VISVESVARAYA TECHNOLOGICAL UNIVERSITY

"JnanaSangama", Belgaum -590014, Karnataka.

LAB RECORD

Bio Inspired Systems (23CS5BSBIS)

Submitted by

Tamanna Barman(1WA23CS006)

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 Dec-2025

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 **Tamanna Barman (1WA23CS006),** 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.

Dr.Shwetha KS Assistant Professor Department of CSE, BMSCE	Dr. Kavitha Sooda Professor & HOD Department of CSE, BMSCE

INDEX

INDEX

Name Tamanna Barman

Standard 5th Section G Roll No. 1WA23CS006

Subject BIS Lab copy.

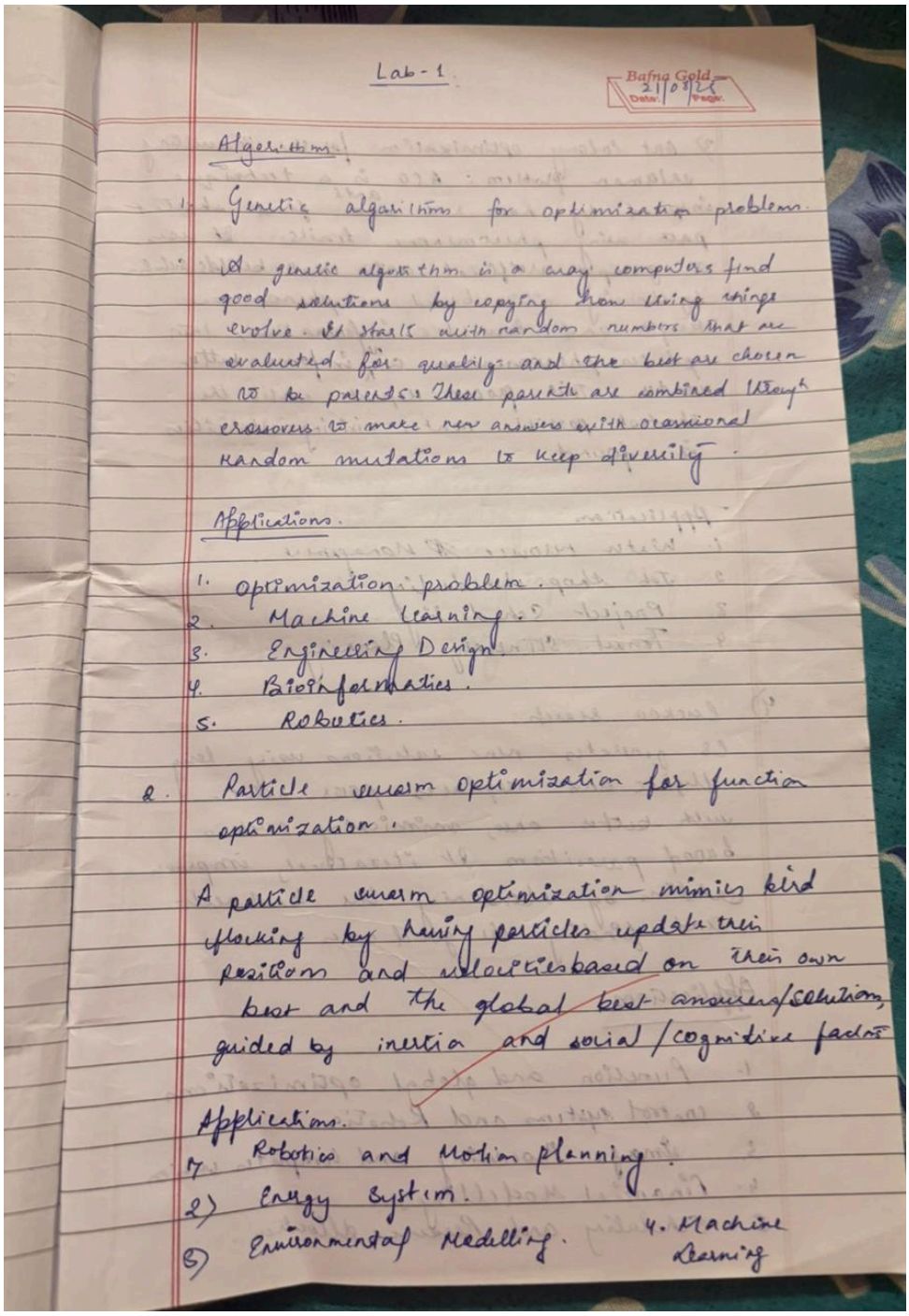
SL No.	Date	Title	Page No.	Teacher Sign / Remarks
1	21.08.25	LAB-1 Bio Inspired Systems algorithms	10	
2	21.08.25	LAB-2. Genetic algorithm pseudocode, code & output.	10	
3	28.08.25	LAB-3. Optimization via Gene expression		
4.	11.9.25	Lab-4 particle swarm optimization; Job scheduling	10	
5.	9-10-25	lab 5 ant optimization Travels sales Person	10	
6	16-10-25	Lab 6 Cuckoo Search Algo - knapsack Problem.	10/10	
7.	6/11/25	Lab 7 grey wolf Optimization	10/10	
8	13/11/25	Lab 8 Parallel Cellular Algorithm	10	

10/10

20/11

Github Link:https://github.com/1WA23CS006/bio-inspired-systems-lab-programs-

Program 1 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.



3) Ant Colony optimization for the travelling salesman problem: ACO is a technique inspired by how real ants find shortest path using pheromone trails. It uses a group of artificial ants that builds solutions by choosing cities based on pheromone strength and heuristic information, then updates pheromones on the route in force better routes. This process repeats until the shortest possible route visiting all cities is found.

- Applications
1. Water resource Management.
2. Job shop scheduling
3. Project Scheduling.
4. Travel Itinerary Planning.

4) Cuckoo search:
CS generates new solutions using levy flights and replaces poor solutions with better ones, mimicking cuckoo brood parasitism. It iteratively improves solutions by abandoning the bad nests and finding the good ones.

Applications

1. Function and global optimizations
2. control systems and Robotics.
3. Image Processing and computer vision
4. Financial Modelling.
5. Scheduling and Resource allocation.

5)

6)

5) Grey wolf optimized.

Grey wolf optimizer mimics how grey wolves hunt, with alpha, beta and delta wolves leading the search with other follows. The wolves update their positions by following the leaders to gradually find the best solution.

Application

1. Engineering Design.
2. Machine learning.
3. Renewable Energy.
4. Medical Imaging & Diagnosis.

6) Parallel Cellular Algorithms and Programs:

21.08.2025

LAB-2

Genetic algorithm (GA)

→ A genetic algorithm is an optimization and search technique based on the principle of natural selection and genetics. It belongs to the class of evolutionary algorithm and is widely used to solve complex optimization problems.

⇒ select initial population.
→ calculate the fitness.
→ selecting mating pool.
→ crossover.
→ Mutation.

The initial population for the value 'x' ranging from 0 to 31.

Stage no.	Initial population	(x) value	fitness function (x^2)
1	01100	12	144
2	11001	25	625
3	00101	5	25
4	10011	19	361
	Sum		1155
	Average		288.75
	Maximum		625

$$\text{Prob} = \frac{f(x)}{\Sigma f(x)}$$

Prob.	% Prob	Expected count.
0.1247.	1. 12.47.	1. 0.49 → 1
0.5411	2. 54.11	2. 2.1625 → 2
0.0216	3. 2.16	3. 0.0866 → 0
0.3126		

1. Single point Crossover.

				after crossover
P1 —	11	100	P1	11 | 101
P2 —	10	101	P2	10 | 100

2. Multipoint Crossover.
3. Uniform Crossover.

Children
C1 = 11 + 101 = 11101 (29) } offspring.
C2 = 10 + 100 = 10100 (20)

Two pt crossover.
P1 = 01010 (10)
P2 = 11001 (25)

after 1st and 4th bit
P1 = 0 | 101 | 0
P2 = 1 | 100 | 1

Swap the middle part.
C1 = 0 + 100 + 0 = 01000 (8)
C2 = 1 + 101 + 1 = 11011 (27)
New offspring : 8 and 27

Uniform crossover

bits → 50% from P1
50% from P2.

expected = $\frac{f(x_i)}{\text{avg}\,(\Sigma f(x))}$

Selecting Mating pool

String no.	Mating pool	crossover pt	Offspring after crossover	x	$f(x)=x^2$
1	01101	4	01101	13	169
2	11001		11000	24	576
3	11001	2	11011	27	729
4	10011		10001	17	289

total = 1763
avg = 440.75
max : 729

Mutation

String no.	offspring after crossover	mutation chromosome for flipping	offspring after mutation	x value	$f(x)=x^2$
1	01101	10000	11101	29	841
2	11000	00000	11000	24	576
3	11011	00000	11011	27	729
4	10001	00100	10100	20	400

Bafna Gold
Date: Page:

Genetic Algorithm.

```
import random, math
pop, GENES = 50, 16
MUT-RATE, CROSS-RATE = 0.01, 0.8
GENS = 5
X-MIN, X-MAX = -4, 8

def fitness(x):
    return x * math.sin(10 * math.pi * x) + 1

def decode(chromo):
    val = int(chromo, 2)
    return X-MIN + (X-MAX - X-MIN) * val / ((2**GENES - 1)

def init-pop():
    return [''.join(random.choice('01') for _ in range(GENES)) for _ in range(POP)]

def evaluate(pop):
    return [fitness(decode(ind)) for ind in pop]

def select(pop, fits):
    selected = []
    for _ in range(POP):
        i, j = random.sample(range(POP), 2)
        selected.append(pop[i] if fits[i] > fits[j]
            else pop[j])
        return selected
```

```
def crossover(p1, p2):
    if random.random() < CROSS_RATE:
        point = random.randint(1, GENES-1)
        return p1[:point] + p2[point:],
        p2[:point] + p1[point:]
    return p1, p2

def mutate(chromo):
    return ''.join('1' if bit == '0' else '0'
    if random.random() < MUT_RATE else
    bit for bit in chromo)

def ga():
    pop = init_pop(POP)
    for gen in range(GENS):
        fits = evaluate(pop)
        best = max(fits)
        best_ind = pop[fits.index(best)]
        print(f"Gen {gen+1}: Best
        fitness = {best:5f}, x =
        {decode(best_ind):.5f}")
        selected = select(pop, fits)
        pop = []
        for i in range(0, POP, 2):
            c1, c2 = crossover(selected[i],
                               selected[i+1])
            pop.append(mutate(c1))
            pop.append(mutate(c2))
    fits = evaluate(pop)
    best = max(fits)
    best_ind = pop[fits.index(best)]
    print(f"\n Best soln: x = {decode(best_ind):
```

Bafna Gold
Date: Page:

{f}, f(x) = {best :. 5f}")

if name == "__main__":
    ga()

output :

Gen 1 : Best fitness = 2.83121, x = 1.85512
Gen 2 : Best fitness = 1.69346, x = 1.81250
Gen 3 : Best fitness = 2.24999, x = 1.24999
Gen 4 : Best fitness = 2.24999, x = 1.24999
Gen 5 : Best fitness = 1.69346, x = 1.81250

Best solution : x = 2.00000, f(x) = 1.00000

Initial population : ['01100', '10111', '00101', '10011']
[12, 23, 5, 19]

