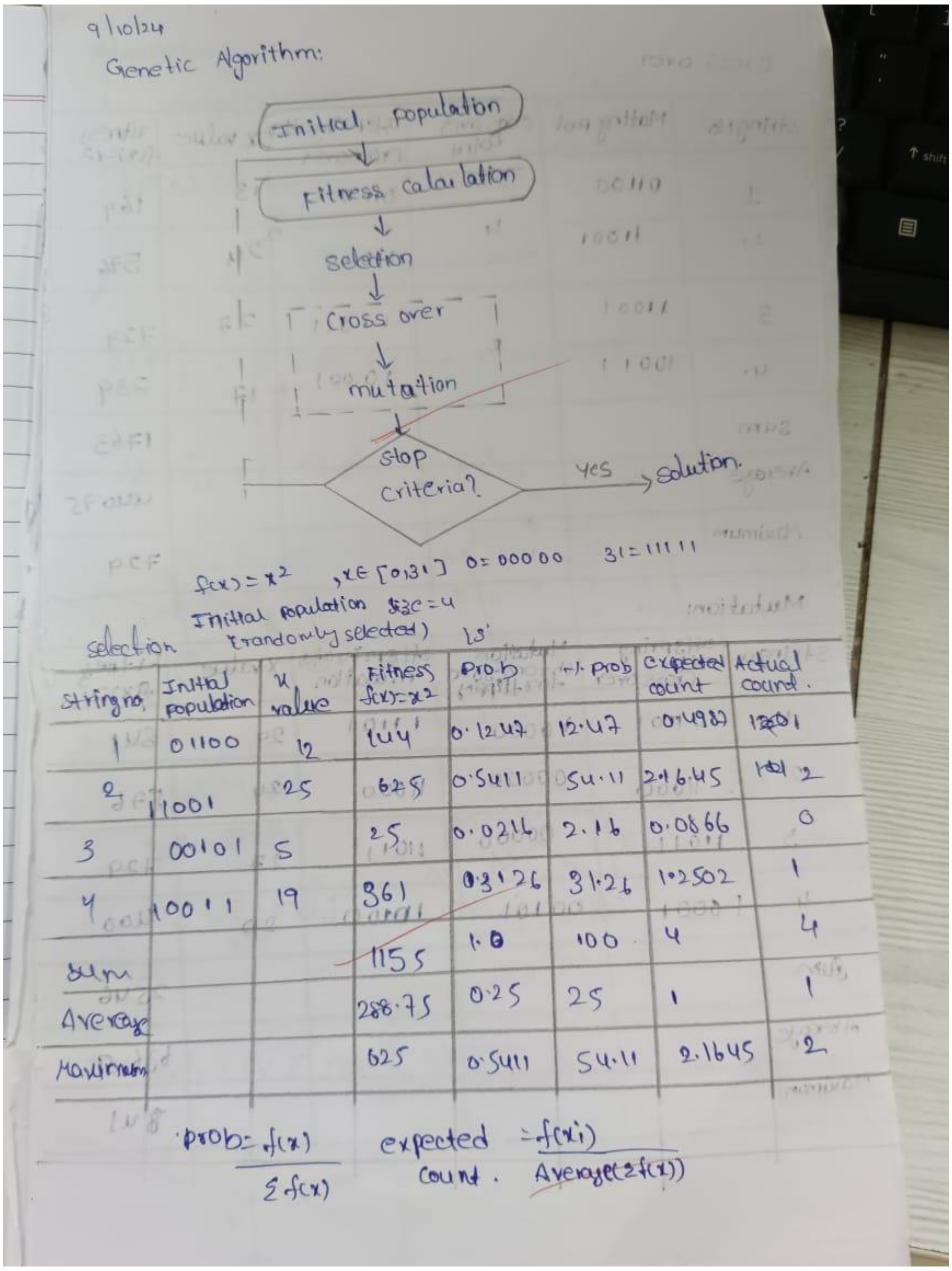
BIS LAB : 1

Program 1

Problem statement: Genetic Algorithm for Optimization Problems

Algorithm:



9/10/24

Genetic Algorithm:

Initial population → Fitness calculation → selection → Cross over → mutation → Stop criteria? → yes → solution.

$f(x) = x^2$, $x \in [0, 31]$ 0 = 00000 31 = 11111

Initial population size = 4 (randomly selected)

selection

String no.	Initial population	x value	Fitness $f(x)=x^2$	Prob	% prob	expected count	Actual count.
1	01100	12	144	0.1247	12.47	0.4987	1
2	11001	25	625	0.5411	54.11	2.1645	2
3	00101	5	25	0.0216	2.16	0.0866	0
4	10011	19	361	0.3126	31.26	1.2502	1
sum			1155	1.0	100	4	4
Average			288.75	0.25	25	1	1
Maximum			625	0.5411	54.11	2.1645	2

$\text{prob} = \frac{f(x)}{\Sigma f(x)}$ $\text{expected count} = \frac{f(x_i)}{\text{Average}(\Sigma f(x))}$

Cross over

String No.	Mating pool	Cross over point	offspring after crossover	x value	fitness f(x)=x²
1	01100	4	01101	13	169
2.	11001		11000	24	576
3	11001	2	11011	27	729
4.	10011		10001	17	289
sum					1763
Average					440·75
Maximum					729

Mutation:

String no.	offspring after cross over	Mutation chromosome for flipping	offspring after mutation	x value	Fitness f(x)=x²
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
Average					636·5
Maximum					841

Code:

```
import random
def fitness(x):
    return x**2
def create_population(pop_size):
    population = []
    for _ in range(pop_size):
        individual = random.randint(0, 31)
        population.append(individual)
    return population
def select_parents(population):
```

```
    population.sort(key=lambda x: fitness(x), reverse=True)
    return population[0], population[1]
def crossover(parent1, parent2):
    crossover_point = random.randint(0, 4)
    mask1 = parent1 >> crossover_point
    mask2 = parent2 & ((1 << crossover_point) - 1)
    child = (mask1 << crossover_point) | mask2
    return child
def mutate(individual, mutation_rate=0.3):
    if random.random() < mutation_rate:
        bit_to_flip = random.randint(0, 4)
        individual ^= (1 << bit_to_flip)
    return individual
def genetic_algorithm(pop_size, generations, mutation_rate):
    population = create_population(pop_size)
    for generation in range(generations):
        parent1, parent2 = select_parents(population)
        new_population = [parent1]

        for _ in range((pop_size - 1) // 2):
            child1 = crossover(parent1, parent2)
            child2 = crossover(parent2, parent1)
            child1 = mutate(child1, mutation_rate)
            child2 = mutate(child2, mutation_rate)
            new_population.append(child1)
            new_population.append(child2)
        population = new_population
        best_individual = max(population, key=lambda x: fitness(x))
        print(f"Generation {generation + 1}: Best individual = {best_individual}, Fitness = {fitness(best_individual)}")
    return best_individual
pop_size = 6
generations = 20
mutation_rate = 0.6
best = genetic_algorithm(pop_size, generations, mutation_rate)
print(f"Best solution found: x = {best}, f(x) = {fitness(best)}")
```

