1) = (X_\alpha + X_\beta + X_\delta)/3$
    b Enforce search boundaries
    c Evaluate fitness of updated positions
    d Update α, β, δ wolves
    e t = t+1
6 Return alpha wolf position as the best solution
```

Output:

```
Best Makespan : 21.6072072O
Best Task allocation (Task → VM)
    Task 1 → VM 2
    Task 2 → VM 1
    Task 3 → VM 4

    Task 20 → VM 4
```

Application: Cloud Task Allocation

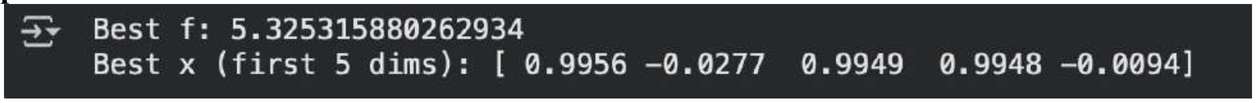
Code:

```
import numpy as np

def grey_wolf_optimization(
    objective,
    bounds,
    n_wolves=25,
    n_iter=250,
    seed=None,
    return_history=True,
):
    rng = np.random.default_rng(seed)

    bounds = np.array(bounds, dtype=float)
    lb, ub = bounds[:, 0], bounds[:, 1]
    d = len(bounds)

    X = rng.uniform(lb, ub, size=(n_wolves, d))
```

```
fitness = np.apply_along_axis(objective, 1, X)

def top3(X_arr, fit_arr):
    idx = np.argsort(fit_arr)
    return (
        X_arr[idx[0]].copy(),
        float(fit_arr[idx[0]]),
        X_arr[idx[1]].copy(),
        float(fit_arr[idx[1]]),
        X_arr[idx[2]].copy(),
        float(fit_arr[idx[2]]),
    )

X_alpha, f_alpha, X_beta, f_beta, X_delta, f_delta = top3(X, fitness)

history = np.empty(n_iter, dtype=float) if return_history else None

for t in range(n_iter):
    a = 2.0 - 2.0 * (t / (n_iter - 1 if n_iter > 1 else 1))

    r1 = rng.random((n_wolves, d))
    r2 = rng.random((n_wolves, d))
    A1 = 2 * a * r1 - a
    C1 = 2 * r2

    r1 = rng.random((n_wolves, d))
    r2 = rng.random((n_wolves, d))
    A2 = 2 * a * r1 - a
    C2 = 2 * r2

    r1 = rng.random((n_wolves, d))
    r2 = rng.random((n_wolves, d))
    A3 = 2 * a * r1 - a
    C3 = 2 * r2

    D_alpha = np.abs(C1 * X_alpha - X)
    D_beta  = np.abs(C2 * X_beta  - X)
    D_delta = np.abs(C3 * X_delta - X)

    X1 = X_alpha - A1 * D_alpha
    X2 = X_beta  - A2 * D_beta
    X3 = X_delta - A3 * D_delta

    X_new = (X1 + X2 + X3) / 3.0
    X_new = np.clip(X_new, lb, ub)

    f_new = np.apply_along_axis(objective, 1, X_new)
```

```
        replace = f_new < fitness
        if np.any(replace):
            X[replace] = X_new[replace]
            fitness[replace] = f_new[replace]

        X_alpha, f_alpha, X_beta, f_beta, X_delta, f_delta = top3(X, fitness)

        if return_history:
            history[t] = f_alpha

    return (X_alpha, f_alpha, history) if return_history else (X_alpha, f_alpha)


if __name__ == "__main__":
    def rastrigin(x):
        A = 10.0
        return A * x.size + np.sum(x**2 - A * np.cos(2 * np.pi * x))

    dim = 10
    bounds = [(-5.12, 5.12)] * dim

    best_x, best_f, hist = grey_wolf_optimization(
        rastrigin,
        bounds,
        n_wolves=30,
        n_iter=500,
        seed=42,
        return_history=True,
    )