Generation 1

x = 12, bin = 01100, fit = 144, prob = 0.136, exp_count = 0.54

x = 23, bin = 10111, fit = 529, prob = 0.500, exp-count = 2.00

x = 5, bin = 00101, fit = 25, prob = 0.024, exp_count = 0.09

x = 19, bin = 10011, fit = 361, prob = 0.313, exp-count = 1.25

Generation 2

x = 28, bin = 11100, fit = 784, prob = 0.635, exp_count = 2.54

x = 9, bin = 01001, fit = 81, prob = 0.066, exp-count = 0.26

x=3, bin = 00011, fit=9, prob=0.007, exp.count = 0.03

x=19, bin = 10011, fit = 361, prob = 0.292, exp-count = 1.17

Generation 3.

x = 25, bin = 11001, fit = 625, prob = 0.334, exp-count = 1.33

x = 8, bin = 01000, fit = 64, prob = 0.034, exp-count = 0.14

x = 28, bin = 11100, fit = 784, prob = 0.419, exp-count = 1.67

x = 20, bin = 10100, fit = 400, prob = 0.214, exp-count = 0.85

Generation 4.

x = 21, bin = 10101, fit = 441, prob = 0.252, exp-count = 1.01

x = 24, bin = 11000, fit = 576, prob = 0.330, exp-count 1.32

x = 21, bin = 10101, fit = 441, prob = 0.252, exp-count 1.01

x = 17, bin = 10001, fit = 289, prob = 0.165, exp-count = 0.66

Generation 5.

x = 21, bin = 10101, fit = 441, prob : 0.208, exp-count = 0.83

x = 20, bin = 10100, fit = 400, prob : 0.188, exp-count = 0.75

x = 29, bin = 11101, fit = 841, prob : 0.396, exp-count = 1.54

x = 21, bin = 10101, fit = 441, prob : 0.208, exp-count = 0.83

Final best solution : 29 11101 fitness = 841

Code:

```
import random

CHROM_LENGTH = 5
CROSS_RATE = 0.8
MUT_RATE = 0.1

def fitness(x):
    return x**2

def encode(x):
    return format(x, f'0{CHROM_LENGTH}b')

def decode(b):
    return int(b, 2)

def roulette_selection(pop, fitnesses):
    total_fit = sum(fitnesses)
    pick = random.uniform(0, total_fit)
    current = 0
    for i, f in enumerate(fitnesses):
        current += f
        if current > pick:
            return pop[i]
    return pop[-1]

def crossover(p1, p2):
    if random.random() < CROSS_RATE:
        point = random.randint(1, CHROM_LENGTH-1)
        c1 = p1[:point] + p2[point:]
        c2 = p2[:point] + p1[point:]
        return c1, c2
    return p1, p2

def mutate(chrom):
    chrom_list = list(chrom)
    for i in range(CHROM_LENGTH):
        if random.random() < MUT_RATE:
            chrom_list[i] = '1' if chrom_list[i] == '0' else '0'
    return ''.join(chrom_list)

def genetic_algorithm():
    user_input = input("Enter initial population values (space-separated, e.g. 12 23 5 19): ")
    values = list(map(int, user_input.split()))
```

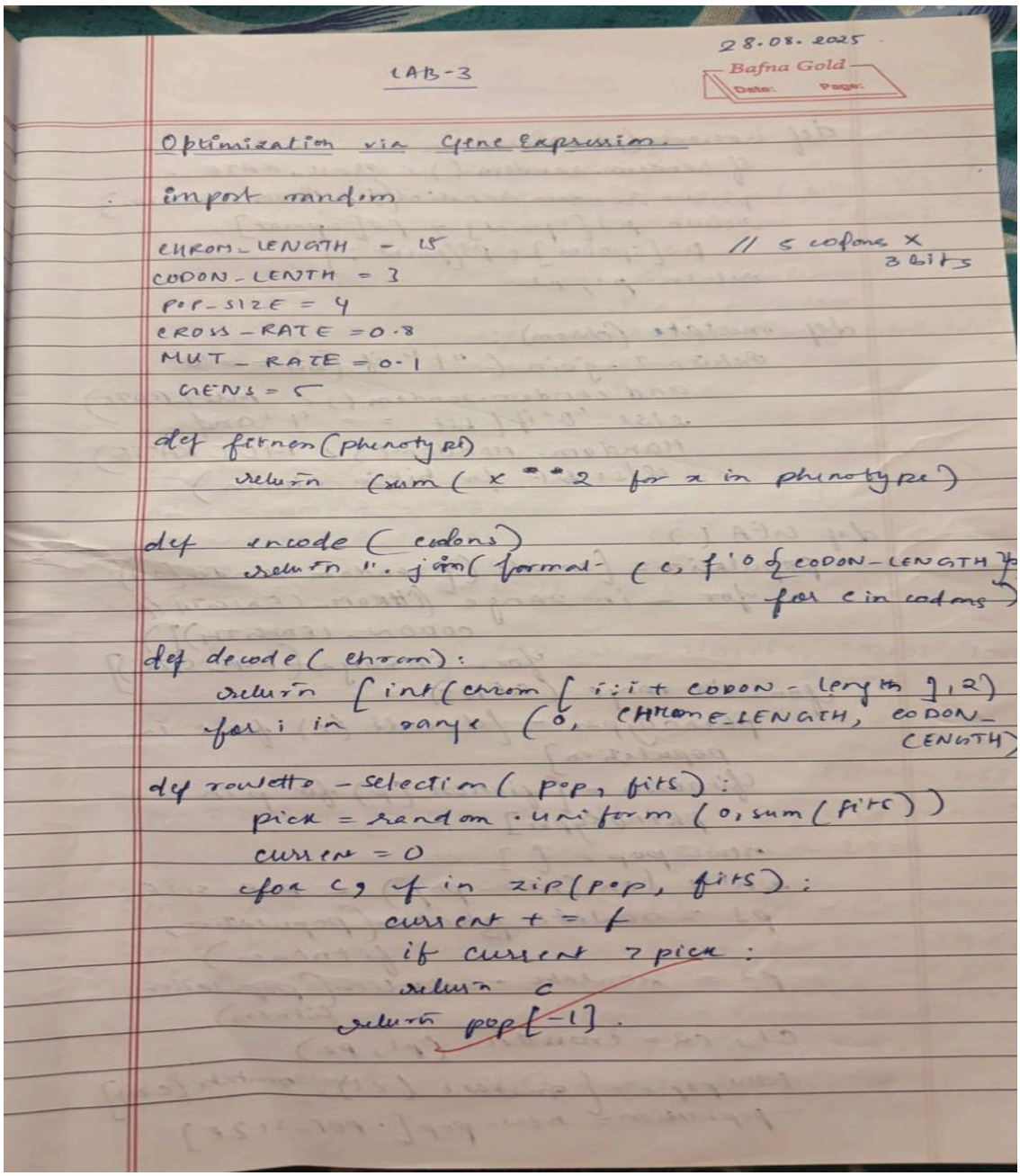
```
    generations = int(input("Enter number of generations to run: "))

    POP_SIZE = len(values)
    population = [encode(x) for x in values]

    print("\nInitial Population:", population, [decode(c) for c in population])

    global_best = population[0]
    global_best_fit = fitness(decode(global_best))

    for gen in range(1, generations + 1):
        decoded = [decode(c) for c in population]
        fitnesses = [fitness(x) for x in decoded]

        best_idx = fitnesses.index(max(fitnesses))
        if fitnesses[best_idx] > global_best_fit:
            global_best = population[best_idx]
            global_best_fit = fitnesses[best_idx]

        total_fit = sum(fitnesses)
        probs = [f / total_fit for f in fitnesses]
        expected = [p * POP_SIZE for p in probs]

        print(f"\nGeneration {gen}")
        for i in range(POP_SIZE):
            print(f"x={decoded[i]}, bin={population[i]}, fit={fitnesses[i]}, "
                  f"prob={probs[i]:.3f}, exp_count={expected[i]:.2f}")

        new_pop = [global_best]

        while len(new_pop) < POP_SIZE:
            p1 = roulette_selection(population, fitnesses)
            p2 = roulette_selection(population, fitnesses)
            c1, c2 = crossover(p1, p2)
            c1, c2 = mutate(c1), mutate(c2)
            new_pop.extend([c1, c2])

        population = new_pop[:POP_SIZE]

    decoded = [decode(c) for c in population]
    fitnesses = [fitness(x) for x in decoded]
    best_idx = fitnesses.index(max(fitnesses))
    print("\nFinal Best Solution:", decoded[best_idx], population[best_idx], "fitness=",
fitnesses[best_idx])
```

genetic_algorithm()
OUTPUT:

```
=== RESTART: C:/Users/student/AppData/Local/Programs/Python/Python313/gene.py ==
Initial Population: ['01100', '11001', '00101', '10011'] [12, 25, 5, 19]

Generation 1
x=12, bin=01100, fit=144, prob=0.125, exp_count=0.50
x=25, bin=11001, fit=625, prob=0.541, exp_count=2.16
x=5, bin=00101, fit=25, prob=0.022, exp_count=0.09
x=19, bin=10011, fit=361, prob=0.313, exp_count=1.25

Generation 2
x=19, bin=10011, fit=361, prob=0.211, exp_count=0.85
x=19, bin=10011, fit=361, prob=0.211, exp_count=0.85
x=25, bin=11001, fit=625, prob=0.366, exp_count=1.46
x=19, bin=10011, fit=361, prob=0.211, exp_count=0.85

Generation 3
x=17, bin=10001, fit=289, prob=0.137, exp_count=0.55
x=19, bin=10011, fit=361, prob=0.171, exp_count=0.68
x=25, bin=11001, fit=625, prob=0.295, exp_count=1.18
x=29, bin=11101, fit=841, prob=0.397, exp_count=1.59

Generation 4
x=9, bin=01001, fit=81, prob=0.050, exp_count=0.20
x=25, bin=11001, fit=625, prob=0.384, exp_count=1.54
x=9, bin=01001, fit=81, prob=0.050, exp_count=0.20
x=29, bin=11101, fit=841, prob=0.517, exp_count=2.07

Generation 5
x=29, bin=11101, fit=841, prob=0.517, exp_count=2.07
x=25, bin=11001, fit=625, prob=0.384, exp_count=1.54
x=9, bin=01001, fit=81, prob=0.050, exp_count=0.20
x=9, bin=01001, fit=81, prob=0.050, exp_count=0.20

Final Best Solution: 29 11101 fitness= 841
```

Problem 2

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



LAB-3

28.08.2025

Bafna Gold

Date: Page:

Optimization via Gene Expression.

```
import random

CHROM_LENGTH = 15          # 5 codons x 3 bits
CODON_LENTH = 3
POP_SIZE = 4
CROSS_RATE = 0.8
MUT_RATE = 0.1
GENS = 5

def fitness(phenotype)
    return (sum( x ** 2 for x in phenotype)

def encode ( codons)
    return ''.join( format- ( c, f'0{CODON_LENGTH}b
                    for c in codons )

def decode ( chrom):
    return [int(chrom[ i:i + CODON - length ],2)
    for i in range (0, CHROMOSE_LENGTH, CODON_
                                        LENGTH)

def roulette - selection ( pop, fits):
    pick = random.uniform (0, sum(fits))
    current = 0
    for c, f in zip(pop, fits):
        current += f
        if current > pick:
            return c
    return pop[-1].
```

```
def crossover(p1, p2):
    if random.random() < CROSS_RATE:
        point = random.randint(1, CHROM_LENGTH-1)
        return p1[:point] + p2[:point],
               p2[:point] + p1[point:]
    return p1, p2

def mutate(chrom):
    return ''.join("1" if (bit == '0'
        and random.random() < MUT_RATE)
        else "0" if (bit == '1' and
        random.random() < MUT_RATE)
        else bit for bit in chrom)

def GA():
    population = [encode([random.randint(0,7)
        for _ in range(CHROM_LENGTH //
                       CODON_LENGTH)])
                  for _ in range(POP_SIZE)]
    for _ in range(GENS):
        phenotypes = [decode(c) for c in
            population]
        fitnesses = [fitness(p) for p in
            phenotypes]
        new_pop = []
        while len(new_pop) < POP_SIZE:
            p1 = roulette_selection(population,
                                    fitnesses)
            p2 = roulette_selection(population,
                                    fitness)
            c1, c2 = crossover(p1, p2)
            new_pop += [mutate(c1), mutate(c2)]
        population = new_pop[:POP_SIZE]
```

Bafna Gold
Date: Page:

phenotypes = [decode(c) for c in population]
fitnesses = [fitness(f) for p in phenotypes]
best = fitnesses.index(max(fitnesses))
~~print("best phenotype:", phenotypes[best],~~
~~"fitness:", fitnesses[best])~~ chromosome
print("Final Best Solution:",
decoded[best_idx], population ~~best_idx~~
], "fitness =", fitness[best_idx])

~~GA()~~

~~output~~ genetic_algorithm

output

Enter 4 chromosomes (each 5 bits, only 0 or 1)
chromosome 1 : 00100
chromosome 2 : 10111
chromosome 3 : 00101
chromosome 4 : 10011

Initial population: ['00100', '10111', '00101', '10011']

Gen 1: Best Chromosomes : 10111, Expected_value =23,
Fitness = 1058

Gen 2: Best chromosomes : 11111 , Expected value : 31
fitness = 1922.