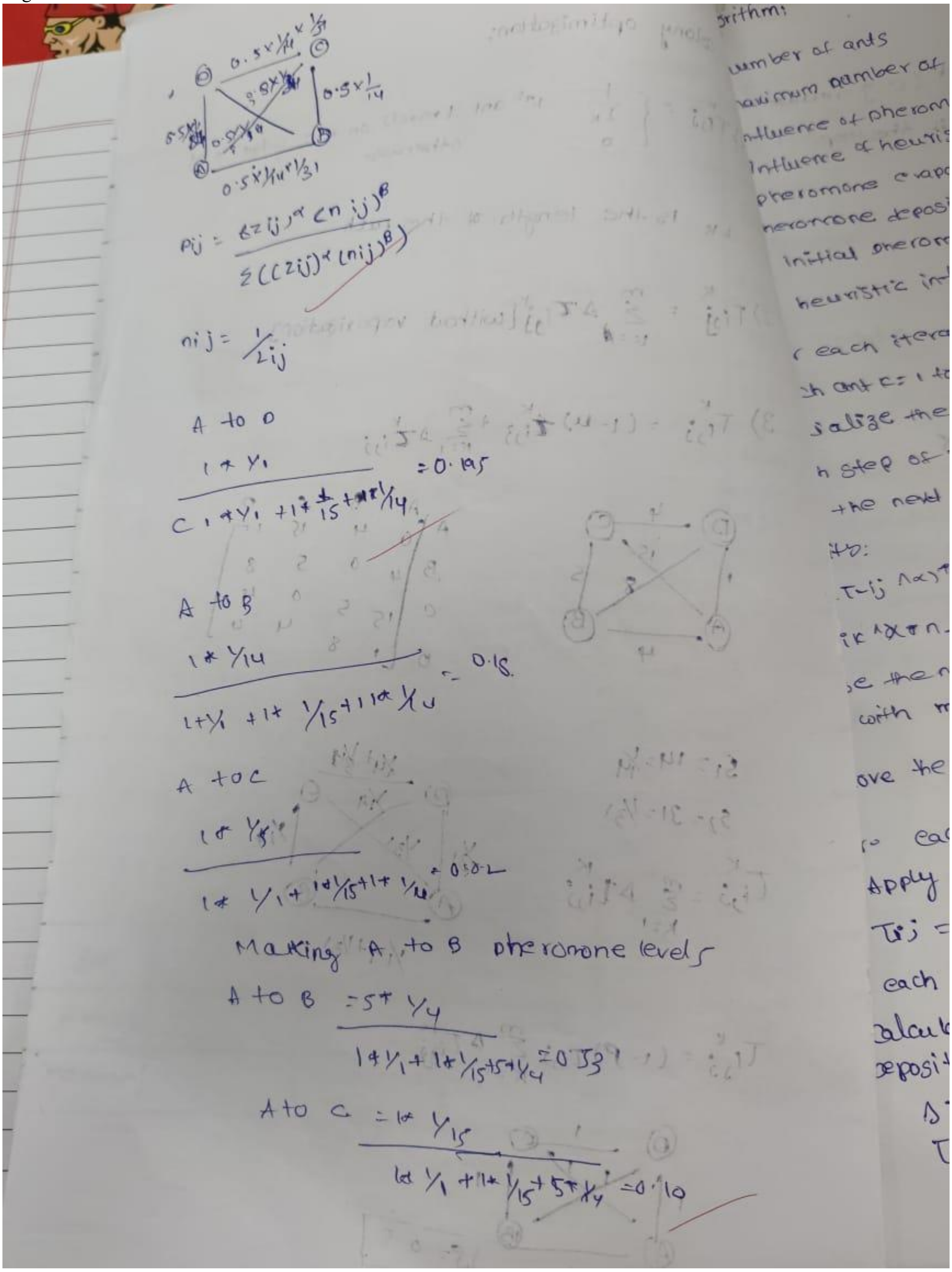
Output:

```
Generation 1: Best individual = 28, Fitness = 784
Generation 2: Best individual = 30, Fitness = 900
Generation 3: Best individual = 31, Fitness = 961
Generation 4: Best individual = 31, Fitness = 961
Generation 5: Best individual = 31, Fitness = 961
Generation 6: Best individual = 31, Fitness = 961
Generation 7: Best individual = 31, Fitness = 961
Generation 8: Best individual = 31, Fitness = 961
Generation 9: Best individual = 31, Fitness = 961
Generation 10: Best individual = 31, Fitness = 961
Generation 11: Best individual = 31, Fitness = 961
Generation 12: Best individual = 31, Fitness = 961
Generation 13: Best individual = 31, Fitness = 961
Generation 14: Best individual = 31, Fitness = 961
Generation 15: Best individual = 31, Fitness = 961
Generation 16: Best individual = 31, Fitness = 961
Generation 17: Best individual = 31, Fitness = 961
Generation 18: Best individual = 31, Fitness = 961
Generation 19: Best individual = 31, Fitness = 961
Generation 20: Best individual = 31, Fitness = 961
Best solution found: x = 31, f(x) = 961
```

LAB 2 WEEK:
Program 2

Problem statement: Ant Colony Optimization for the Traveling Salesman Problem

Algorithm:



$$p_{ij} = \frac{(\tau_{ij})^{\alpha}(\eta_{ij})^{\beta}}{\sum((\tau_{ij})^{\alpha}(\eta_{ij})^{\beta})}$$

$$\eta_{ij} = \frac{1}{\tau_{ij}}$$

A to D

$$\frac{1*1/1}{1*1/1 + 1*\frac{1}{15} + 1*1/14} = 0.195$$

A to B

$$\frac{1*1/14}{1+1/1 + 1*1/15 + 1*1/4} = 0.18$$

A to C

$$\frac{1*1/15}{1*1/1 + 1*1/15 + 1*1/4} = 0.032$$

Making A to B pheromone levels

$$A \text{ to } B = \frac{5*1/4}{1*1/1 + 1*1/15 + 5*1/4} = 0.539$$

$$A \text{ to } C = \frac{1*1/15}{1*1/1 + 1*1/15 + 5*1/4} = 0.10$$

...rithm:
...umber of ants
...aximum number of
...nfluence of pherom...
Influence of heuris...
pheromone evapo...
pheromone deposi...
Initial pherom...
heuristic in...
... each itera...
...h ant k = 1 to
...ialize the
...h step of
the next
...to:
...
...e the n...
with m...
...ove the
...r each
Apply
τ_{ij} =
each
calcul...
deposi...

Algorithm:

N = number of ants

T = maximum number of iterations.

α = Influence of pheromone

β = Influence of heuristic information.

ρ = pheromone evaporation rate

Q = pheromone deposit constant.

τij = initial pheromone level on all edges

ηij = heuristic information.

For each iteration t = 1 to T:

for each ant k = 1 to N:

Initialize the ant's tour

For each step of the ant's tour,

select the next node j, based on the transition probability:

P_ij = (τ_ij ^α) * (η_ij ^ β) / sum-over-allowed-nodes (τ_ik ^α * η_ik ^ β)

choose the next node j using probability P_ij, favoring paths with more pheromone and better heuristic info.

Move the ant to the select node.

For each edge (i, j):

Apply pheromone evaporation:

τij = (1-ρ) * τ_ij

For each ant K:

Calculate the tour length L_k

Deposit pheromone on the edges in the ant's path.

Δτij = Q/L_k

τ_ij = τ_ij = τ_ij + Δτ_ij for all edges (i, j) in ant k's tour.

optionally, keep track of the best solution found so far.

