|  |
| --- |
| **VISVESVARAYA TECHNOLOGICAL UNIVERSITY**  **“JnanaSangama”, Belgaum -590014, Karnataka.**    **LAB RECORD**  **Bio Inspired Systems (23CS5BSBIS)**  ***Submitted by***  **Manav Kumar (1BM22CS348)**  ***in partial fulfilment 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**  **Sep-2024 to Jan-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 **Manav Kumar (1BM22CS348),** who is bonafide student of **B.M.S. College of Engineering.** It is in partial fulfilment 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.

|  |  |
| --- | --- |
| Supreeth S  Assistant Professor  Department of CSE, BMSCE | Dr. Kavitha Sooda  Professor & HOD  Department of CSE, BMSCE |

**Index**

|  |  |  |  |
| --- | --- | --- | --- |
| **Sl.**  **No.** | **Date** | **Experiment Title** | **Page No.** |
| 1 | 09/10/24 | Genetic Algorithm for Optimization Problems | 4-9 |
| 2 | 13/11/24 | Particle Swarm Optimization for Function  Optimization | 10-14 |
| 3 | 23/10/24 | Ant Colony Optimization for the  Traveling Salesman Problem | 15-21 |
| 4 | 27/11/24 | Cuckoo Search (CS) | 22-27 |
| 5 | 20/11/24 | Grey Wolf Optimizer (GWO) | 28-33 |
| 6 | 16/12/24 | Parallel Cellular Algorithm and Program | 34-39 |
| 7 | 16/12/24 | Optimization via Gene Expression Algorithm | 40-46 |

Github Link: https://github.com/Manav-Kumar123/MANAV\_1BM22CS348\_BIS

**Laboratory Program - 1**

**Genetic Algorithm for Optimization Problems**

Genetic Algorithms (GA) are inspired by the process of natural selection and genetics, where

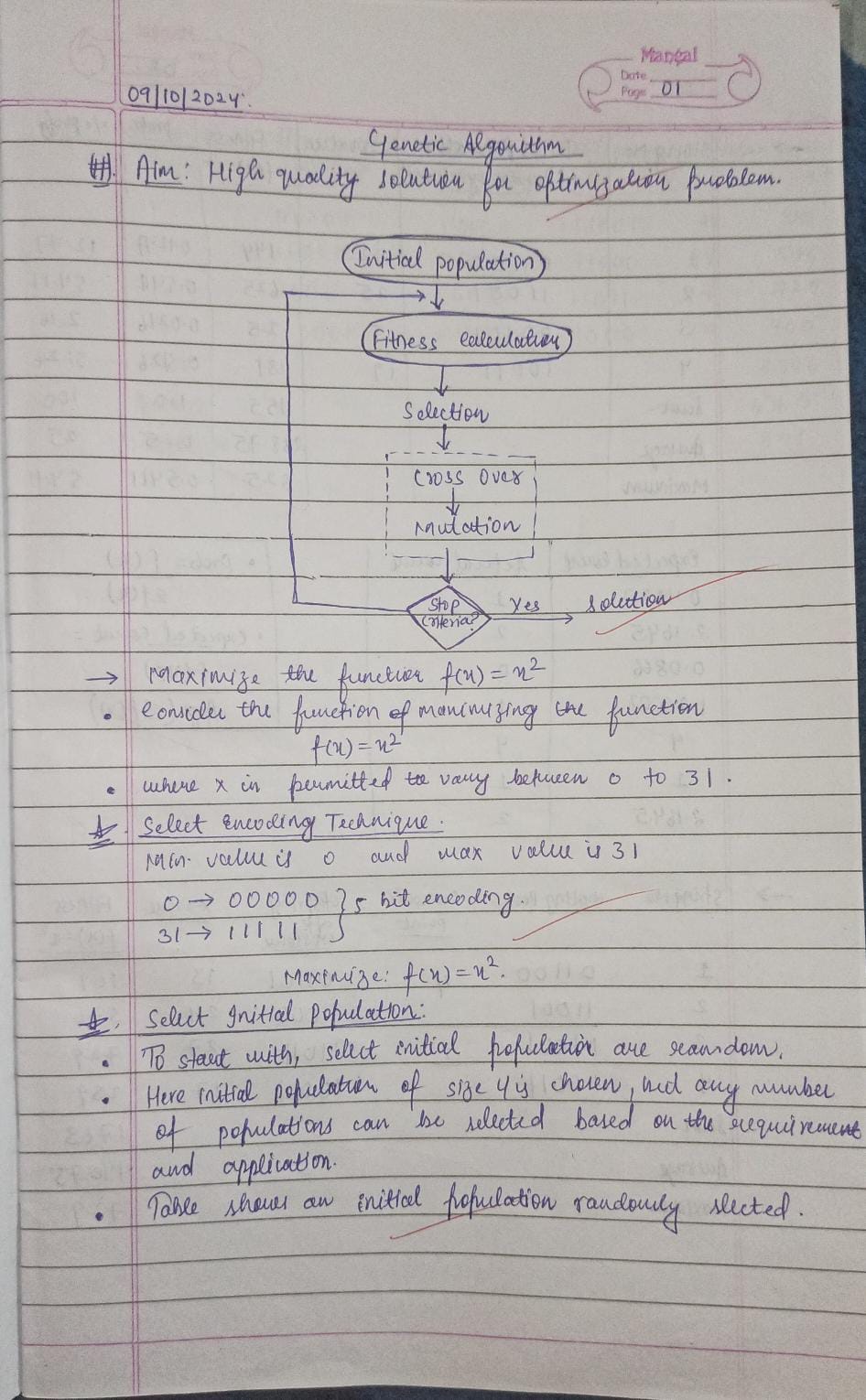
the fittest individuals are selected for reproduction to produce the next generation. GAs are