Gen 3 : Best chromosomes : 11111 , Expected_value : 31,
fitness = 1922

Gen 4: Best chromosomes : 10111, Expected value : 23,
fitness = 1058

Gen 5: Best chromosomes : 11011
Expected_value : 27
fitness = 1458.

Best Solution : 11111 expected_value = 31,
fitness = 1922

```
import random

POP_SIZE = 4
CHROM_LENGTH = 5
MAX_GENERATIONS = 5
MUTATION_RATE = 0.1

def gene_expression(chromosome):
    return int(chromosome, 2)

def fitness(chromosome):
    x = gene_expression(chromosome)
    return x * 2* x

def get_population_from_input():
    population = []
    print(f"Enter {POP_SIZE} chromosomes (each {CHROM_LENGTH} bits, only 0 or 1):")
    while len(population) < POP_SIZE:
        chrom = input(f"Chromosome {len(population) + 1}: ").strip()
        if len(chrom) == CHROM_LENGTH and all(c in '01' for c in chrom):
            population.append(chrom)
        else:
            print(f"Invalid chromosome! Please enter exactly {CHROM_LENGTH} bits (0 or 1).")
    return population

def select(population):
    fitnesses = [fitness(chrom) for chrom in population]
    total_fitness = sum(fitnesses)
    pick = random.uniform(0, total_fitness)
    current = 0
    for i, chrom in enumerate(population):
        current += fitnesses[i]
        if current > pick:
            return chrom

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

def mutate(chromosome):
    mutated = ''
    for bit in chromosome:
```

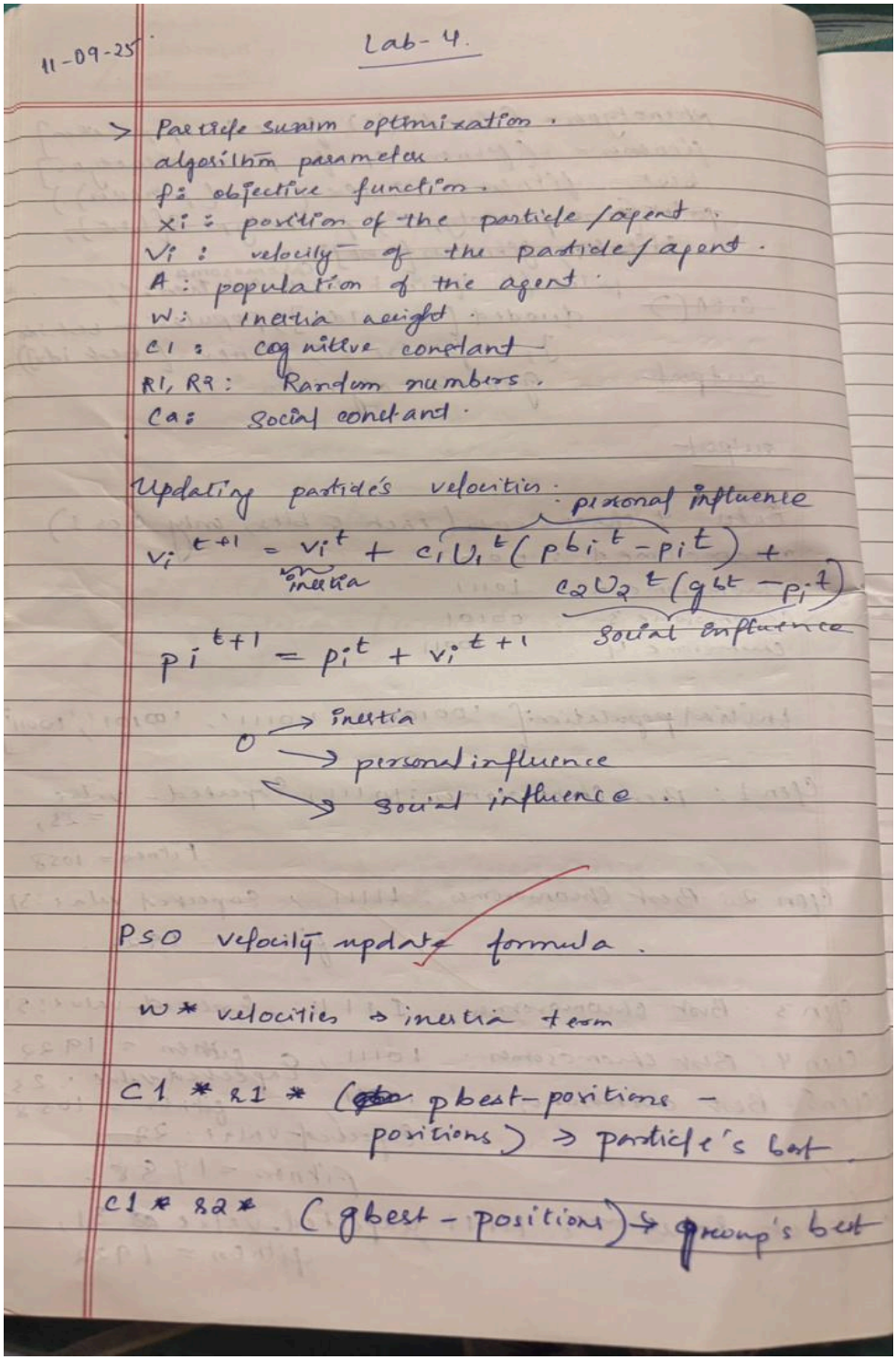
```
        if random.random() < MUTATION_RATE:
            mutated += '1' if bit == '0' else '0'
        else:
            mutated += bit
    return mutated

def genetic_algorithm():
    population = get_population_from_input()
    print(f"Initial Population: {population}")

    best_overall = None
    best_fitness = float('-inf')

    for generation in range(MAX_GENERATIONS):
        new_population = []
        while len(new_population) < POP_SIZE:
            parent1 = select(population)
            parent2 = select(population)
            child1, child2 = crossover(parent1, parent2)
            child1 = mutate(child1)
            child2 = mutate(child2)
            new_population.extend([child1, child2])

        population = new_population[:POP_SIZE]

        best = max(population, key=fitness)
        best_fit = fitness(best)
        if best_fit > best_fitness:
            best_fitness = best_fit
            best_overall = best

        print(f"Generation {generation + 1}: Best Chromosome = {best}, "
              f"Expressed Value = {gene_expression(best)}, Fitness = {best_fit}")

    print(f"\nBest solution after {MAX_GENERATIONS} generations: {best_overall} "
          f"with expressed value = {gene_expression(best_overall)} and fitness = {best_fitness}")

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

OUTPUT:

```
Enter 4 chromosomes (each 5 bits, only 0 or 1):
Chromosome 1: 11001
Chromosome 2: 01101
Chromosome 3: 00110
Chromosome 4: 10011
Initial Population: ['11001', '01101', '00110', '10011']
Generation 1: Best Chromosome = 11101, Expressed Value = 29, Fitness = 1682
Generation 2: Best Chromosome = 11011, Expressed Value = 27, Fitness = 1458
Generation 3: Best Chromosome = 11001, Expressed Value = 25, Fitness = 1250
Generation 4: Best Chromosome = 11010, Expressed Value = 26, Fitness = 1352
Generation 5: Best Chromosome = 11011, Expressed Value = 27, Fitness = 1458

Best solution after 5 generations: 11101 with expressed value = 29 and fitness = 1682
```

Problem 3

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.



Bafna Gold
Date: Page:

Position update formula

$$x_i(t+1) = x_i(t) + v_i(t+1)$$

positions t = velocities.

Job Scheduling - Application

Code:

```
import numpy as np
processing_time = input("Enter job processing times separated by spaces: ")
processing_times = np.array([float(x) for x in processing_time.strip().split()])

num_particles = 30
num_iterations = 100
w = 0.8
c1 = 1.5
c2 = 1.5
positions = np.random.uniform(0, 1, (num_particles, num_jobs))
velocities = np.random.uniform(-0.1, 0.1, (num_particles, num_jobs))

def fitness(priority_vector):
    job_order = np.argsort(priority_vector)
    completion_time = 0
    total_completion = 0
    for job in job_order:
        completion_time += processing_time[job]
```

```
        total_completion += completion_time
    return total_completion

pbest_positions = positions.copy()
pbest_scores = np.array([fitness(p)
    for p in positions])

gbest_idx = np.argmin(pbest_scores)
gbest_position = pbest_positions
    [gbest_idx].copy()
gbest_score = pbest_scores[gbest_idx]

for iteration in range(num_iterations):
r1 = np.random.rand(num_particles, num_jobs)
r2 = np.random.rand(num_particles, num_jobs)

velocities = (w * velocities + c1 * r1 *
        (pbest_positions - positions)
        + c2 * r2 * (gbest_positions
        - positions))
positions += velocities
positions = np.clip(positions, 0, 1)
scores = np.array([fitness(p) for p in
positions])
better_mask = scores < pbest_scores
pbest_positions[better_mask] =
    positions[better_mask]
pbest_scores[better_mask] =
    scores[better_mask]

min_idx = np.argmin(pbest_scores)
```

```
if pbest_scores = pbest_scores [min_idx]
    < gbest_score:
        gbest_score = pbest_scores [min_idx]
        gbest_position = pbest_positions [min_idx]
                                    .copy()
if (iteration + 1) % 10 == 0 or iterations
                                    == 0:
print(f" Iteration {iteration + 1}: Best
    Total completion time = {gbest_score: .2f}
    ")

best schedule = np.argsort (gbest_position)
print("\n Optimal job order (0-based):",
        best_schedule)
print(" Processing time in optimal order:",
        processing_times[best_schedule])
printf "Minimum total completion time:
                {gbest_score: .2f}")
```

Output:

```
Enter job processing times separated by spaces
: 1 2 3 4 5 6 17 8 9 10

Iteration 1: Best Total completion Time = 243.00
Iteration 2: Best- Total completion Time = 236.00
    .
    .
    .
Iteration 100: Best- Total completion Time = 236.00

Optimal job order (0-based): [0 1 2 3 4 5 7
                                8 9 6]
Processing time in order: [1. 2. 3. 4. 5. 6. 8. 9. 10. 17.]
Minimum Total completion time: 236.00
```

11/09

```
import numpy as np

num_drones = 5
area_size = 20
num_particles = 30
iterations = 10

w = 0.5
c1 = 1.5
c2 = 1.5

dim = num_drones * 2

particles = np.random.uniform(0, area_size, (num_particles, dim))
velocities = np.zeros((num_particles, dim))

pbest_positions = particles.copy()
pbest_scores = np.full(num_particles, -np.inf)

gbest_position = None
gbest_score = -np.inf

def fitness(position):
    x = np.sum(position)
    return (x*2 + 5*x + 20)

for iter in range(iterations):
    for i in range(num_particles):
        score = fitness(particles[i])

        if score > pbest_scores[i]:
            pbest_scores[i] = score
            pbest_positions[i] = particles[i].copy()

        if score > gbest_score:
            gbest_score = score
            gbest_position = particles[i].copy()

    r1, r2 = np.random.rand(), np.random.rand()
    for i in range(num_particles):
        velocities[i] = (w * velocities[i] +
                         c1 * r1 * (pbest_positions[i] - particles[i]) +
                         c2 * r2 * (gbest_position - particles[i]))
```

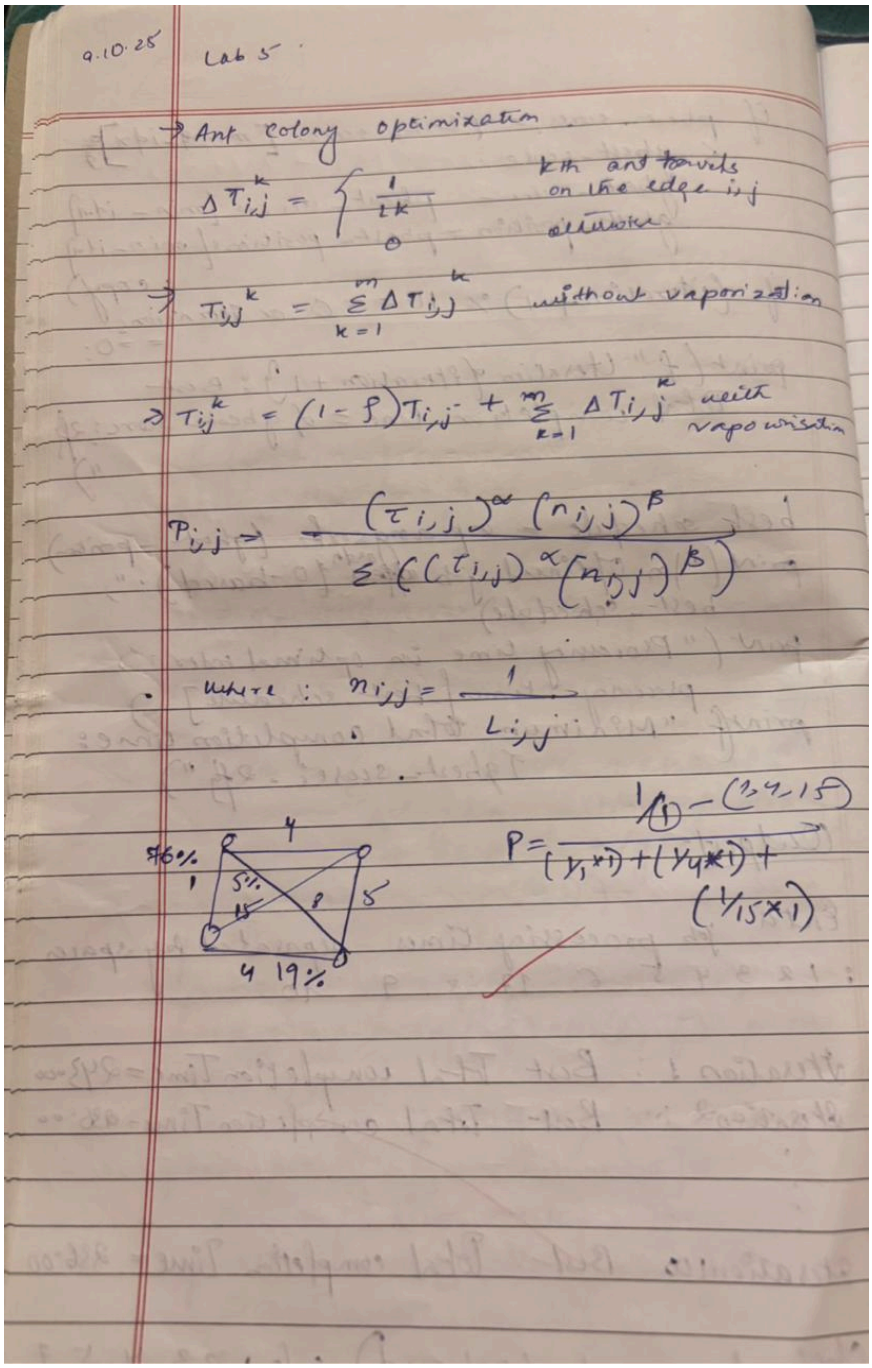
```
        particles[i] += velocities[i]
        particles[i] = np.clip(particles[i], 0, area_size-1)

    print(f"Iteration {iter}, Best Fitness: {gbest_score:.2f}")

print("\nOptimized drone waypoints (x,y):")
optimized_coords = gbest_position.reshape((num_drones, 2))
for idx, coord in enumerate(optimized_coords):
    print(f"Drone {idx+1}: {coord}")
```

OUTPUT:

```
Iteration 0, Best Fitness: 995.60
Iteration 1, Best Fitness: 995.60
Iteration 2, Best Fitness: 995.60
Iteration 3, Best Fitness: 1062.14
Iteration 4, Best Fitness: 1109.77
Iteration 5, Best Fitness: 1126.03
Iteration 6, Best Fitness: 1131.01
Iteration 7, Best Fitness: 1143.85
Iteration 8, Best Fitness: 1151.87
Iteration 9, Best Fitness: 1155.98

Optimized drone waypoints (x,y):
Drone 1: [19. 19.]
Drone 2: [13.47083229 13.14074042]
Drone 3: [ 9.53393196 19.        ]
Drone 4: [19. 19.]
Drone 5: [19.         12.13802119]
```

Problem 4

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



Bafna Gold
Date: Page:

Algorithm - ant optimization travel salesman

steps:

1. Initialization
- initialize pheromone levels T_{ij} on all edges to a small positive constant.

$$T_{ij} = T_0$$

2. Repeat for each iteration $t = 1, 2 \ldots T$
   a) for each $k = 1, 2, 3 \ldots$ an

$$P_{ij}^{k} = \frac{(T_{ij})^{\alpha} (n_{ij})^{\beta}}{\Sigma ((T_{ij})^{\alpha} (n_{ij})^{\beta})}$$

   b) evaluate the length of each ant's tour L_k.
3. update pheromenones.
   a) Evaporation:

$$T_{ij} = (1 - \rho) \cdot T_{ij}$$

   b) Deposit pheromone:
   if for each ant k and edge (i,j) in its tour.

$$T_{ij} = r_{ij} + \Delta t_{ij}^{k}$$

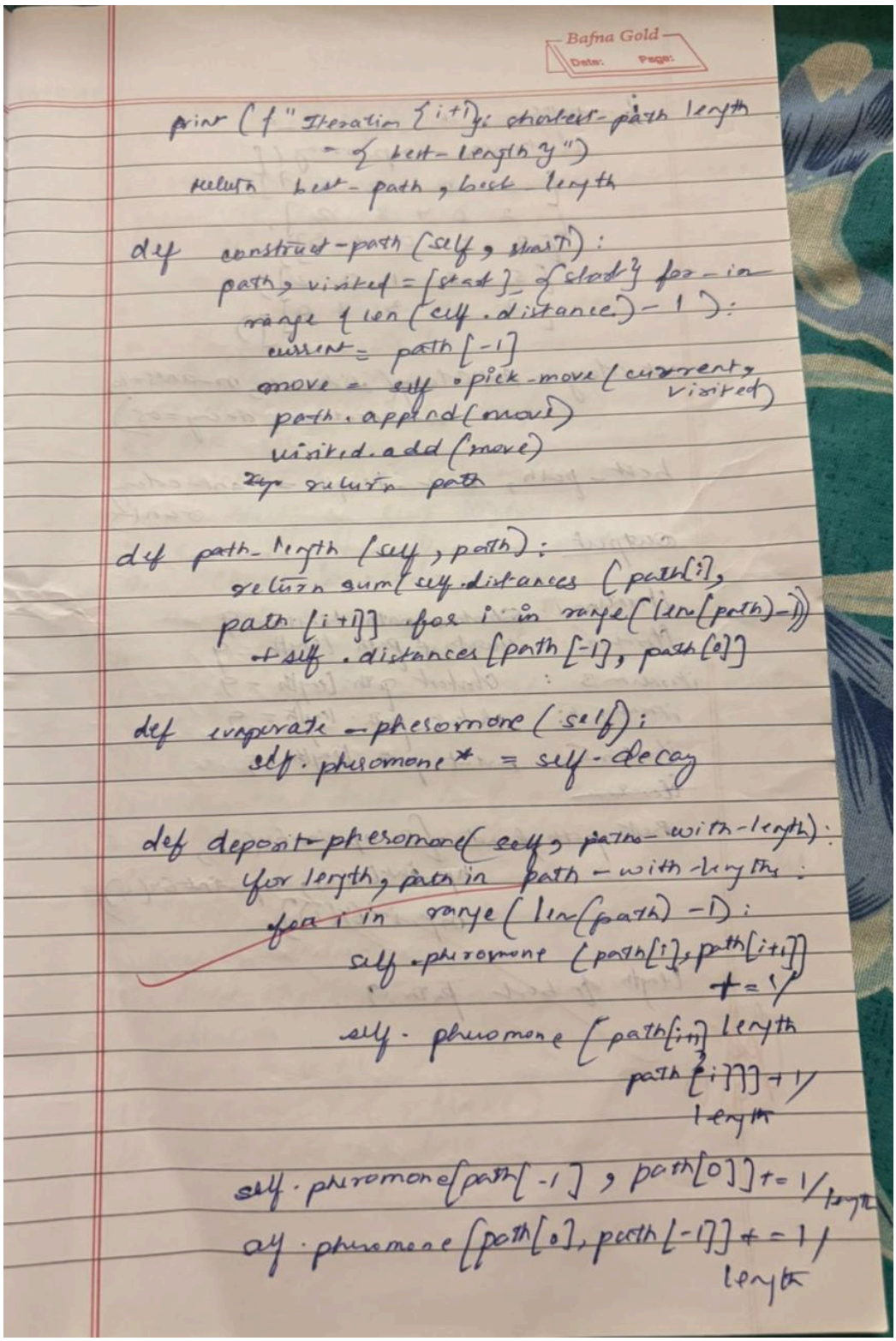
where

$$\Delta T_{ij}^{k} = \frac{Q}{L^{k}}$$

Q is constant representing the total pheromone amount deposited.

4. Keep track of the best tour found so far.
5. After 'T' iterations output the best tour.