Check stopping criteria:

If stopping conditions is met (e.g., maximum iterations or convergence), exit the loop.

Return the best solution found.

Before updating the pheromone.

$$P_1 = \frac{1 * 1/1}{1 * 1/1 + 1 * 1/15 + 1 * 1/4} = 0.75$$

$$P_2 = \frac{1 * 1/4}{1 * 1/1 + 1 * 1/15 + 1 * 1/4} = 0.[illegible]5$$

$$P_3 = \frac{1 * 1/15}{1 * 1/1 + 1 * 1/15 + 1 * 1/4} = 0.05.$$

After updating the pheromone.

$$P_1 = \frac{1 * 1/1}{1 * 1/1 + 1 * 1/15 + 5 * 1/4} = 0.43$$

$$P_2 = \frac{5 * 1/4}{1 * 1/1 + 1 * 1/15 + 5 * 1/4} = 0.53$$

$$P_3 = \frac{1 * 1/15}{1 * 1/1 + 1 * 1/15 + 1 * 1/4} = 0.02.$$

Code:

```python
import numpy as np
import random
class AntColony:
    def __init__(self, graph, n_ants, n_best, n_iterations, decay, alpha=1, beta=1):
        self.graph = graph
        self.n_ants = n_ants
```

```
        self.n_best = n_best
        self.n_iterations = n_iterations
        self.decay = decay
        self.alpha = alpha
        self.beta = beta
        self.pheromone = np.ones(self.graph.shape) / len(graph)
        self.nodes = ['A', 'B', 'C', 'D']
    def run(self):
        shortest_path = None
        shortest_distance = float('inf')
        for _ in range(self.n_iterations):
            all_paths = self.gen_all_paths()
            self.spread_pheronome(all_paths, shortest_path, shortest_distance)
            shortest_path, shortest_distance = self.best_path(all_paths)
        shortest_path = [(self.nodes[from_node], self.nodes[to_node]) for from_node, to_node in shortest_path]
        return shortest_path, shortest_distance
    def spread_pheronome(self, all_paths, shortest_path, shortest_distance):
        for path, dist in all_paths:
            for from_node, to_node in path:
                self.pheromone[from_node][to_node] += 1.0 / dist
        self.pheromone *= self.decay
    def gen_path(self, start):
        path = []
        visited = set()
        visited.add(start)
        current = start
        while len(visited) < len(self.graph):
            next_node = self.pick_next_node(current, visited)
            path.append((current, next_node))
            visited.add(next_node)
            current = next_node
        path.append((current, start))
        return path
    def gen_all_paths(self):
        all_paths = []
        for ant in range(self.n_ants):
```

```
            path = self.gen_path(random.randint(0, len(self.graph)-1))
            distance = self.calculate_distance(path)
            all_paths.append((path, distance))
        return all_paths
    def calculate_distance(self, path):
        distance = 0
        for from_node, to_node in path:
            distance += self.graph[from_node][to_node]
        return distance
    def best_path(self, all_paths):
        best = min(all_paths, key=lambda x: x[1])
        return best
    def pick_next_node(self, current, visited):
        pheromone = np.copy(self.pheromone[current])
        pheromone[list(visited)] = 0
        attractiveness = np.copy(self.graph[current])
        attractiveness[list(visited)] = 0
        pheromone = pheromone ** self.alpha
        attractiveness = attractiveness ** self.beta
        prob = pheromone * attractiveness
        prob /= prob.sum()
        return np.random.choice(range(len(self.graph)), p=prob)
graph = np.array([
    [0, 10, 15, 20],
    [10, 0, 35, 25],
    [15, 35, 0, 30],
    [20, 25, 30, 0]
])
aco = AntColony(graph, n_ants=5, n_best=2, n_iterations=100, decay=0.95, alpha=1, beta=2)
shortest_path, shortest_distance = aco.run()
print("Shortest path (in terms of nodes): ", shortest_path)
print("Shortest distance: ", shortest_distance)
```

```
Shortest path:  [(1, 2), (2, 3), (3, 0), (0, 1)]
Shortest distance:  95
```

LAB 3 WEEK:
Program 3

Problem statement: Particle Swarm Optimization for Function Optimization

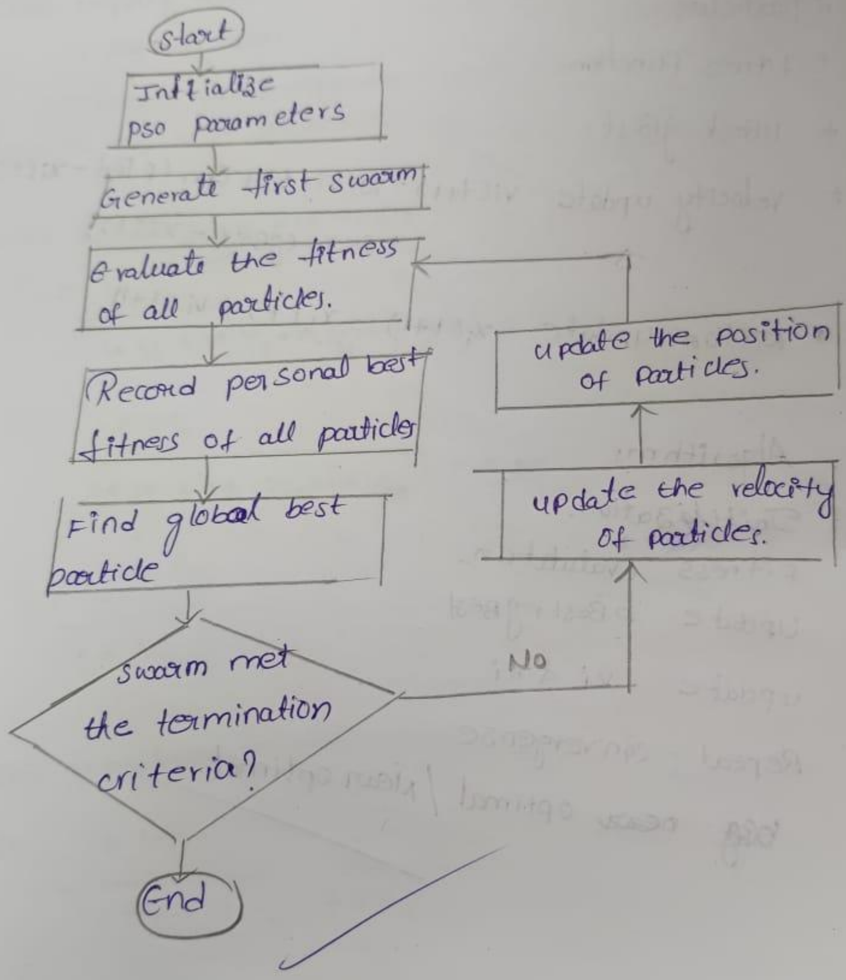
Algorithm:

13/11/24
particle swarm optimization (PSO)

* particles
* fitness Function.
* pBest gBest.
* velocity update $v_i(t+1) = w \cdot v_i(t) + c_1 r_1 (pBest - x_i(t)) + c_2 r_2 (gBest - x_i(t))$
* position update $x_i(t+1) = x_i(t) + v_i(t+1)$

Algorithm:
Initilization.
Fitness Evalutation.
Update pBest, gBest.
update v_i & x_i
Repeat · convergense
~~Beg~~ ~~near~~ optimal / Near optimal

Flow chart of Algorithm:



Algorithm steps:

1. create a 'population' of agents (particles) uni-formly distributed over x.