    print("Best f:", best_f)
    print("Best x (first 5 dims):", np.round(best_x[:5], 4))
```

Output:



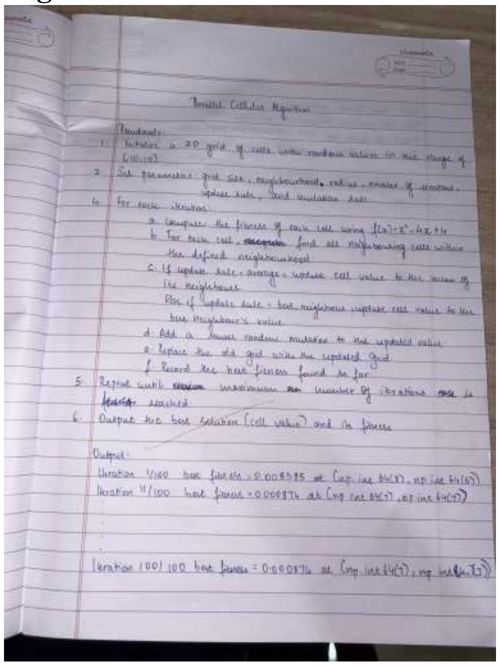
```
Best f: 5.325315880262934
Best x (first 5 dims): [ 0.9956 -0.0277  0.9949  0.9948 -0.0094]
```

Program 6

Parallel Cellular Algorithms and Programs:

Parallel Cellular Algorithms are inspired by the functioning of biological cells that operate in a highly parallel and
distributed manner. These algorithms leverage the principles of cellular automata and parallel computing to solve complex optimization problems efficiently. Each cell represents a potential solution and interacts with its neighbors to update its state based on predefined rules. This interaction models the diffusion of information across the cellular grid, enabling the algorithm to explore the search space effectively. Parallel Cellular Algorithms are particularly suitable for large-scale optimization problems and can be implemented on parallel computing architectures for enhanced performance.

Algorithm:



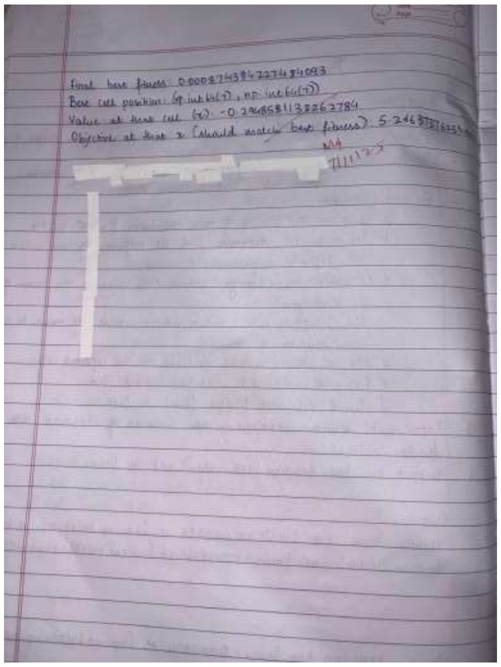
Parallel Cellular Algorithm

Pseudocode:

1. Initialize a 2D grid of cells with random values in the range of [-10,10]
2. Set parameters: grid size, neighbourhood radius, number of iterations, update rule, and mutation rate
4. For each iteration:
    a. Compute the fitness of each cell using $f(x)=x^2-4x+4$
    b. For each cell, ~~[illegible]~~ find all neighbouring cells within the defined neighbourhood
    c. If update rule = average: update cell value to the mean of the neighbours
       Else if update rule = best neighbour: update cell value to the best neighbour's value
    d. Add a small random mutation to the updated value
    e. Replace the old grid with the updated grid
    f. Record the best fitness found so far
5. Repeat until ~~[illegible]~~ maximum ~~[illegible]~~ number of iterations ~~[illegible]~~ is ~~[illegible]~~ reached
6. Output the best solution (cell value) and its fitness

Output:

```
Iteration 1/100 best fitness = 0.008585 at (np.int64(7), np.int64(6))
Iteration 11/100 best fitness = 0.000874 at (np.int64(7), np.int64(7))
.
.
.
Iteration 100/100 best fitness = 0.000874 at (np.int64(7), np.int64(7))
```