```
import numpy as np

class AntColony:
    def __init__(self, distance, n_ants,
                 n_iteration, decay,
                 alpha=1, beta=2):
        self.distances = distances
        self.pheromone = np.ones(distances.shape)/
                         len(distances)
        self.n_ants = n_ants
        self.n_iterations = n_iterations
        self.decay = decay
        self.alpha = alpha
        self.beta = beta

    def run(self):
        best_path, best_length = None, float('inf')
        for i in range(self.n_iterations):
            paths = [self.construct_path(0) for _ in
                     range(self.n_ants)]
            paths_with_length = [(self.path_
                                  length(p), p)
                                 for p in paths]

            self.evaporate_pheromone()
            self.deposit_pheromone(paths_with_
                                   length)
            current_best = min(path_with_
                               length, key=lambda
                               x: x[0])
            if current_best[0] < best_length:
                best_length, best_path = current_
                                         best
```



```
print(f"Iteration {i+1}: shortest path length
    = {best_length}")
return best_path, best_length

def construct_path(self, start):
    path, visited = [start], {start} for _ in
    range(len(self.distances) - 1):
        current = path[-1]
        move = self.pick_move(current, visited)
        path.append(move)
        visited.add(move)
    return path

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

def evaporate_pheromone(self):
    self.pheromone *= self.decay

def deposit_pheromone(self, paths_with_length):
    for length, path in paths_with_length:
        for i in range(len(path) - 1):
            self.pheromone[path[i], path[i+1]]
                += 1/length
            self.pheromone[path[i], path[i+1]]
                += 1/length

self.pheromone[path[-1], path[0]] += 1/length
self.pheromone[path[0], path[-1]] += 1/length
```

```
if __name__ == "__main__":
    distances = np.array([
        [0, 8, 2, 5, 7],
        [2, 0, 4, 8, 2],
        [2, 4, 0, 1, 3],
        [5, 8, 1, 0, 2],
        [7, 2, 3, 2, 0]
    ])

ant_colony = AntColony(distances, n_ants=10,
                       n_iterations=5, decay=0.5)

best_path, best_length = ant_colony.run()
```

Output

Iteration 1: Shortest path length = 9
Iteration 2: shortest path length = 9
iteration 3 : Shortest path length = 9
iteration 4: shortest path length = 9
iteration 5: shortest path length = 9
~~iteration~~

Best path found : [0, np.int64(2), np.int64(3), np.int64(4), np.int64(1)]

length of best path : 9

```
import numpy as np
import random

NUM_ANTS =2
NUM_ITERATIONS = 50
ALPHA = 1.0
BETA = 5.0
EVAPORATION = 0.5
Q = 100

def input_matrix(name):
    print(f"Enter the {name} matrix row by row (space-separated). Type 'done' when finished:")
    matrix = []
    while True:
        row = input()
        if row.strip().lower() == 'done':
            break
        row_values = list(map(float, row.strip().split()))
        matrix.append(row_values)
    return np.array(matrix)

print("Input Cost Matrix (Distance Matrix):")
dist_matrix = input_matrix("cost")
NUM_CITIES = len(dist_matrix)

print("\nInput Initial Pheromone Matrix:")
pheromone = input_matrix("pheromone")

assert dist_matrix.shape == (NUM_CITIES, NUM_CITIES), "Cost matrix must be square."
assert pheromone.shape == (NUM_CITIES, NUM_CITIES), "Pheromone matrix must be
square."

best_distance = float('inf')
best_path = []

for iteration in range(NUM_ITERATIONS):
    all_paths = []
    all_distances = []

    for ant in range(NUM_ANTS):
        path = [random.randint(0, NUM_CITIES - 1)]

        while len(path) < NUM_CITIES:
            current_city = path[-1]
```

```
            probabilities = []

            for next_city in range(NUM_CITIES):
                if next_city not in path:
                    tau = pheromone[current_city][next_city] ** ALPHA
                    eta = (1 / dist_matrix[current_city][next_city]) ** BETA
                    probabilities.append(tau * eta)
                else:
                    probabilities.append(0)

            probabilities = np.array(probabilities)
            probabilities_sum = probabilities.sum()
            if probabilities_sum == 0:
                break
            probabilities /= probabilities_sum

            next_city = np.random.choice(range(NUM_CITIES), p=probabilities)
            path.append(next_city)

        if len(path) < NUM_CITIES:
            continue

        path.append(path[0])
        distance = sum(dist_matrix[path[i]][path[i + 1]] for i in range(NUM_CITIES))
        all_paths.append(path)
        all_distances.append(distance)

        if distance < best_distance:
            best_distance = distance
            best_path = path

    pheromone *= (1 - EVAPORATION)

    for i in range(len(all_paths)):
        for j in range(NUM_CITIES):
            from_city = all_paths[i][j]
            to_city = all_paths[i][j + 1]
            pheromone[from_city][to_city] += Q / all_distances[i]
            pheromone[to_city][from_city] += Q / all_distances[i]

    if iteration % 10 == 0 or iteration == NUM_ITERATIONS - 1:
        print(f"Iteration {iteration}: Best Distance = {best_distance:.2f}")

print("\nBest Path Found:")
```

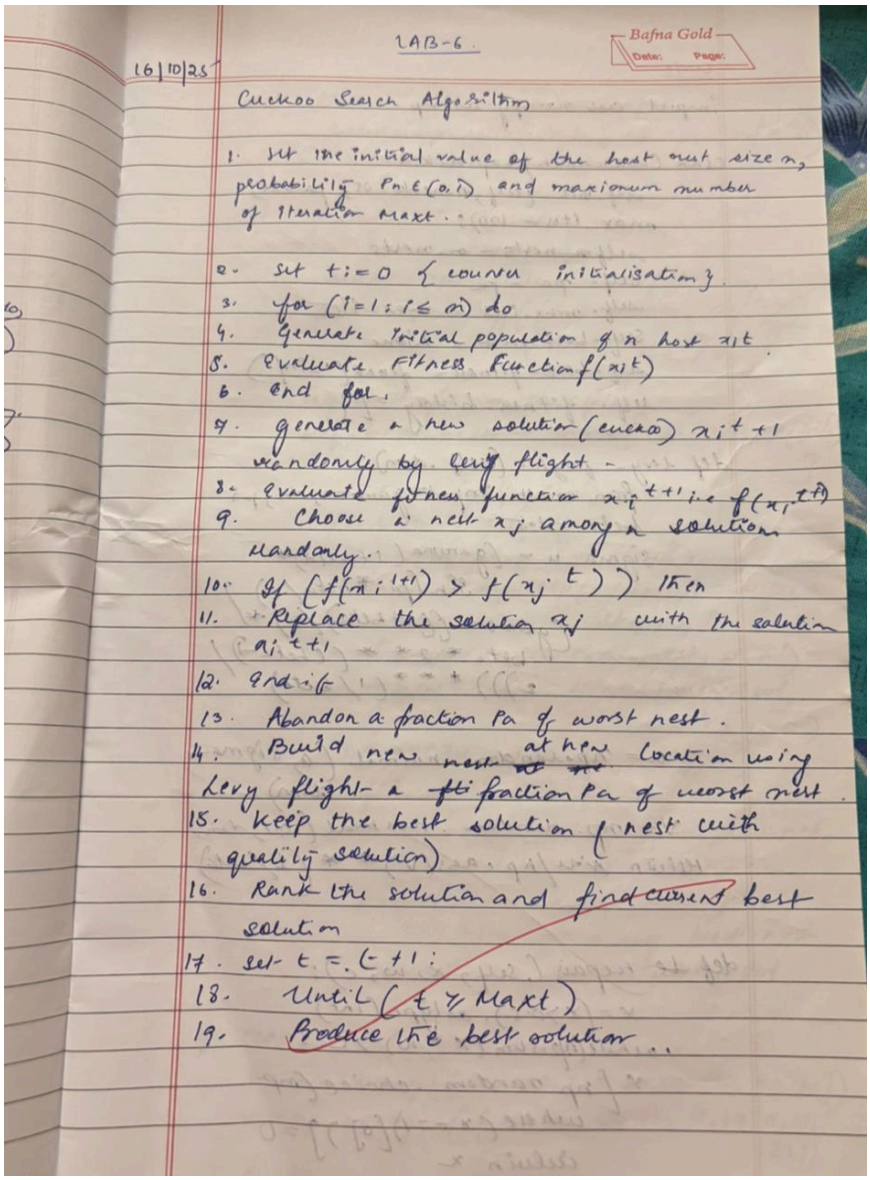
```
print(" -> ".join(map(str, best_path)))
print(f"Total Distance: {best_distance:.2f}")
```

OUTPUT:

```
0 5 15 4
5 0 4 8
15 4 0 1
4 8 1 0
done

Input Initial Pheromone Matrix:
Enter the pheromone matrix row by row (space-separated). Type
'done' when finished:
0 4 10 3
4 0 1 2
10 1 0 1
3 2 1 0
done
Iteration 0: Best Distance = 14.00
Iteration 10: Best Distance = 14.00
Iteration 20: Best Distance = 14.00
Iteration 30: Best Distance = 14.00
Iteration 40: Best Distance = 14.00
Iteration 49: Best Distance = 14.00

Best Path Found:
3 -> 2 -> 1 -> 0 -> 3
Total Distance: 14.00
```

Problem 5

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



LAB-6

16/10/25

Cuckoo Search Algorithm

1. Set the initial value of the host nest size n, probability Pa ∈ (0,1) and maximum number of iteration Maxt.
2. Set t:=0 {counter initialization}.
3. for (i=1; i≤ n) do
4. Generate initial population of n host xi t
5. Evaluate Fitness Function f(xi t)
6. End for.
7. generate a new solution (cuckoo) xi t+1 randomly by Levy flight.
8. Evaluate fitness function xi t+1 i.e f(xi t+1)
9. Choose a nest xj among n solution randomly.
10. If (f(xi t+1) > f(xj t)) then
11. Replace the solution xj with the solution ai t+1
12. End if
13. Abandon a fraction Pa of worst nest.
14. Build new nest at new location using Levy flight a fraction Pa of worst nest.
15. Keep the best solution (nest with quality solution)
16. Rank the solution and find current best solution
17. set t = t+1:
18. Until (t > Maxt)
19. Produce the best solution.

code.