2. Evaluate each particle's position according to the objective function

$$y = F(x) = -x^2 + 5x + 20$$

3. If a particle's current position is better than its previous best position, update it.

4. Determine the best particle (according to the particle's previous best positions).

5. update particles' velocities:

$$v_i^{t+1} = \underbrace{v_i^t}_{\text{Inertia}} + \underbrace{c_1 U_1^t (pb_i^t - p_i^t)}_{\text{personal Influence}} + \underbrace{c_2 U_2^t (gb^t - p_i^t)}_{\text{social influence}}$$

6. Move particles to the their new positions:

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

7. Go to step 2 until stopping criteria are satisfied.

particle's velocity:

$v_i^{t+1} = \underbrace{v_i^t}_{\text{inertia}} + \underbrace{c_1 U_1^t (pb_i^t - p_i^t)}_{\text{personal influence}} + \underbrace{c_2 U_2^t (gb^t - p_i^t)}_{\text{social influence.}}$

$c_1 = 0 \quad c_2 = 0$

$v_{ij}^{t+1} = v_{ij}^t$ then all particles continue flying at their current speed.

$c_1 > 0 \quad c_2 = 0$ independent particles.

$v_i^{t+1} = v_j^t + c_1 u_i^t (pb_i^t - p_i^t)$

$c_1 = 0 \quad c_2 > 0$ all particles are attracted to a single point in the entric swarm.

$v_{ij}^{t+1} = v_{ij}^t + c_2 r_{2j}^t [gbest - x_{ij}^t]$

Psuedo code:

1. p= particle initilization();
2. for i=1 to max
3. for each particle p in P do:
   f(p)= f(p)
4. If p is better than f(pbest)
   pbest=p;
5. end

6. end
7. gbest = b
8. for each
9. v_i^{t+1} =
   in
10. p_i^{t+1}
11. end
12. end

output:

enter th
enter th
enter th
enter t
enter

iterati
partic
particle
partic

gobal

itera

partic
partic
parti
globa
o

6. end.

7. gbest = best p in p.

8. for each particle p in pcb

9. $v_i^{t+1} = \underbrace{v_i^t}_{\text{interia}} + \underbrace{c_1 v_1^t (pb_i^t - p_i^t)}_{\text{personal influence}} + \underbrace{c_2 v_2^t (g_b^t - p_i^t)}_{\text{social influence}}$

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

11. end

12. end.

output:

enter the no. of particles: 3

enter the no. of iteration: 2

enter the interia weight (Eg: 0.5): 1

enter the cognitive constant (Eg: 1.5): 1

enter the social constant (Eg: 1.5): 1

o/p verified 13/11/24

iteration 1: ~~position~~

particle 1: position: 1.2619 velocity = 1.8032 fitness = 24.7170

particle 2: position: 2.4424 velocity = 5.4452 fitness = 26.2462

particle 3: position: 3.8861 velocity = 0.000 fitness = 24.304

15

global best position: 24424 global fitness = 26.2462.

iteration 2:

particle 1: position = 3.8530 velocity = 2.5912 fitness = 24.4192

particle 2: position = 3.6052 velocity = 0.3806 fitness = 25.0278

particle 3: position = 2.4424 velocity = 0.3806 fitness = 25.0278

global best position = 2.4424 global fitness = 26.2462

optimal solution = position = 2.4424, value = 26.2462.

Code:

```
import numpy as np
def objective_function(x):
    return np.sum(x**2)
class ParticleSwarmOptimization:
    def __init__(self, objective_function, dim, num_particles, max_iter, w=0.5, c1=1, c2=2):
```

```
        self.objective_function = objective_function
        self.dim = dim
        self.num_particles = num_particles
        self.max_iter = max_iter
        self.w = w
        self.c1 = c1
        self.c2 = c2
        self.positions = np.random.uniform(-5, 5, (num_particles, dim))
        self.velocities = np.random.uniform(-1, 1, (num_particles, dim))
        self.personal_best_positions = np.copy(self.positions)
        self.personal_best_scores = np.array([objective_function(pos) for pos in self.positions])
        self.global_best_position = self.personal_best_positions[np.argmin(self.personal_best_scores)]
        self.global_best_score = np.min(self.personal_best_scores)
    def update_velocities(self):
        r1 = np.random.rand(self.num_particles, self.dim)
        r2 = np.random.rand(self.num_particles, self.dim)
        cognitive_component = self.c1 * r1 * (self.personal_best_positions - self.positions)
        social_component = self.c2 * r2 * (self.global_best_position - self.positions)
        self.velocities = self.w * self.velocities + cognitive_component + social_component
    def update_positions(self):
        self.positions = self.positions + self.velocities
        self.positions = np.clip(self.positions, -5, 5)
    def evaluate_particles(self):
        scores = np.array([self.objective_function(pos) for pos in self.positions])
        better_mask = scores < self.personal_best_scores
        self.personal_best_positions[better_mask] = self.positions[better_mask]
        self.personal_best_scores[better_mask] = scores[better_mask]
        best_particle = np.argmin(self.personal_best_scores)
        if self.personal_best_scores[best_particle] < self.global_best_score:
            self.global_best_position = self.personal_best_positions[best_particle]
            self.global_best_score = self.personal_best_scores[best_particle]
```

```python
    def run(self):
        for i in range(self.max_iter):
            self.update_velocities()
            self.update_positions()
            self.evaluate_particles()
            print(f"Iteration {i+1}/{self.max_iter}: Global Best Score = {self.global_best_score}")
        return self.global_best_position, self.global_best_score

dim = 2
num_particles = 30
max_iter = 10
w = 0.5
c1 = 1
c2 = 2
pso = ParticleSwarmOptimization(objective_function, dim, num_particles, max_iter, w, c1, c2)
best_position, best_score = pso.run()
print(f"\nBest Position: {best_position}")
print(f"Best Score: {best_score}")
```