```
Final best fitness: 0.000874394227454093
Best cell position: (np.int64(7), np.int64(7))
Value at best cell (x): -0.29485811352627 84
Objective at that x (should match best fitness): 5.2463737655
```

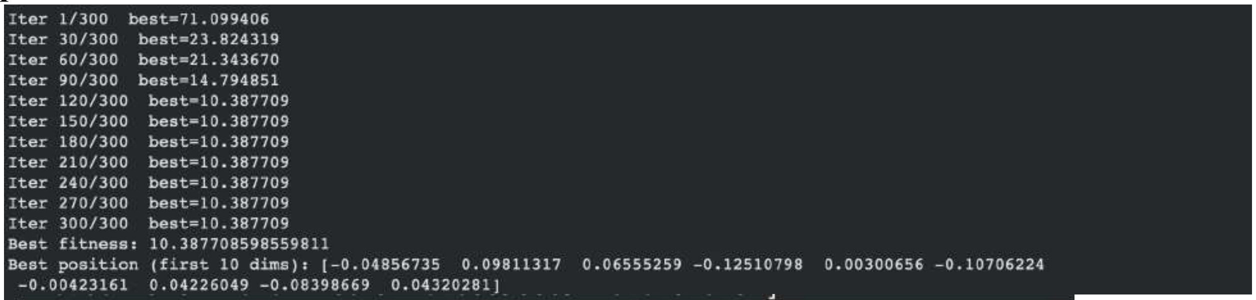
7/11/25

Code:

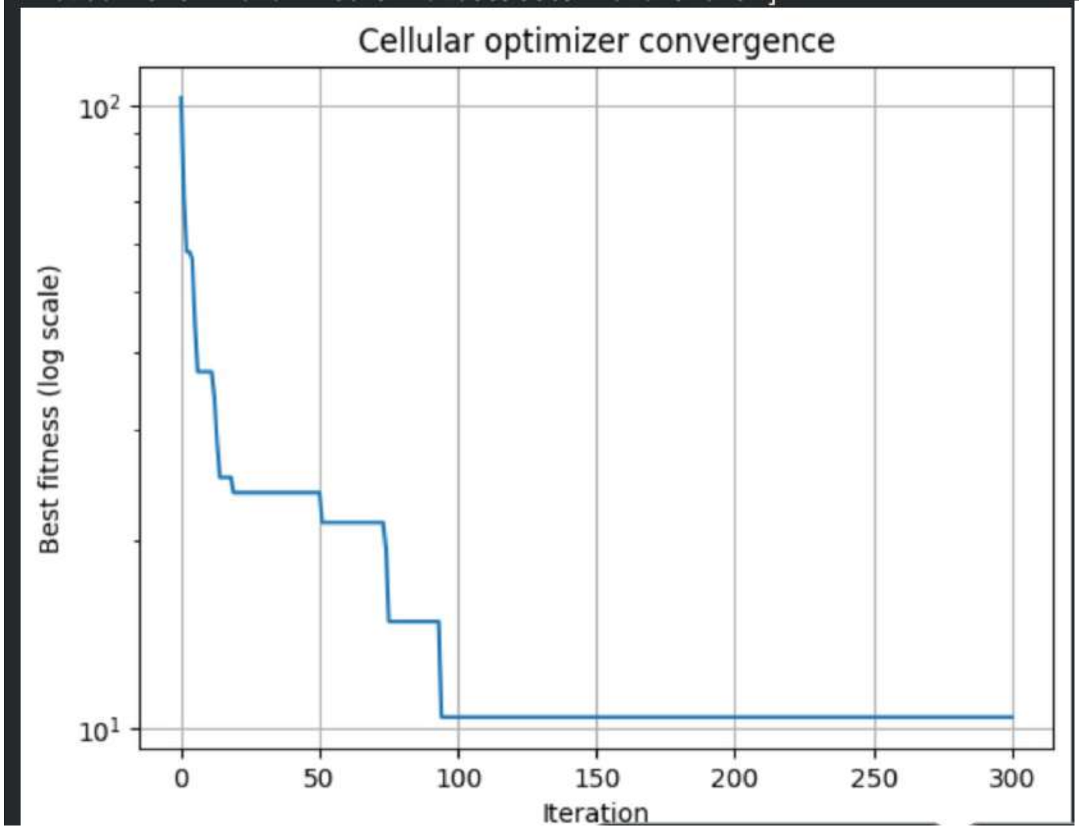
```
import numpy as np
import math
import random
from multiprocessing import Pool, cpu_count