```
import numpy as np

class CuckooSearchKnapsack:
    def __init__(self, n_nests=25, pa=0.25,
    max_iter=100):
        self.n_nests = n_nests
        self.pa = pa
        self.max_iter = max_iter
        self.best_nest = None
        self.best_fitness = float('-inf')
        self.fitness_history = []

def levy_flight(self, dim):
    from math import gamma, sin, pi
    beta = 1.5
    sigma_u = (gamma(1+beta)*
        sin(pi*beta/2)/
        (gamma((1+beta)/2)*
        beta*2**((beta-1)/
        2)))**(1/beta)

    u = np.random.normal(0, sigma_u,
                         dim)
    v = np.random.normal(0, 1, dim)
    return u/np.abs(v)**(1/beta)

def repair(self, x, w, c):
    x = (x > 0.5).astype(int)
    while(np.sum(x*w) > c:
        x[np.random.choice(np.
            where(x==1)[0])]=0
    return x
```

```
def fitness(self, x, v, w, c):
    return np.sum(x*v) if np.sum(x*w)
    <=c else 0

def optimize(self, v, w, c):
    n_items = len(v)
    nests = np.random.rand(self.n, n_
    fits = np.array([self.fitness(self.repair
        (n,w,c),v,w,c) for n in
        nests])
    self.best_fit = max(fits)
    self.best_nest = nests[np.argmax(fits)
        ]

    for _ in range(self.iter):
        for i in range(self.n):
            step = self.levy(n_items)
            new_nest = np.clip(nests[i] + 0.01*
                step, 0, 1)
            new_sol = self.repair(new_nest, w, c)
            fit_new = self.fitness(new_sol,
                v, w, c)
            j = np.random.randint(self.n)
            if fit_new > fits[j]:
                nests[j], fits[j] = new_nest,
                    fit_new
            if fit_new > self.best_fit:
                self.best_fit, self.best_nest =
                    fit_new, new_sol

if __name__ == "__main__":
    v = np.array([60, 100, 120, 80, 50, 70, 90, 100])
    w = np.array([10, 20, 30, 15, 12, 18, 8, 25,
        22])
    c = 80
```

```
cs = CuckooSearch(30, 0.25, 100)
sol, val = cs.optimize(v, wt, c)
sel = np.where(sol == 1)[0]
print("selected:", sel+1)
print(f"Value: ${val},
      Weight: {sum(sel * w)}
      kg")
```

Output:

```
=== Cuckoo Search for 0/1 Knapsack
    Problem ===

Number of items: 8
Knapsack capacity: 80
Items (value, weight):
  Item 1: ($60, 10kg)
  Item 2: ($100, 20kg)
  Item 3: ($120, 30kg)
  Item 4: ($80  15kg)
  Item 5: ($50  12kg)
  Item 6: ($70  18kg)
  Item 7: ($90  25kg)
  Item 8: ($110 22kg)

--- solution ---

Selected items: [1 2 4 5 8]
Total value: $400
Total weight: 79kg / 80kg
```

Item Details:

Item 1: Value = $60, weight = 10 kg
Item 2: Value = $100, weight = 20 kg
Item 3: value = $80, weight = 15 kg
Item 4: value = $50, weight = 12 kg
Item 5: value = $110, weight = 22 kg

Convergence (every 20 iterations):

Iter 1: Best value: $370
Iter 21: Best Value: $400
Iter 41: Best Value: $400
Iter 61: Best Value: $400
Iter 81: Best Value: $400

16/10

```
import random
import math

weights = [10, 20, 30, 40, 15, 25, 35]
values = [60, 100, 120, 240, 80, 150, 200]
capacity = 100

n_items = len(weights)
n_nests = 15
max_iter = 50
pa = 0.25

def fitness(solution):
    total_weight = sum(w for w, s in zip(weights, solution) if s == 1)
    total_value = sum(v for v, s in zip(values, solution) if s == 1)
    if total_weight > capacity:
        return 0
    else:
        return total_value

def generate_nest():
    return [random.randint(0, 1) for _ in range(n_items)]

def levy_flight(Lambda=1.5):
    sigma_u = (math.gamma(1 + Lambda) * math.sin(math.pi * Lambda / 2) /
               (math.gamma((1 + Lambda) / 2) * Lambda * 2 ** ((Lambda - 1) / 2))) ** (1 / Lambda)
    u = random.gauss(0, sigma_u)
    v = random.gauss(0, 1)
    step = u / (abs(v) ** (1 / Lambda))
    return step

def get_cuckoo(nest, best_nest):
    new_nest = []
    for xi, bi in zip(nest, best_nest):
        step = levy_flight()
        val = xi + step * (xi - bi)

        s = 1 / (1 + math.exp(-val))
        new_val = 1 if s > 0.5 else 0
        new_nest.append(new_val)
    return new_nest

def cuckoo_search():
    nests = [generate_nest() for _ in range(n_nests)]
```

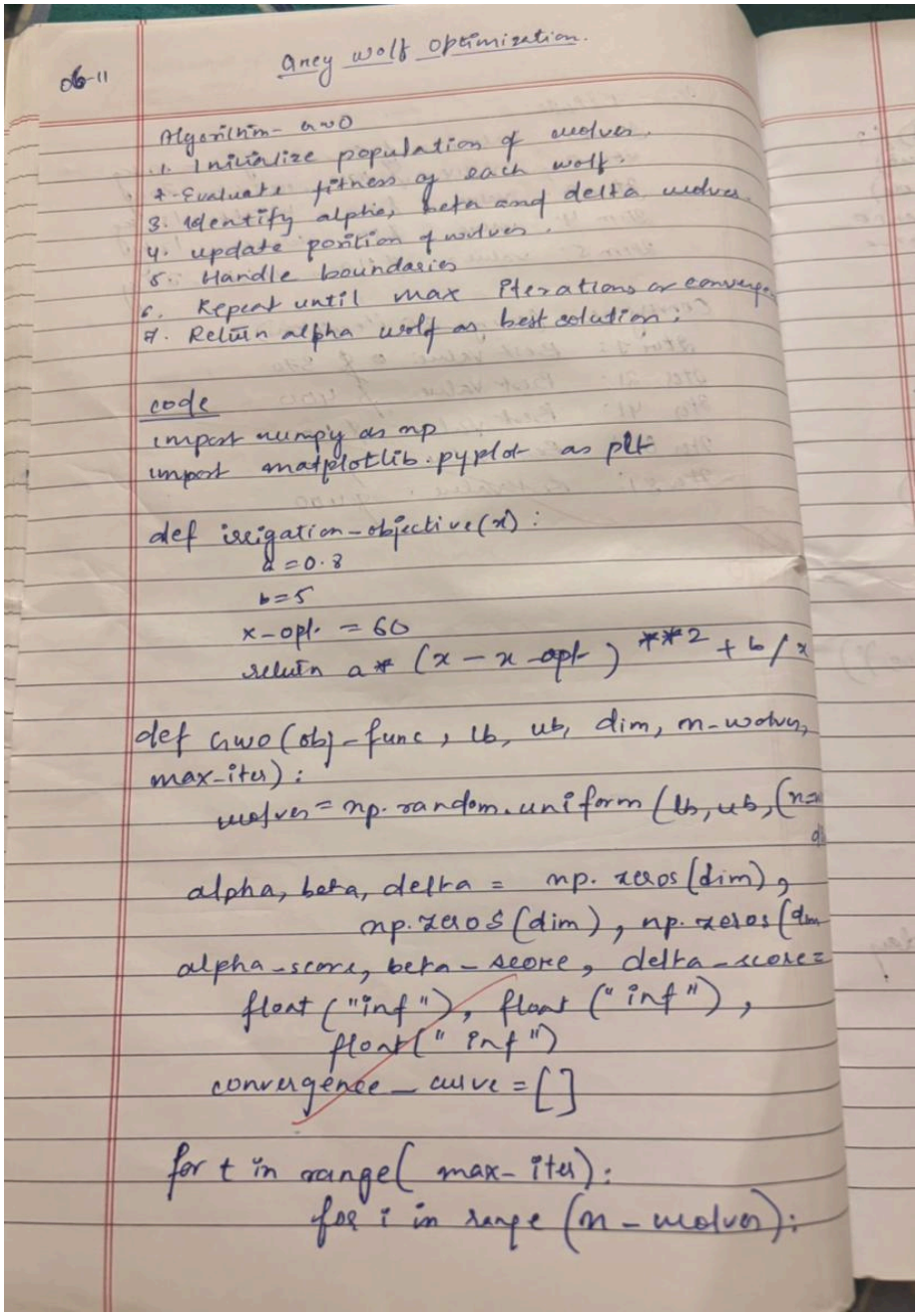
```
    fitness_values = [fitness(nest) for nest in nests]

    best_index = fitness_values.index(max(fitness_values))
    best_nest = nests[best_index][:]
    best_fitness = fitness_values[best_index]

    for iteration in range(1, max_iter + 1):
        for i in range(n_nests):
            new_nest = get_cuckoo(nests[i], best_nest)
            new_fitness = fitness(new_nest)
            if new_fitness > fitness_values[i]:
                nests[i] = new_nest
                fitness_values[i] = new_fitness

        for i in range(n_nests):
            if random.random() < pa:
                nests[i] = generate_nest()
                fitness_values[i] = fitness(nests[i])

        current_best_index = fitness_values.index(max(fitness_values))
        current_best_fitness = fitness_values[current_best_index]

        if current_best_fitness > best_fitness:
            best_fitness = current_best_fitness
            best_nest = nests[current_best_index][:]

        if iteration % 10 == 0:
            print(f"Iteration {iteration}: Best value so far = {best_fitness}")

    return best_nest, best_fitness

if __name__ == "__main__":
    best_solution, best_value = cuckoo_search()
    total_weight = sum(w for w, s in zip(weights, best_solution) if s == 1)
    print(f"\nBest packing solution (1 = selected): {best_solution}")
    print(f"Total value of supplies packed: {best_value}")
    print(f"Total weight: {total_weight}")
```

OUTPUT:

```
Iteration 10: Best value so far = 570
Iteration 20: Best value so far = 570
Iteration 30: Best value so far = 590
Iteration 40: Best value so far = 590
Iteration 50: Best value so far = 590

Best packing solution (1 = selected): [0, 0, 0, 1, 0, 1, 1]
Total value of supplies packed: 590
Total weight: 100
```

Program 6

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.



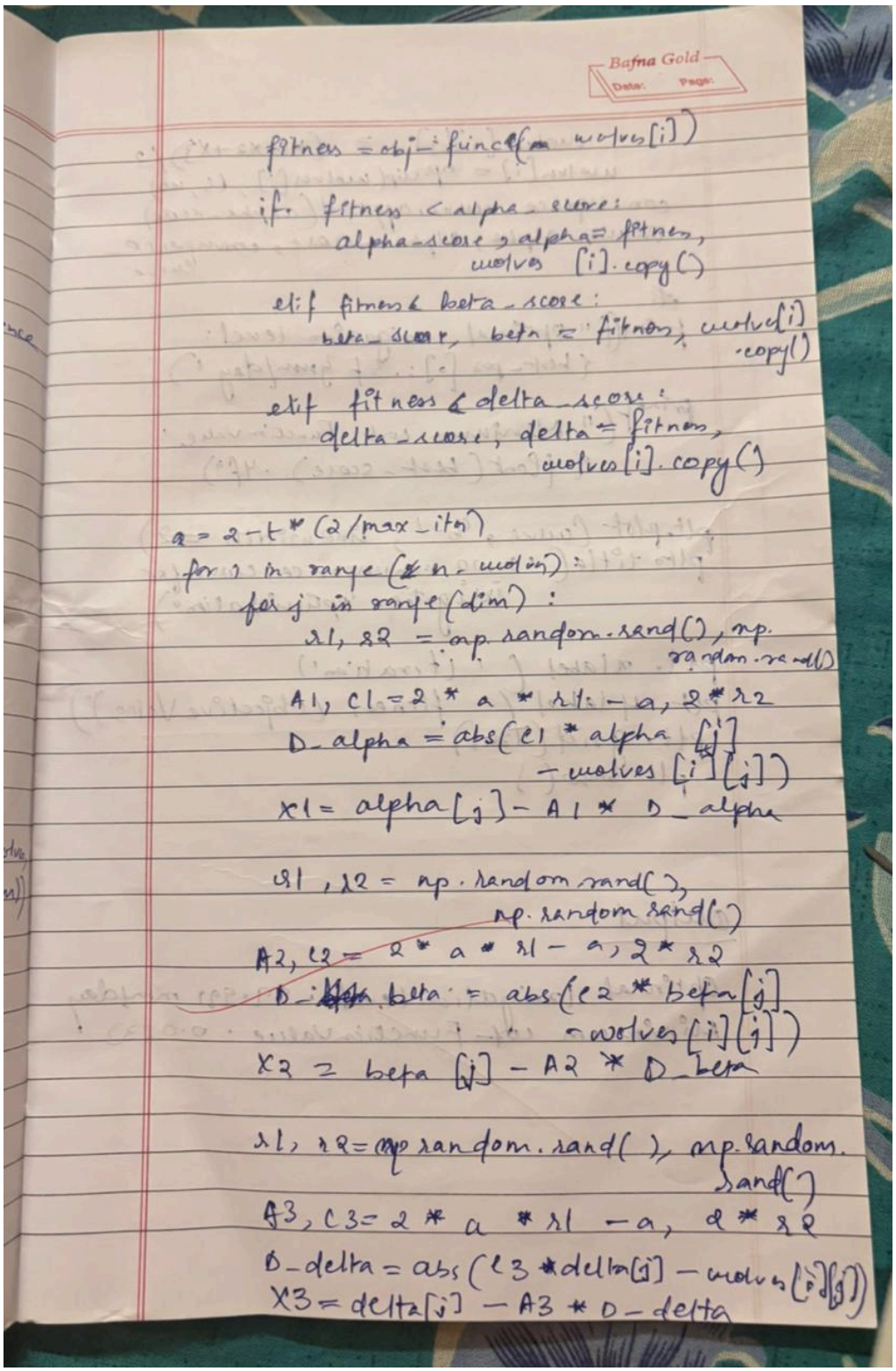
06-11

Grey wolf optimization.

Algorithm- GWO
1. Initialize population of wolves.
2. Evaluate fitness of each wolf.
3. Identify alpha, beta and delta wolves.
4. update position of wolves.
5. Handle boundaries
6. Repeat until max Iterations or convergence
7. Return alpha wolf as best solution.

code