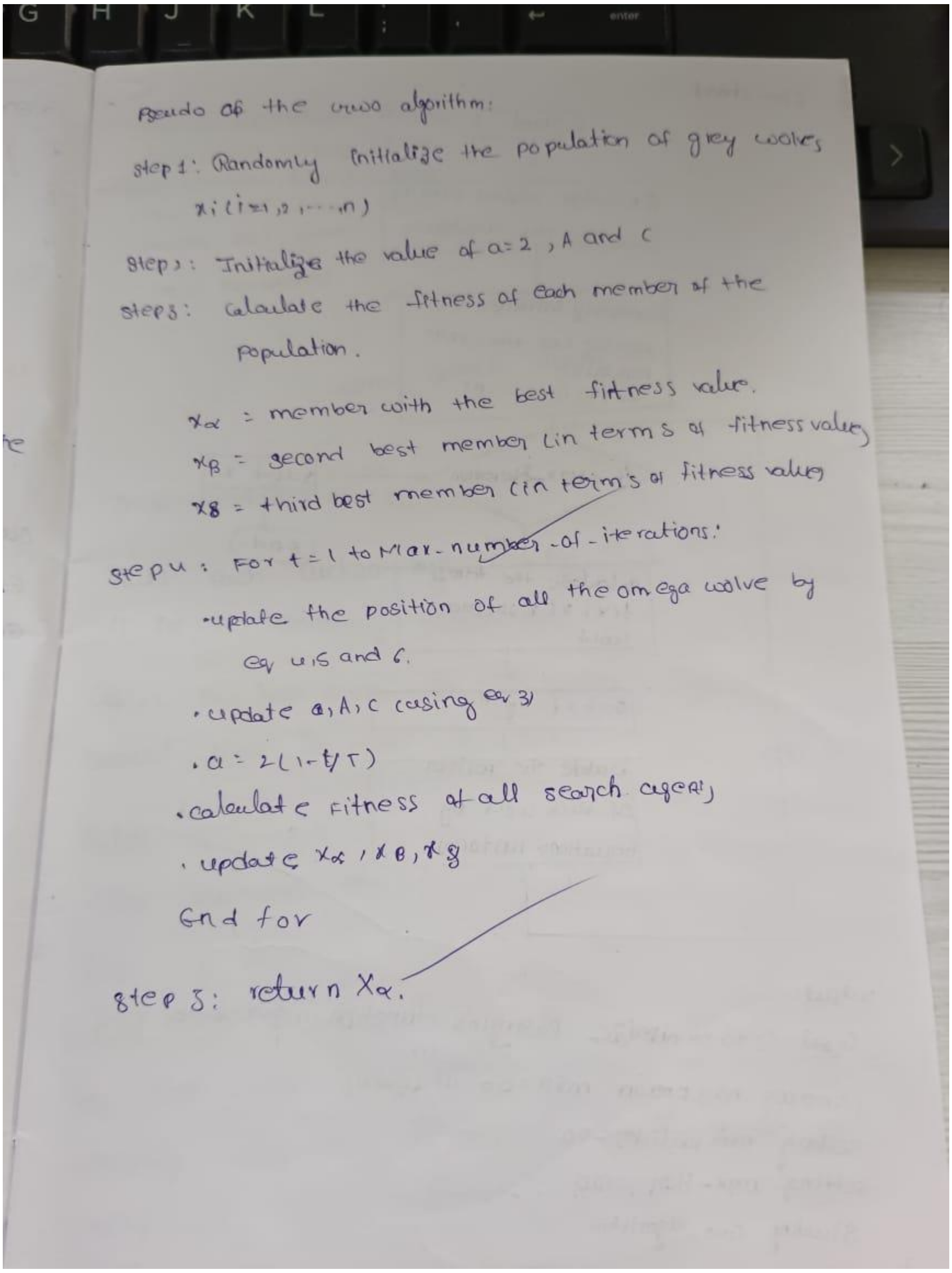
```
Iteration 1/10: Global Best Score = 0.02723138518053337
Iteration 2/10: Global Best Score = 0.02723138518053337
Iteration 3/10: Global Best Score = 0.02723138518053337
Iteration 4/10: Global Best Score = 0.02723138518053337
Iteration 5/10: Global Best Score = 0.000857224508156451
Iteration 6/10: Global Best Score = 0.000857224508156451
Iteration 7/10: Global Best Score = 0.000857224508156451
Iteration 8/10: Global Best Score = 0.000857224508156451
Iteration 9/10: Global Best Score = 5.556248703274748e-05
Iteration 10/10: Global Best Score = 5.556248703274748e-05

Best Position: [-0.00724152 -0.00176718]
Best Score: 5.556248703274748e-05
```

LAB 4 WEEK:

Program 4

Problem statement: Grey Wolf Optimizer (GWO)

Algorithm:



Pseudo of the GWO algorithm:

step 1: Randomly initialize the population of grey wolves x_i ($i = 1, 2, \dots, n$)

step 2: Initialize the value of $a = 2$, A and C

step 3: Calculate the fitness of each member of the population.

x_α = member with the best fitness value.

x_β = second best member (in terms of fitness value)

x_δ = third best member (in term's of fitness value)

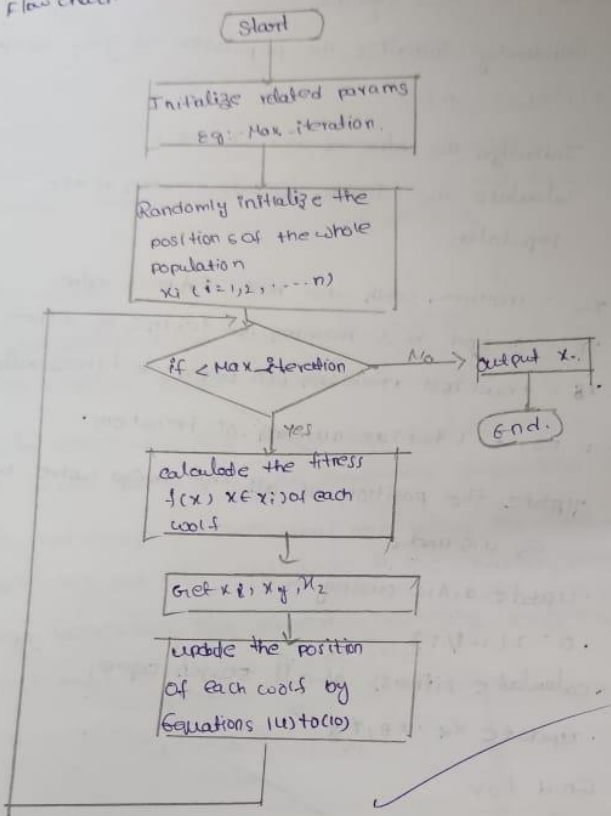
step 4: For t = 1 to Max-number-of-iterations:

- update the position of all the omega wolve by eq 4, 5 and 6.
- update a, A, C using eq 3)
- $a = 2(1 - t/T)$
- calculate fitness of all search agent
- update $x_\alpha, x_\beta, x_\delta$

End for

step 5: return X_α.

Flow chart



output:

Goal is to minimize Rastrigin's function in 3 variables

Functions has known min = 0.0 at (0,0,0)

setting num particles = 50

setting max-iter = 100

Starting GWO algorithm.

iter = 10 best fitness = 2.996

Iter = 20 best fitness = 2.944

Iter = 30 best fitness = 0.470

Iter = 40 best fitness = 0.985

Iter = 50 best fitness = 0.005

Iter = 60 best fitness = 0.001

Iter = 70 best fitness = 0.048

Iter = 80 best fitness = 0.001

Iter = 90 best fitness = 0.000

GWO completed.