def rastrigin(x):
    x = np.asarray(x)
    n = x.size
    return 10 * n + ((x**2 - 10 * np.cos(2 * math.pi * x)).sum())


class CellularOptimizer:
    def __init__(
        self,
        obj_fn,
        dim,
        grid_shape=(20, 20),
        bounds=(-5.12, 5.12),
        init_scale=1.0,
        neighborhood="moore",
        move_rate=0.5,
        mutation_std=0.1,
        n_iters=200,
        use_parallel=False,
        seed=None,
    ):
        if seed is not None:
            np.random.seed(seed)
            random.seed(seed)

        self.obj_fn = obj_fn
        self.dim = dim
        self.grid_shape = grid_shape
        self.n_cells = grid_shape[0] * grid_shape[1]
        self.bounds = np.array(bounds, dtype=float)
        self.init_scale = init_scale
        self.move_rate = move_rate
        self.mutation_std = mutation_std
        self.n_iters = n_iters
        self.neighborhood = neighborhood
        self.use_parallel = use_parallel

        low, high = self.bounds
        self.positions = np.random.uniform(low, high, size=(grid_shape[0], grid_shape[1], dim)) * init_scale
        self.fitness = np.full((grid_shape[0], grid_shape[1]), np.inf, dtype=float)
        self.best_pos = None
        self.best_fit = np.inf

    def _get_neighbors_idx(self, i, j):
        rows, cols = self.grid_shape
        neighbors = []
        for di in (-1, 0, 1):
            for dj in (-1, 0, 1):
                ni = (i + di) % rows
                nj = (j + dj) % cols
                neighbors.append((ni, nj))
        return neighbors

    def _evaluate_one(self, pos):
        return self.obj_fn(pos)

    def evaluate_fitness(self):
        flat_positions = self.positions.reshape((-1, self.dim))
        if self.use_parallel:
            with Pool(min(cpu_count(), 8)) as p:
                flat_f = p.map(self._evaluate_one, list(flat_positions))
            flat_f = np.asarray(flat_f, dtype=float)
        else:
            flat_f = np.asarray([self._evaluate_one(x) for x in flat_positions], dtype=float)

        self.fitness = flat_f.reshape(self.grid_shape)
```

```
        min_idx = np.unravel_index(np.argmin(self.fitness), self.grid_shape)
        if self.fitness[min_idx] < self.best_fit:
            self.best_fit = float(self.fitness[min_idx])
            self.best_pos = self.positions[min_idx].copy()

    def step(self):
        rows, cols = self.grid_shape
        new_positions = self.positions.copy()
        for i in range(rows):
            for j in range(cols):
                neighbors = self._get_neighbors_idx(i, j)
                best_n = min(neighbors, key=lambda ij: self.fitness[ij])
                best_pos = self.positions[best_n]
                curr_pos = self.positions[i, j]
                direction = best_pos - curr_pos
                new_pos = curr_pos + self.move_rate * direction
                new_pos = new_pos + np.random.normal(0, self.mutation_std, size=self.dim)
                new_pos = np.clip(new_pos, self.bounds[0], self.bounds[1])
                new_positions[i, j] = new_pos
        self.positions = new_positions

    def run(self, verbose=False, record_history=False):
        history = []
        self.evaluate_fitness()
        history.append(self.best_fit)
        for t in range(1, self.n_iters + 1):
            self.step()
            self.evaluate_fitness()
            history.append(self.best_fit)
            if verbose and (t % max(1, self.n_iters // 10) == 0 or t == 1):
                print(f"Iter {t}/{self.n_iters}  best={self.best_fit:.6f}")
        if record_history:
            return self.best_pos, self.best_fit, np.asarray(history)
        return self.best_pos, self.best_fit