```
import numpy as np
import matplotlib.pyplot as plt

def irrigation_objective(x):
    a=0.8
    b=5
    x_opt = 60
    return a * (x - x_opt) **2 + b / x

def gwo(obj_func, lb, ub, dim, n_wolves,
max_iter):
    wolves = np.random.uniform(lb, ub, (n_
                                        d
    alpha, beta, delta = np.zeros(dim),
                np.zeros(dim), np.zeros(dim
    alpha_score, beta_score, delta_score=
        float("inf"), float("inf"),
            float("inf")
    convergence_curve = []

    for t in range(max_iter):
        for i in range(n_wolves):
```


Bafna Gold
Date: Page:
fitness = obj_func(wolves[i])
if fitness < alpha_score:
alpha_score, alpha = fitness, wolves[i].copy()
elif fitness < beta_score:
beta_score, beta = fitness, wolves[i].copy()
elif fitness < delta_score:
delta_score, delta = fitness, wolves[i].copy()
a = 2 - t * (2/max_iter)
for i in range(n_wolves):
for j in range(dim):
r1, r2 = np.random.rand(), np.random.rand()
A1, C1 = 2 * a * r1 - a, 2 * r2
D_alpha = abs(C1 * alpha[j] - wolves[i][j])
X1 = alpha[j] - A1 * D_alpha
r1, r2 = np.random.rand(), np.random.rand()
A2, C2 = 2 * a * r1 - a, 2 * r2
D_beta = abs(C2 * beta[j] - wolves[i][j])
X2 = beta[j] - A2 * D_beta
r1, r2 = np.random.rand(), np.random.rand()
A3, C3 = 2 * a * r1 - a, 2 * r2
D_delta = abs(C3 * delta[j] - wolves[i][j])
X3 = delta[j] - A3 * D_delta

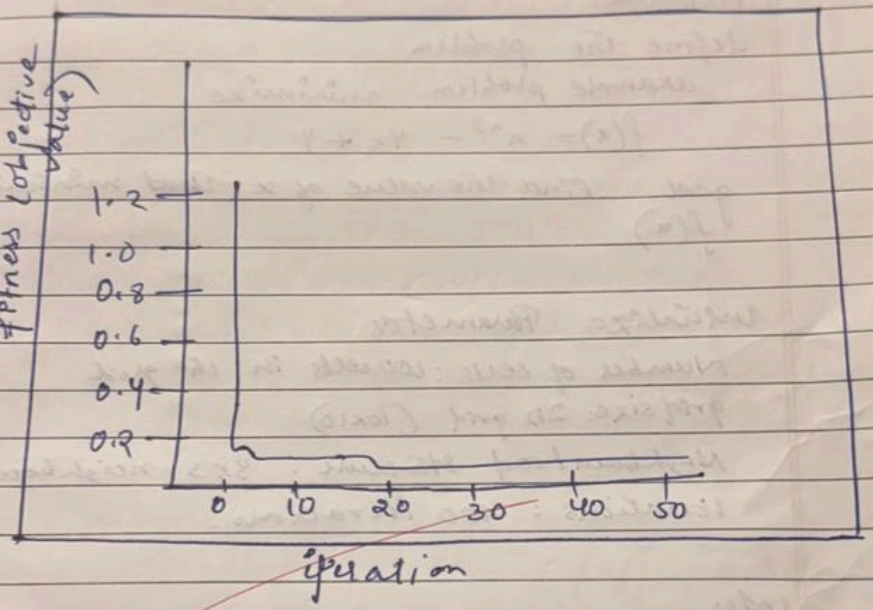
```
        wolves[i][j] = (X1+X2+X3)/3
        wolves[i] = np.clip(wolves[i], lb, ub)
    convergence_curve.append(alpha_score)
return alpha, alpha_score, convergence_curve

print(f"Optimal irrigation level:
      {best_pos[0]:.3f} mm/day")
print(f"Minimum cost Function Value:
      {float(best_score):.4f}")

plt.plot(curve, 'b-', linewidth=2)
plt.title('GWO convergence curve for
          irrigation optimization')
plt.xlabel('Iteration')
plt.ylabel('Fitness (Objective Value)')
plt.grid(True)
plt.show()
```

Output:

Optimal Irrigation Level: 59.991 mm/day
Minimum cost Function Value: 0.0833

Bafna Gold
Date: Page:

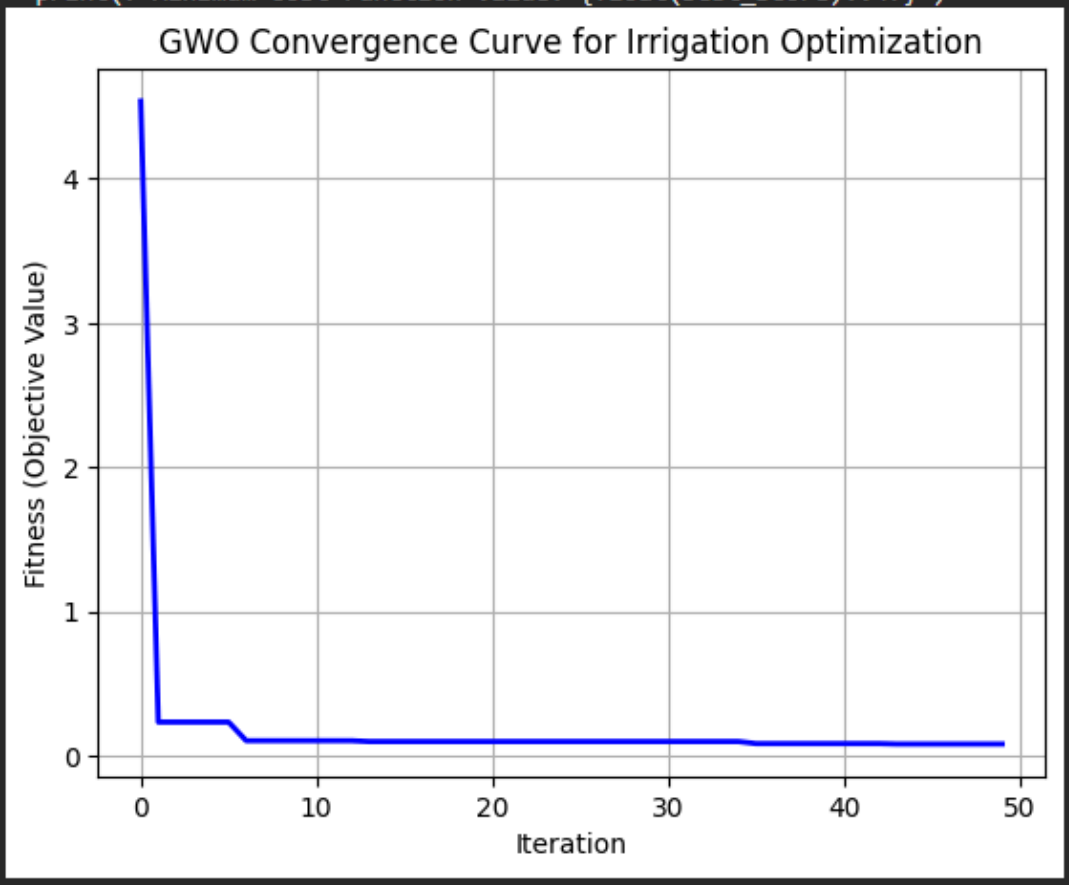
Q WO convergence curve for irrigation optimization



```
import numpy as np
import matplotlib.pyplot as plt

def irrigation_objective(x):
    # Parameters for irrigation problem
    a = 0.8      # yield penalty factor
    b = 5        # water cost factor
    x_opt = 60   # optimal irrigation level (e.g., mm/day)
    return a * (x - x_opt)**2 + b / x

def GWO(obj_func, lb, ub, dim, n_wolves, max_iter):
    # Initialize wolves randomly within bounds
    wolves = np.random.uniform(lb, ub, (n_wolves, dim))
    alpha, beta, delta = np.zeros(dim), np.zeros(dim), np.zeros(dim)
    alpha_score, beta_score, delta_score = float("inf"), float("inf"), float("inf")

    convergence_curve = []

    for t in range(max_iter):
        for i in range(n_wolves):
            fitness = obj_func(wolves[i])
            # Update alpha, beta, delta
            if fitness < alpha_score:
                alpha_score, alpha = fitness, wolves[i].copy()
            elif fitness < beta_score:
                beta_score, beta = fitness, wolves[i].copy()
            elif fitness < delta_score:
                delta_score, delta = fitness, wolves[i].copy()