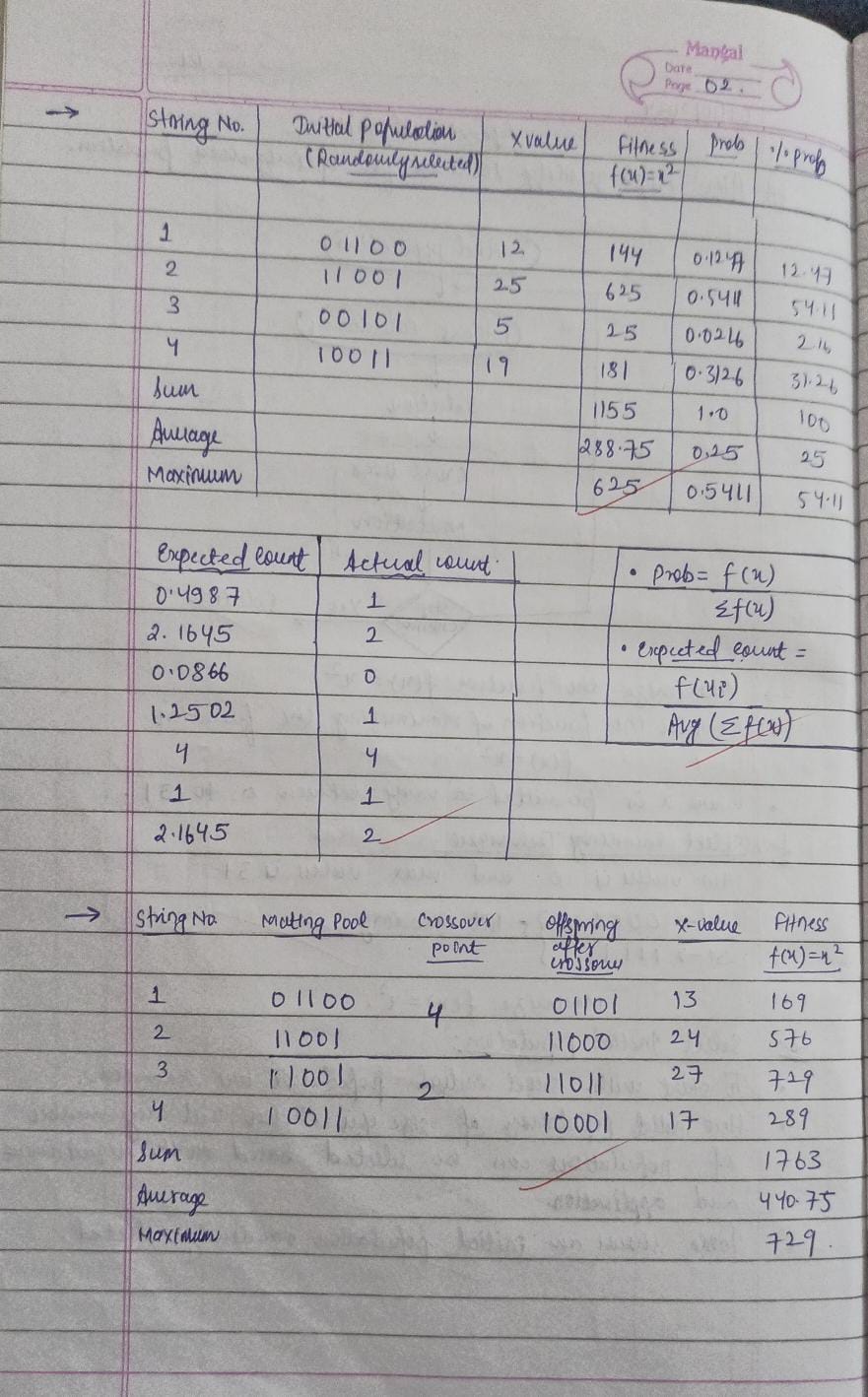
widely used for solving optimization and search problems. Implement a Genetic Algorithm