if __name__ == "__main__":
    dim = 10
    grid_shape = (20, 20)
    opt = CellularOptimizer(
        obj_fn=rastrigin,
        dim=dim,
        grid_shape=grid_shape,
        bounds=(-5.12, 5.12),
        init_scale=1.0,
        move_rate=0.4,
        mutation_std=0.2,
        n_iters=300,
        use_parallel=False,
        seed=42,
    )
    best_pos, best_fit, history = opt.run(verbose=True, record_history=True)
    print("Best fitness:", best_fit)
print("Best position (first 10 dims):", best_pos[:10])

try:
    import matplotlib.pyplot as plt

    plt.plot(history)
    plt.yscale("log")
    plt.xlabel("Iteration")
    plt.ylabel("Best fitness (log scale)")
    plt.title("Cellular optimizer convergence")
    plt.grid(True)
    plt.show()
except Exception:
    pass
```

Output:



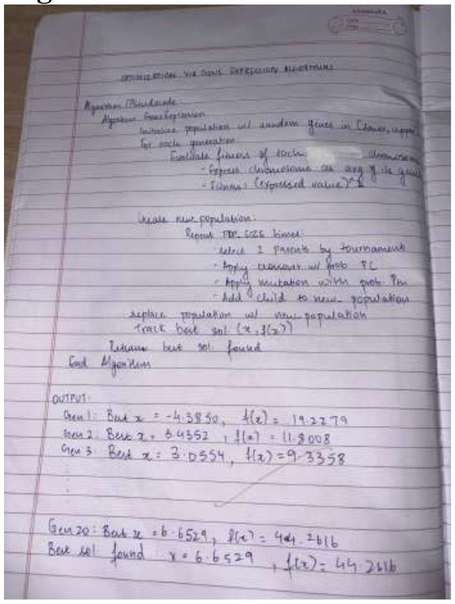
```
Iter 1/300  best=71.099406
Iter 30/300  best=23.824319
Iter 60/300  best=21.343670
Iter 90/300  best=14.794851
Iter 120/300  best=10.387709
Iter 150/300  best=10.387709
Iter 180/300  best=10.387709
Iter 210/300  best=10.387709
Iter 240/300  best=10.387709
Iter 270/300  best=10.387709
Iter 300/300  best=10.387709
Best fitness: 10.387708598559811
Best position (first 10 dims): [-0.04856735  0.09811317  0.06555259 -0.12510798  0.00300656 -0.10706224
 -0.00423161  0.04226049 -0.08398669  0.04320281]
```



Program 7

Optimization via Gene Expression Algorithms:
Gene Expression Algorithms (GEA) are inspired by the biological process of gene expression in living organisms.
This process involves the translation of genetic information encoded in DNA into functional proteins. In GEA, solutions to optimization problems are encoded in a manner similar to genetic sequences. The algorithm evolves
these solutions through selection, crossover, mutation, and gene expression to find optimal or near-optimal solutions. GEA is effective for solving complex optimization problems in various domains, including engineering,
data analysis, and machine learning.

Algorithm:



OPTIMIZATION VIA GENE EXPRESSION ALGORITHMS

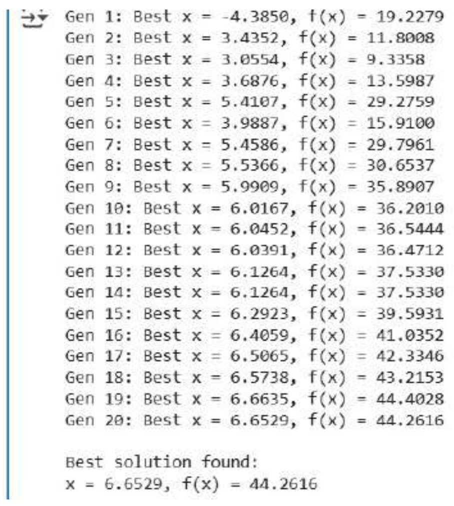
```
Algorithm (Pseudocode
  Algorithm GeneExpression
      Initialize population w/ random genes in [lower, upper]
      For each generation
          Evaluate fitness of each [illegible] chromosome
              - Express chromosome as avg of its genes
              - Fitness: (expressed value)^2