Code:

```
import numpy as np

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

class GreyWolfOptimizer:
    def __init__(self, objective_function, dim, num_wolves, max_iter, lb=-5, ub=5):
        self.objective_function = objective_function
        self.dim = dim
        self.num_wolves = num_wolves
        self.max_iter = max_iter
        self.lb = lb
        self.ub = ub
        self.positions = np.random.uniform(self.lb, self.ub, (self.num_wolves, self.dim))
        self.alpha_position = np.zeros(self.dim)
        self.beta_position = np.zeros(self.dim)
        self.delta_position = np.zeros(self.dim)
        self.alpha_score = float("inf")
        self.beta_score = float("inf")
        self.delta_score = float("inf")
    def update_positions(self, a):
        for i in range(self.num_wolves):
            A1 = 2 * a * np.random.rand(self.dim) - a
            C1 = 2 * np.random.rand(self.dim)
            D_alpha = abs(C1 * self.alpha_position - self.positions[i])
            X1 = self.alpha_position - A1 * D_alpha
            A2 = 2 * a * np.random.rand(self.dim) - a
            C2 = 2 * np.random.rand(self.dim)
            D_beta = abs(C2 * self.beta_position - self.positions[i])
            X2 = self.beta_position - A2 * D_beta
            A3 = 2 * a * np.random.rand(self.dim) - a
            C3 = 2 * np.random.rand(self.dim)
```

```
            D_delta = abs(C3 * self.delta_position - self.positions[i])
            X3 = self.delta_position - A3 * D_delta
            self.positions[i] = (X1 + X2 + X3) / 3
            self.positions[i] = np.clip(self.positions[i], self.lb, self.ub)
    def evaluate_fitness(self):
        for i in range(self.num_wolves):
            fitness = self.objective_function(self.positions[i])
            if fitness < self.alpha_score:
                self.alpha_score = fitness
                self.alpha_position = self.positions[i]
            elif fitness < self.beta_score:
                self.beta_score = fitness
                self.beta_position = self.positions[i]
            elif fitness < self.delta_score:
                self.delta_score = fitness
                self.delta_position = self.positions[i]
    def run(self):
        a = 2
        for t in range(self.max_iter):
            a = 2 - t * (2 / self.max_iter)
            self.update_positions(a)
            self.evaluate_fitness()
            print(f"Iteration {t+1}/{self.max_iter}: Alpha Score = {self.alpha_score}")
        return self.alpha_position, self.alpha_score
dim = 2
num_wolves = 30
max_iter = 10
lb = -5
ub = 5
gwo = GreyWolfOptimizer(objective_function, dim, num_wolves, max_iter, lb, ub)
best_position, best_score = gwo.run()
```

print(f"\nBest Position: {best_position}")

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

```
Iteration 1/10: Alpha Score = 0.00039706625480835054
Iteration 2/10: Alpha Score = 0.00039706625480835054
Iteration 3/10: Alpha Score = 0.00039706625480835054
Iteration 4/10: Alpha Score = 0.00039706625480835054
Iteration 5/10: Alpha Score = 0.00039706625480835054
Iteration 6/10: Alpha Score = 0.00016150473046534417
Iteration 7/10: Alpha Score = 0.00015472273041535067
Iteration 8/10: Alpha Score = 7.06318886984631e-05
Iteration 9/10: Alpha Score = 4.878663579776153e-05
Iteration 10/10: Alpha Score = 4.838048252992564e-05

Best Position: [-0.0007949   0.00691004]
Best Score: 4.838048252992564e-05
```

LAB 5 WEEK:

Program 5

Problem statement: Cuckoo Search (CS)

Algorithm:

Algorithm steps:

1. Initialize nests: Randomly generate an initial population of nest x_i
2. Evaluate fitness: Evaluate the fitness $f(x_i)$ of each nest.
3. Repeat until termination:

Generate new solutions for each cuckoo:

- Generate new solution x_i^{new} by Levy flight:

$$x_i^{new} = x_i + \alpha \cdot Levy(\lambda)$$

- levy is a random walk drawn from the levy distribution α is a step size.

- Evaluate fitness of new solution: $f(x_i^{new})$
- Replace the old nest with the new nest if its better,

~~replace t~~

- Abandon bad nests: with probability P_a, some host nest are abandoned. This mimics the abandonment behaviour of host birds discovering parasitic eggs.

4. Termination

- Algorithm stops when either the maximum no of generations T_{max} is reached or the fitness value meets the stopping criterion.

output :

Generation 1 : Best fitness = ~~21.0829099709810003~~ 48.4028249137589 84

Generation 2 : Best fitness = 27.034367677 83532

Generation 3 : Best fitness = 23.5505190662 0632

Generation 4 : Best fitness = 37.4087377928 63825

optimal solution Found:

Best Solution : [-5.17905866 3.25382708]

Best Fitness : 37.408737792863825.

o/p

27/11/24

Code:

```
import numpy as np
def objective_function(x):
    return np.sum(x**2)
class CuckooSearch:
    def __init__(self, objective_function, dim, num_nests, max_iter, pa=0.25, lb=-5, ub=5):
        self.objective_function = objective_function
        self.dim = dim
        self.num_nests = num_nests
```

```
        self.max_iter = max_iter
        self.pa = pa
        self.lb = lb
        self.ub = ub
        self.nests = np.random.uniform(self.lb, self.ub, (self.num_nests, self.dim))
        self.fitness = np.array([self.objective_function(nest) for nest in self.nests])
        self.best_nest = self.nests[np.argmin(self.fitness)]
        self.best_fitness = np.min(self.fitness)
    def levy_flight(self):
        step = np.random.normal(0, 1, self.dim) * np.random.uniform(0, 1)**(1/1.5)
        return step
    def generate_new_solution(self, nest):
        step = self.levy_flight()
        new_nest = nest + step
        new_nest = np.clip(new_nest, self.lb, self.ub)
        return new_nest
    def abandon_worst_nests(self):
        num_worst = int(self.pa * self.num_nests)
        worst_indices = np.argsort(self.fitness)[-num_worst:]
        for i in worst_indices:
            new_nest = np.random.uniform(self.lb, self.ub, self.dim)
            self.nests[i] = new_nest
            self.fitness[i] = self.objective_function(new_nest)
    def run(self):
        for t in range(self.max_iter):
            for i in range(self.num_nests):
                new_nest = self.generate_new_solution(self.nests[i])
                new_fitness = self.objective_function(new_nest)
                if new_fitness < self.fitness[i]:
                    self.nests[i] = new_nest
                    self.fitness[i] = new_fitness
```

```
            if new_fitness < self.best_fitness:
                self.best_nest = new_nest
                self.best_fitness = new_fitness
        self.abandon_worst_nests()
        print(f"Iteration {t+1}/{self.max_iter}: Best Fitness = {self.best_fitness}")
    return self.best_nest, self.best_fitness
dim = 2
num_nests = 30
max_iter = 10
pa = 0.25
lb = -5
ub = 5
cs = CuckooSearch(objective_function, dim, num_nests, max_iter, pa, lb, ub)
best_nest, best_fitness = cs.run()
print(f"\nBest Nest: {best_nest}")
print(f"Best Fitness: {best_fitness}")
```

```
Iteration 1/10: Best Fitness = 1.1321338482886207
Iteration 2/10: Best Fitness = 1.1321338482886207
Iteration 3/10: Best Fitness = 0.9581521162910784
Iteration 4/10: Best Fitness = 0.9581521162910784
Iteration 5/10: Best Fitness = 0.4939211503841384
Iteration 6/10: Best Fitness = 0.026043904239381056
Iteration 7/10: Best Fitness = 0.008822631651070133
Iteration 8/10: Best Fitness = 0.003456235504629316
Iteration 9/10: Best Fitness = 0.003456235504629316
Iteration 10/10: Best Fitness = 0.003456235504629316

Best Nest: [-0.01878645 -0.05570731]
Best Fitness: 0.003456235504629316
```

LAB 6 WEEK:

Program 6

Problem statement: Parallel Cellular Algorithms and Programs

Algorithm:

Ramesh

Parallel cellular algorithm:

cells dive on a gird have a state and neighbourhood, interaction and dependency with neighbourhood state.

core principles

(i) cells as solution.
(ii) neighbour interaction.
(iii) parallelism.
(iv) distributed approch.

steps:

(i) define problem.
(ii) initialize parameters.
(iii) initialize population.
(iv) evaluate fitness.
(v) update solution,
(vi) iterate
(vii) output best solution.