        # Linearly decreasing 'a' from 2 to 0
        a = 2 - t * (2 / max_iter)

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

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

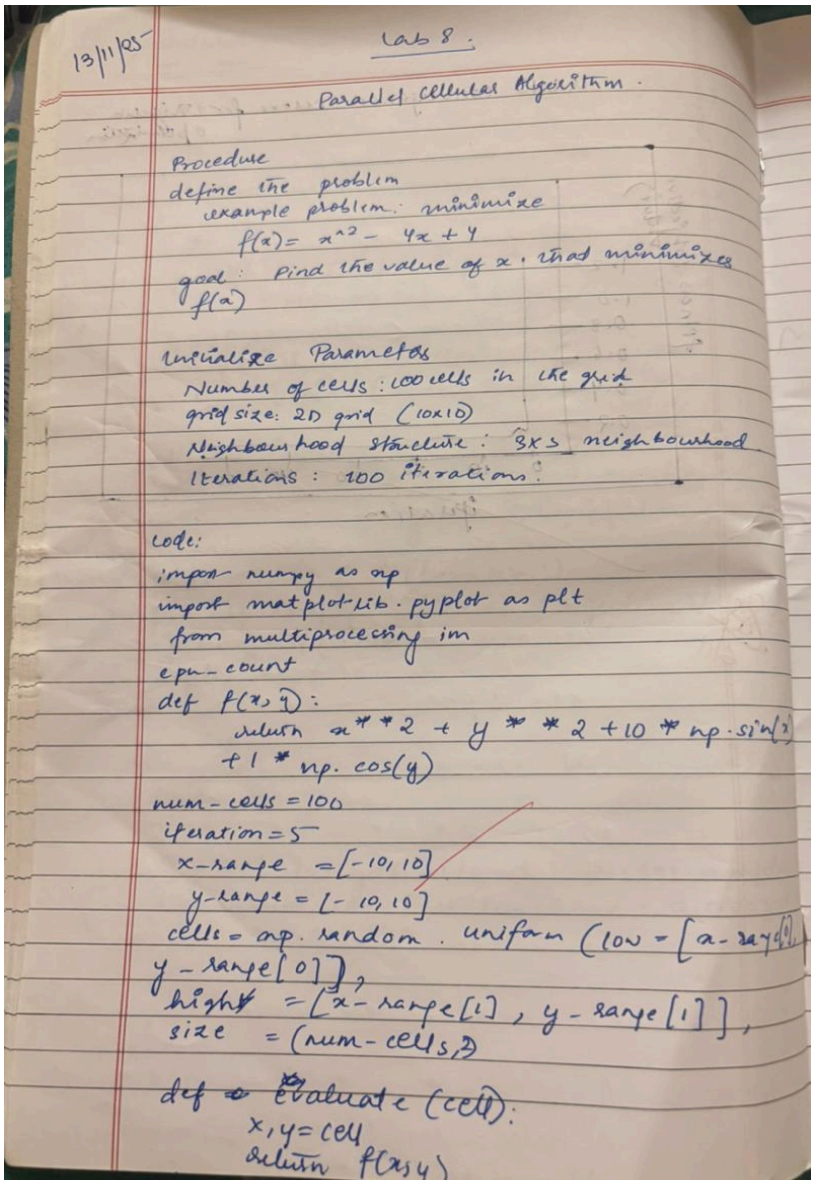
```
            r1, r2 = np.random.rand(), np.random.rand()
            A3, C3 = 2 * a * r1 - a, 2 * r2
            D_delta = abs(C3 * delta[j] - wolves[i][j])
            X3 = delta[j] - A3 * D_delta

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

        # Boundary handling
        wolves[i] = np.clip(wolves[i], lb, ub)

    convergence_curve.append(alpha_score)

  return alpha, alpha_score, convergence_curve

dim = 1
lb, ub = 10, 100     # irrigation limits (mm/day)
n_wolves = 20
max_iter = 50

best_pos, best_score, curve = GWO(irrigation_objective, lb, ub, dim, n_wolves, max_iter)
print(f"Optimal Irrigation Level: {best_pos[0]:.3f} mm/day")
print(f"Minimum Cost Function Value: {float(best_score):.4f}")

plt.plot(curve, 'b-', linewidth=2)
plt.title('GWO Convergence Curve for Irrigation Optimization')
plt.xlabel('Iteration')
plt.ylabel('Fitness (Objective Value)')
plt.grid(True)
plt.show()
```
OUTPUT:

```
Optimal Irrigation Level: 59.990 mm/day
Minimum Cost Function Value: 0.0834
/tmp/ipython-input-2606069732.py:76: DeprecationWarning: Conversion of an array w
  print(f"Minimum Cost Function Value: {float(best_score):.4f}")
```



Program 7

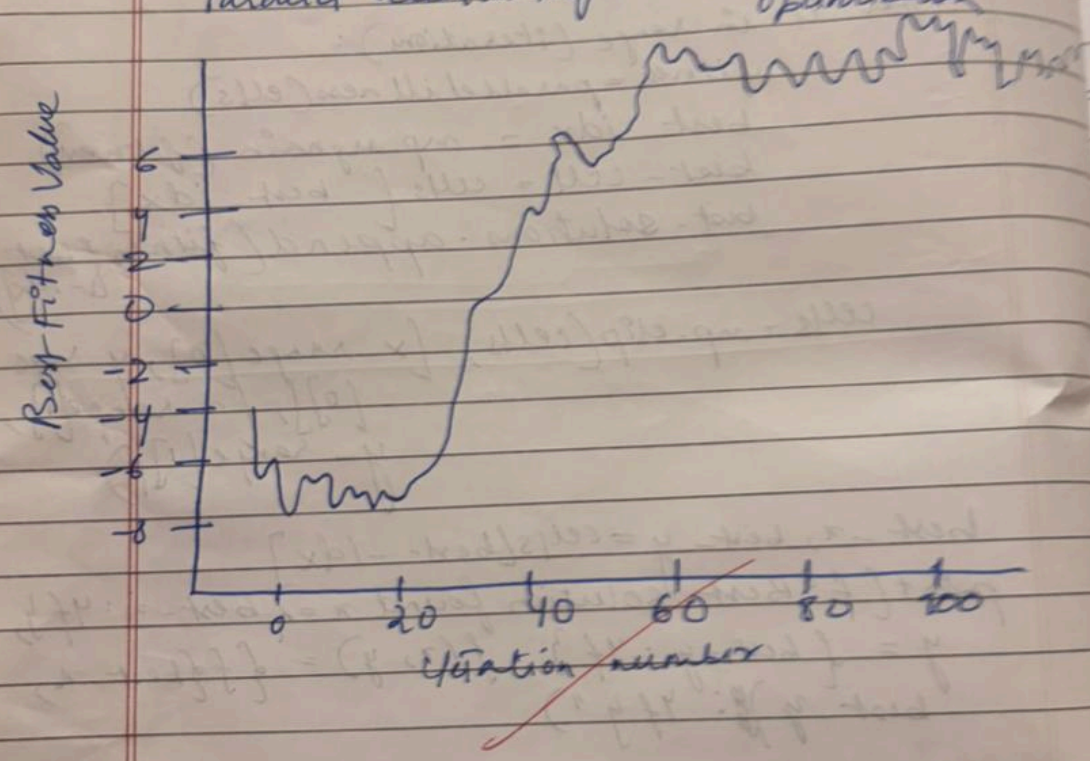
Parallel Cellular Algorithms are inspired by the functioning of biological cells that operate in a highly parallel and distributed manner. These algorithms leverage the principles of cellular automata and parallel computing to solve complex optimization problems efficiently. Each cell represents a potential 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.



```
def parallel_fitness(cells):
    with Pool(cpu_count()) as p:
        fitness = p.starmap(f, [(x,y) for x,y in cells])
    return np.array(fitness)

best_solutions = []

for it in range(iterations):
    fitness = parallel_fitness(cells)
    best_idx = np.argmin(fitness)
    best_cell = cells[best_idx]
    best_solutions.append(fitness[best_idx])

    cells = np.clip(cells, [x_range[0], y_range[0]], [x_range[1], y_range[1]])

best_x, best_y = cells[best_idx]
print(f"Best solution found x = {best_x:.4f}, y = {best_y:.4f}, f(x, y) = {f(best_x, best_y):.4f}")

plt.plot(best_solutions)
plt.title("parallel cellular algorithm - function optimization")
plt.xlabel("iteration")
plt.ylabel("Best Fitness Value")
plt.grid(True)
plt.show()
```

Output-

Best solution found: x = 0.0632,
y = 0.6227, f(x,y) = 6.7094



```
#PCA
import numpy as np
import random

num_customers = int(input("Enter number of customers (excluding depot): "))
num_vehicles = int(input("Enter number of vehicles: "))

print("\nEnter the distance matrix (including depot 0):")
print(f"Matrix should be {num_customers + 1} x {num_customers + 1}")
distance_matrix = []

for i in range(num_customers + 1):
    row = list(map(int, input(f"Row {i+1}: ").split()))
    distance_matrix.append(row)

distance_matrix = np.array(distance_matrix)

rows = int(input("\nEnter number of grid rows: "))
cols = int(input("Enter number of grid columns: "))
grid_dim = (rows, cols)
population_size = rows * cols

num_generations = int(input("\nEnter number of generations: "))

def generate_individual():
    perm = list(range(1, num_customers + 1))
    random.shuffle(perm)
    return perm

population = [generate_individual() for _ in range(population_size)]

def fitness(individual):
    split_points = np.linspace(0, num_customers, num_vehicles + 1, dtype=int)
    total_distance = 0
    for i in range(num_vehicles):
        route = [0] + individual[split_points[i]:split_points[i+1]] + [0]
        for j in range(len(route) - 1):
            total_distance += distance_matrix[route[j], route[j+1]]
    return total_distance

def get_neighbors(idx):
    r, c = divmod(idx, grid_dim[1])
    neighbors = []
    for dr in [-1, 0, 1]:
```

```
        for dc in [-1, 0, 1]:
            nr, nc = r + dr, c + dc
            if 0 <= nr < grid_dim[0] and 0 <= nc < grid_dim[1]:
                n_idx = nr * grid_dim[1] + nc
                if n_idx != idx:
                    neighbors.append(n_idx)
    return neighbors

def crossover(parent1, parent2):
    size = len(parent1)
    a, b = sorted(random.sample(range(size), 2))
    child = [None] * size
    child[a:b] = parent1[a:b]

    pointer = b
    for gene in parent2[b:] + parent2[:b]:
        if gene not in child:
            if pointer == size:
                pointer = 0
            child[pointer] = gene
            pointer += 1
    return child

def mutate(individual):
    a, b = random.sample(range(len(individual)), 2)
    individual[a], individual[b] = individual[b], individual[a]
    return individual

def pca_iteration(pop):
    new_pop = pop.copy()
    for idx in range(len(pop)):
        neighbors = get_neighbors(idx)
        partner_idx = random.choice(neighbors)
        parent1 = pop[idx]
        parent2 = pop[partner_idx]

        child = crossover(parent1, parent2)
        if random.random() < 0.2:
            child = mutate(child)

        if fitness(child) < fitness(pop[idx]):
            new_pop[idx] = child
    return new_pop
```

```
for gen in range(num_generations):
    population = pca_iteration(population)
    best_fitness = min(fitness(ind) for ind in population)
    print(f"Generation {gen+1}: Best total distance = {best_fitness}")

best_individual = min(population, key=fitness)
print("\nBest route assignment (split evenly):")
split_points = np.linspace(0, num_customers, num_vehicles + 1, dtype=int)
for i in range(num_vehicles):
    route = [0] + best_individual[split_points[i]:split_points[i+1]] + [0]
    print(f"Vehicle {i+1} route: {route}")
print(f"Total distance: {fitness(best_individual)}")
```

OUTPUT:

```
Enter number of customers (excluding depot): 3
Enter number of vehicles: 2

Enter the distance matrix (including depot 0):
Matrix should be 4 x 4
Row 1: 0 2 9 10
Row 2: 2 0 6 4
Row 3: 9 6 0 8
Row 4: 10 4 8 0

Enter number of grid rows: 3
Enter number of grid columns: 3

Enter number of generations: 10
Generation 1: Best total distance = 31
Generation 2: Best total distance = 31
Generation 3: Best total distance = 31
Generation 4: Best total distance = 31
Generation 5: Best total distance = 31
Generation 6: Best total distance = 31
Generation 7: Best total distance = 31
Generation 8: Best total distance = 31
Generation 9: Best total distance = 31
Generation 10: Best total distance = 31

Best route assignment (split evenly):
Vehicle 1 route: [0, 1, 0]
Vehicle 2 route: [0, 2, 3, 0]
Total distance: 31
```