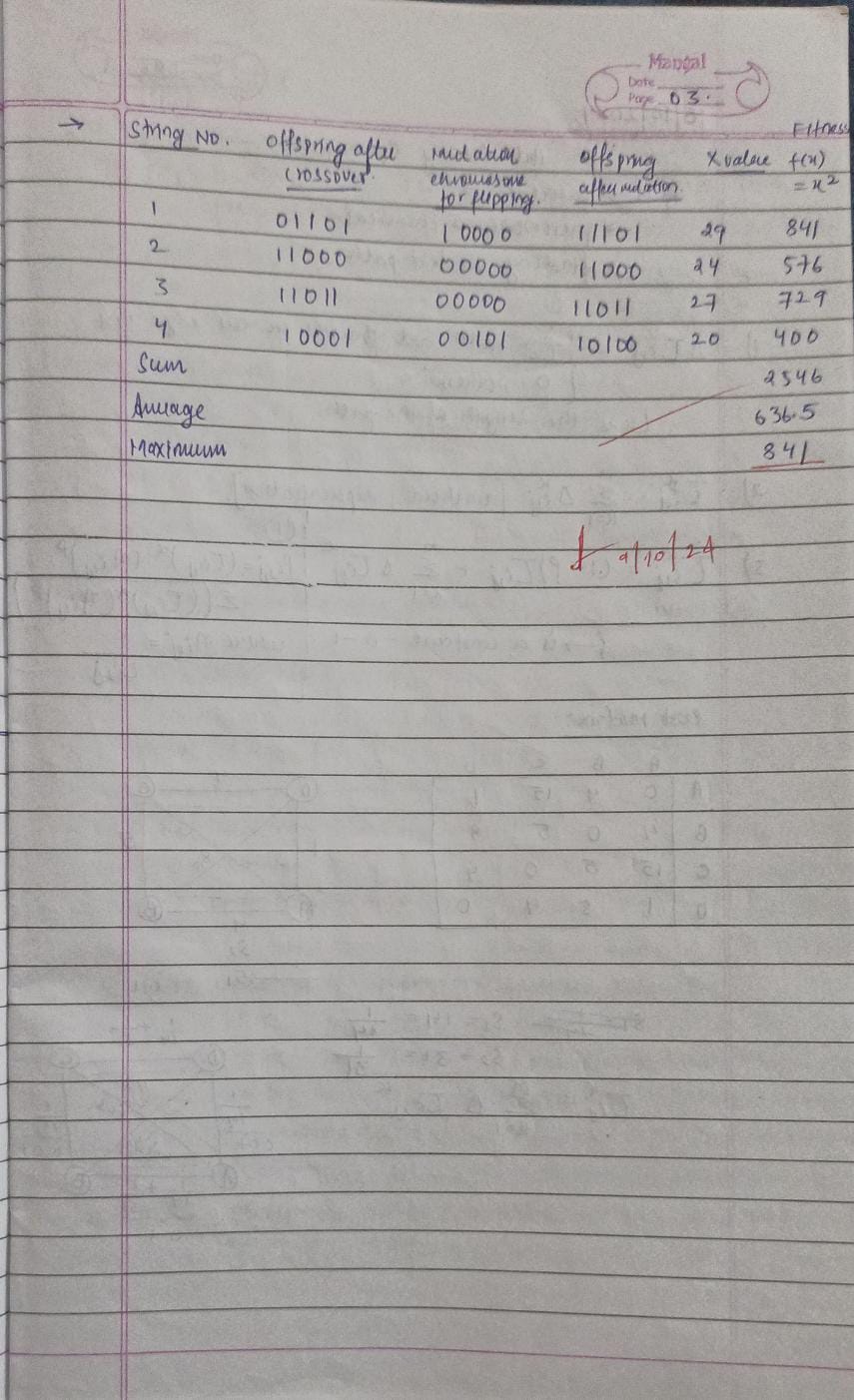
using Python to solve a basic optimization problem, such as finding the maximum value of a

mathematical function.