Statement:

minimize $f(x) = x^2 - 4x + 4$

number of cells = 100

grid size = 10 x 10 20

neighbourhood structure 3x3

iterations = 100.

input : f(x), grid size, maxiter, neighbors

output : best solution.

initialize:

create grid, assign random structure to cells bestsol = rom,

bestfit = ∞.

ii) evaluate initial fitness.

compute for each cell using f(x)

update best sol, best fit

iii) iterate (maxiter):

for each cell

update state based on neighbours calculate fitness

for all cells update bestsol

iv) output:

best sol and its fitness.

Applications:

i) optimization problems.

ii) image processing.

iii) resource allocation.

iv) parallel computing.

O/P

output:

Iteration 1/10: Best Fitness = 4.4338109246151464e-05

⋮

Iteration 10/10: Best Fitness = 5.315313718057261e-07.

Best cell: 1.9992709380192222

Best Fitness: 5.3153137180572058e-07.

Code:

import numpy as np

def objective_function(x):

    return x**2 - 4*x + 4

class ParallelCellularAlgorithm:

    def __init__(self, objective_function, grid_size, max_iter, lb=-5, ub=5):

        self.objective_function = objective_function

        self.grid_size = grid_size

```python
        self.max_iter = max_iter
        self.lb = lb
        self.ub = ub
        self.cells = np.random.uniform(self.lb, self.ub, (grid_size, grid_size))
        self.fitness = np.array([[self.objective_function(cell) for cell in row] for row in self.cells])
        self.best_cell = self.cells[np.unravel_index(np.argmin(self.fitness), self.fitness.shape)]
        self.best_fitness = np.min(self.fitness)
    def update_state(self, cell, neighbors):
        best_neighbor = min(neighbors, key=lambda x: self.objective_function(x))
        new_cell = best_neighbor + np.random.normal(0, 0.1)
        return np.clip(new_cell, self.lb, self.ub)
    def get_neighbors(self, row, col):
        neighbors = []
        for i in range(max(0, row-1), min(self.grid_size, row+2)):
            for j in range(max(0, col-1), min(self.grid_size, col+2)):
                if i != row or j != col:
                    neighbors.append(self.cells[i, j])
        return neighbors
    def run(self):
        for t in range(self.max_iter):
            for i in range(self.grid_size):
                for j in range(self.grid_size):
                    neighbors = self.get_neighbors(i, j)
                    new_cell = self.update_state(self.cells[i, j], neighbors)
                    new_fitness = self.objective_function(new_cell)
                    if new_fitness < self.fitness[i, j]:
                        self.cells[i, j] = new_cell
                        self.fitness[i, j] = new_fitness
                    if new_fitness < self.best_fitness:
                        self.best_cell = new_cell
                        self.best_fitness = new_fitness
```

```
        print(f"Iteration {t+1}/{self.max_iter}: Best Fitness = {self.best_fitness}")

    return self.best_cell, self.best_fitness

grid_size = 5

max_iter = 10

lb = -5

ub = 5

pca = ParallelCellularAlgorithm(objective_function, grid_size, max_iter, lb, ub)

best_cell, best_fitness = pca.run()

print(f"\nBest Cell: {best_cell}")

print(f"Best Fitness: {best_fitness}")
```

```
Iteration 1/10: Best Fitness = 4.4338109446151464e-05
Iteration 2/10: Best Fitness = 7.242042420863015e-06
Iteration 3/10: Best Fitness = 7.242042420863015e-06
Iteration 4/10: Best Fitness = 7.242042420863015e-06
Iteration 5/10: Best Fitness = 5.315313718057268e-07
Iteration 6/10: Best Fitness = 5.315313718057268e-07
Iteration 7/10: Best Fitness = 5.315313718057268e-07
Iteration 8/10: Best Fitness = 5.315313718057268e-07
Iteration 9/10: Best Fitness = 5.315313718057268e-07
Iteration 10/10: Best Fitness = 5.315313718057268e-07

Best Cell: 1.9992709380192222
Best Fitness: 5.315313718057268e-07
```

LAB 7 WEEK:

Program 7

Problem statement: Optimization via Gene Expression Algorithms

Algorithm:

optimization via gene expression:

Algorithm:

Encodes solution line gene sequences and rrern mrougn selection, crossover mutation gene expression.

Steps:

(i) define problem.

ii) initialize parameters.

iii) initialize population.

iv) evaluate fitness.

v) selection.

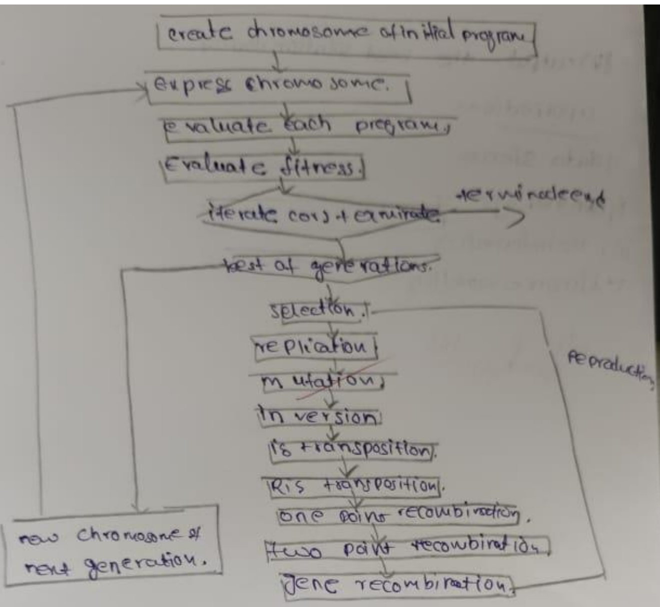
vi) crossover.

vii) mutation

viii) gene expression.

ix) iterals.

x) output the best solution.



Algorithm:

(i) initialize:

generate population.

set parameters: mutation, selection rate, generation,

(ii) repeat for each generation.

evalute f(x)

select parents.