      Create new population:
          Repeat POP_SIZE times:
              - Select 2 parents by tournament
              - Apply crossover w/ prob PC
              - Apply mutation with prob Pm
              - Add child to new population
      replace population w/ new population
      Track best sol (x, f(x))
    Return best sol found
  End Algorithm
```

```
OUTPUT:
Gen 1: Best x = -4.3850, f(x) = 19.2279
Gen 2: Best x = 3.4352 , f(x) = 11.8008
Gen 3: Best x = 3.0554, f(x) = 9.3358


Gen 20: Best x = 6.6529, f(x) = 44.2616
Best sol found  x = 6.6529  , f(x) = 44.2616
```

Code:

```
import random

def objective_function(x):
    return x ** 2

POP_SIZE = 20
GENS = 20
GENE_LENGTH = 10
CROSSOVER_RATE = 0.8
MUTATION_RATE = 0.1
BOUNDS = [-10, 10]

def create_population():
    return [[random.uniform(BOUNDS[0], BOUNDS[1]) for _ in range(GENE_LENGTH)] for _ in
range(POP_SIZE)]

def gene_expression(gene):
    return sum(gene) / len(gene)

def evaluate(population):
    expressed = [gene_expression(g) for g in population]
    return [objective_function(x) for x in expressed], expressed

def select(population, fitness):
    i, j = random.sample(range(len(population)), 2)
    return population[i] if fitness[i] > fitness[j] else population[j]

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

def mutate(gene):
    return [g + random.uniform(-1, 1) if random.random() < MUTATION_RATE else g for g in gene]

def gene_expression_algorithm():
    population = create_population()
    for gen in range(GENS):
        fitness, expressed = evaluate(population)
        new_population = []
        for _ in range(POP_SIZE):
            parent1 = select(population, fitness)
            parent2 = select(population, fitness)
            child = crossover(parent1, parent2)
            child = mutate(child)
            new_population.append(child)
        population = new_population
        best_idx = fitness.index(max(fitness))
        best_x = expressed[best_idx]
        best_fit = fitness[best_idx]
        print(f"Gen {gen+1}: Best x = {best_x:.4f}, f(x) = {best_fit:.4f}")
    return best_x, best_fit

if __name__ == "__main__":
    best_x, best_val = gene_expression_algorithm()
    print("\nBest solution found:")
    print(f"x = {best_x:.4f}, f(x) = {best_val:.4f}")
```

Output:



```
Gen 1: Best x = -4.3850, f(x) = 19.2279
Gen 2: Best x = 3.4352, f(x) = 11.8008
Gen 3: Best x = 3.0554, f(x) = 9.3358
Gen 4: Best x = 3.6876, f(x) = 13.5987
Gen 5: Best x = 5.4107, f(x) = 29.2759
Gen 6: Best x = 3.9887, f(x) = 15.9100
Gen 7: Best x = 5.4586, f(x) = 29.7961
Gen 8: Best x = 5.5366, f(x) = 30.6537
Gen 9: Best x = 5.9909, f(x) = 35.8907
Gen 10: Best x = 6.0167, f(x) = 36.2010
Gen 11: Best x = 6.0452, f(x) = 36.5444
Gen 12: Best x = 6.0391, f(x) = 36.4712
Gen 13: Best x = 6.1264, f(x) = 37.5330
Gen 14: Best x = 6.1264, f(x) = 37.5330
Gen 15: Best x = 6.2923, f(x) = 39.5931
Gen 16: Best x = 6.4059, f(x) = 41.0352
Gen 17: Best x = 6.5065, f(x) = 42.3346
Gen 18: Best x = 6.5738, f(x) = 43.2153
Gen 19: Best x = 6.6635, f(x) = 44.4028
Gen 20: Best x = 6.6529, f(x) = 44.2616

Best solution found:
x = 6.6529, f(x) = 44.2616
```