apply

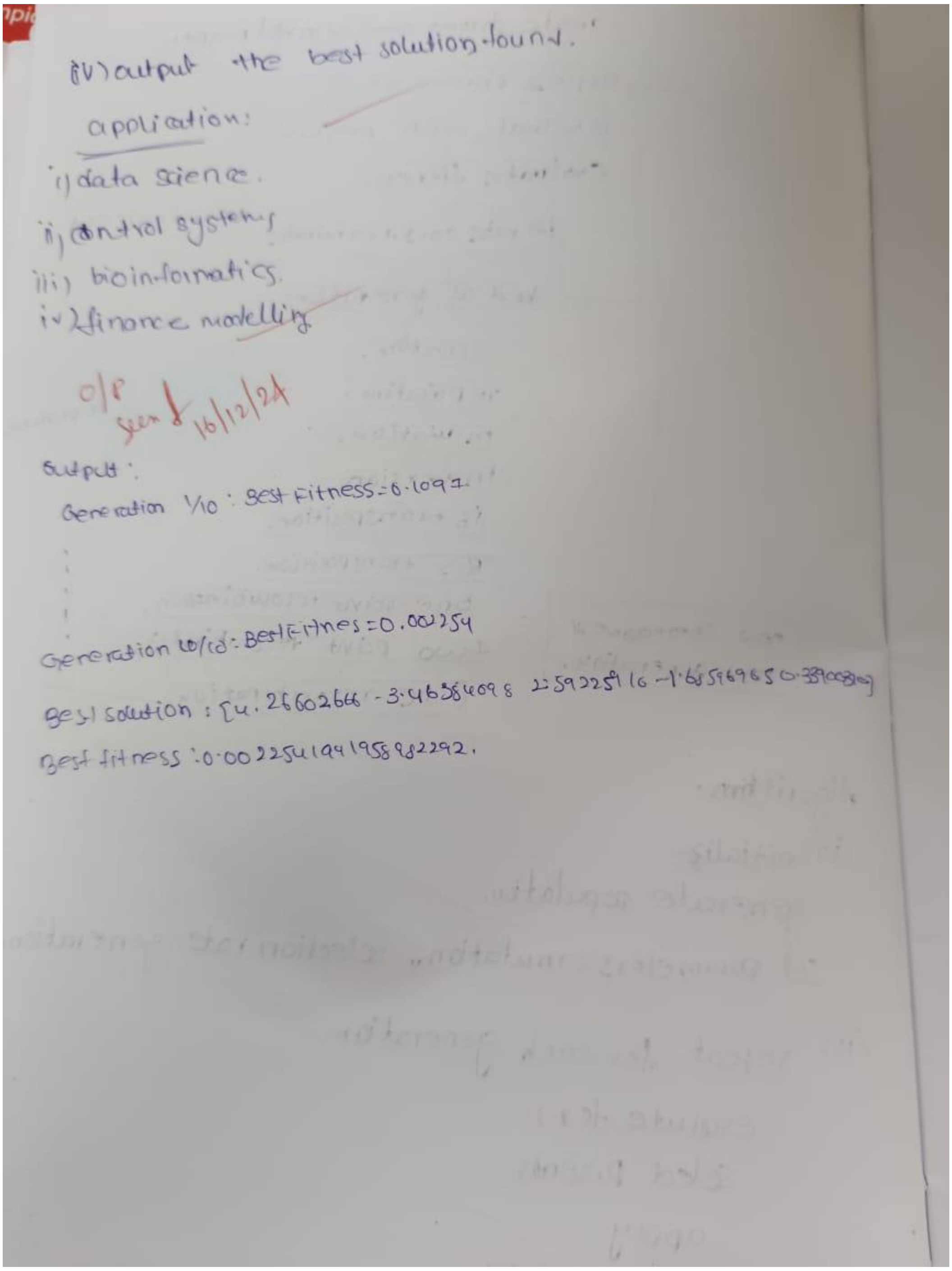
cross over

mutation

geneexpression.

(iii) check stopping criteria

If max iter (or) solution reached stop.


iv) output the best solution found.

application:

i) data sience.

ii) control systems

iii) bioinformatics.

iv) finance modelling

o/p seen 16/12/24

output:

Generation 1/10: Best Fitness = 0.1097

Generation 10/10: Best Fitnes = 0.002254

Best solution: [4.2660266 -3.46384098 2.59225916 -1.65596965 0.33900809]

Best fitness: 0.00225419941958982292.

Code:

```python
import numpy as np

def objective_function(x):
    return x**2 - 4*x + 4

class GeneExpressionAlgorithm:
    def __init__(self, objective_function, population_size, num_genes, max_generations, mutation_rate=0.01, crossover_rate=0.7, lb=-5, ub=5):
        self.objective_function = objective_function
```

```
        self.population_size = population_size
        self.num_genes = num_genes
        self.max_generations = max_generations
        self.mutation_rate = mutation_rate
        self.crossover_rate = crossover_rate
        self.lb = lb
        self.ub = ub
        self.population = np.random.uniform(self.lb, self.ub, (self.population_size, self.num_genes))
        self.fitness = np.array([self.objective_function(individual.sum()) for individual in self.population])
        self.best_solution = self.population[np.argmin(self.fitness)]
        self.best_fitness = np.min(self.fitness)
    def selection(self):
        total_fitness = np.sum(self.fitness)
        probabilities = (total_fitness - self.fitness) / total_fitness
        probabilities /= np.sum(probabilities)
        selected_indices = np.random.choice(np.arange(self.population_size), size=self.population_size,
p=probabilities)
        selected_population = self.population[selected_indices]
        return selected_population
    def crossover(self, parent1, parent2):
        if np.random.rand() < self.crossover_rate:
            crossover_point = np.random.randint(1, self.num_genes)
            child1 = np.concatenate((parent1[:crossover_point], parent2[crossover_point:]))
            child2 = np.concatenate((parent2[:crossover_point], parent1[crossover_point:]))
            return child1, child2
        else:
            return parent1, parent2
    def mutation(self, individual):
        if np.random.rand() < self.mutation_rate:
            mutation_point = np.random.randint(self.num_genes)
            individual[mutation_point] = np.random.uniform(self.lb, self.ub)
```

```
        return individual

    def gene_expression(self, individual):
        return individual.sum()

    def run(self):
        for generation in range(self.max_generations):
            selected_population = self.selection()
            new_population = []
            for i in range(0, self.population_size, 2):
                parent1, parent2 = selected_population[i], selected_population[i+1]
                child1, child2 = self.crossover(parent1, parent2)
                new_population.extend([self.mutation(child1), self.mutation(child2)])


            self.population = np.array(new_population)
            self.fitness = np.array([self.objective_function(self.gene_expression(individual)) for individual
in self.population])
            current_best_solution = self.population[np.argmin(self.fitness)]
            current_best_fitness = np.min(self.fitness)
            if current_best_fitness < self.best_fitness:
                self.best_solution = current_best_solution
                self.best_fitness = current_best_fitness
            print(f"Generation {generation+1}/{self.max_generations}: Best Fitness = {self.best_fitness}")
        return self.best_solution, self.best_fitness
population_size = 30
num_genes = 5
max_generations = 10
lb = -5
ub = 5
gea = GeneExpressionAlgorithm(objective_function, population_size, num_genes, max_generations,
lb=lb, ub=ub)
best_solution, best_fitness = gea.run()
print(f"\nBest Solution: {best_solution}")
print(f"Best Fitness: {best_fitness}")
```

```
Generation 1/10: Best Fitness = 0.10977190093840417
Generation 2/10: Best Fitness = 0.0022541941958982292
Generation 3/10: Best Fitness = 0.0022541941958982292
Generation 4/10: Best Fitness = 0.0022541941958982292
Generation 5/10: Best Fitness = 0.0022541941958982292
Generation 6/10: Best Fitness = 0.0022541941958982292
Generation 7/10: Best Fitness = 0.0022541941958982292
Generation 8/10: Best Fitness = 0.0022541941958982292
Generation 9/10: Best Fitness = 0.0022541941958982292
Generation 10/10: Best Fitness = 0.0022541941958982292

Best Solution: [ 4.26602667 -3.46384098  2.59225916 -1.68596965  0.33900316]
Best Fitness: 0.0022541941958982292
```