Algorithm







Code

import random

def fitness\_function(x, y):

return x \*\* 2

population\_size = 100

mutation\_rate = 0.1

crossover\_rate = 0.8

num\_generations = 50

variable\_bounds = [-10, 10]

def initialize\_population(population\_size, bounds):

population = []

for \_ in range(population\_size):

x = random.uniform(bounds[0], bounds[1])

y = random.uniform(bounds[0], bounds[1])

population.append([x, y])

return population

def evaluate\_population(population):

fitness\_scores = []

for individual in population:

fitness\_scores.append(fitness\_function(individual[0], individual[1]))

return fitness\_scores

def selection(population, fitness\_scores):

total\_fitness = sum(fitness\_scores)

selected\_population = []

for \_ in range(len(population)):

pick = random.uniform(0, total\_fitness)

current = 0

for individual, score in zip(population, fitness\_scores):

current += score

if current > pick:

selected\_population.append(individual)

break

return selected\_population

def crossover(parent1, parent2, crossover\_rate):

if random.random() < crossover\_rate:

crossover\_point = random.randint(1, len(parent1) - 1)

child1 = parent1[:crossover\_point] + parent2[crossover\_point:]

child2 = parent2[:crossover\_point] + parent1[crossover\_point:]

else:

child1, child2 = parent1, parent2

return child1, child2

def mutate(individual, mutation\_rate, bounds):

if random.random() < mutation\_rate:

mutation\_position = random.randint(0, len(individual) - 1)

mutation\_value = random.uniform(bounds[0], bounds[1])

individual[mutation\_position] = mutation\_value

return individual

def genetic\_algorithm():

# Display student information at the start

print("Student Name: Likhith M")

print("USN: 1BM22CS135")

print("-" \* 40)

population = initialize\_population(population\_size, variable\_bounds)

overall\_best\_solution = None

overall\_best\_fitness = float('-inf')

for generation in range(num\_generations):

fitness\_scores = evaluate\_population(population)

generation\_best\_fitness = max(fitness\_scores)

generation\_best\_solution = population[fitness\_scores.index(generation\_best\_fitness)]

if generation\_best\_fitness > overall\_best\_fitness:

overall\_best\_fitness = generation\_best\_fitness

overall\_best\_solution = generation\_best\_solution

# Displaying the output for generations that are multiples of 10

if (generation + 1) % 10 == 0:

print(f"Generation {generation + 1}:")

print(f" Best fitness = {generation\_best\_fitness}")

print(f" Best solution = {generation\_best\_solution}")

print("-" \* 40)

selected\_population = selection(population, fitness\_scores)

next\_population = []

for i in range(0, population\_size, 2):

parent1 = selected\_population[i]

parent2 = selected\_population[i + 1]

child1, child2 = crossover(parent1, parent2, crossover\_rate)

child1 = mutate(child1, mutation\_rate, variable\_bounds)

child2 = mutate(child2, mutation\_rate, variable\_bounds)

next\_population.append(child1)

next\_population.append(child2)

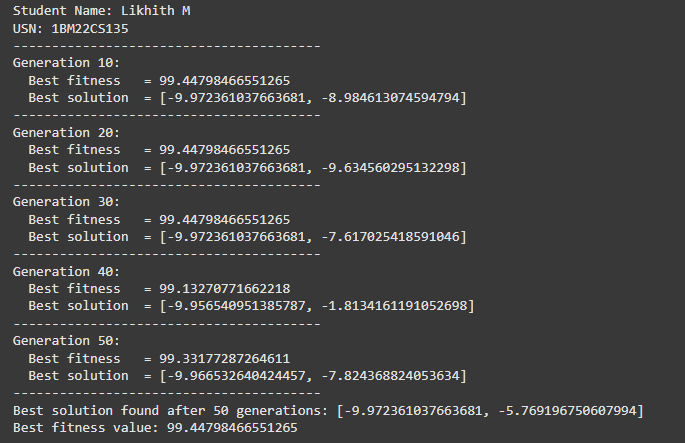
population = next\_population

print(f"Best solution found after {num\_generations} generations: {overall\_best\_solution}")

print(f"Best fitness value: {overall\_best\_fitness}")

genetic\_algorithm()

Output:

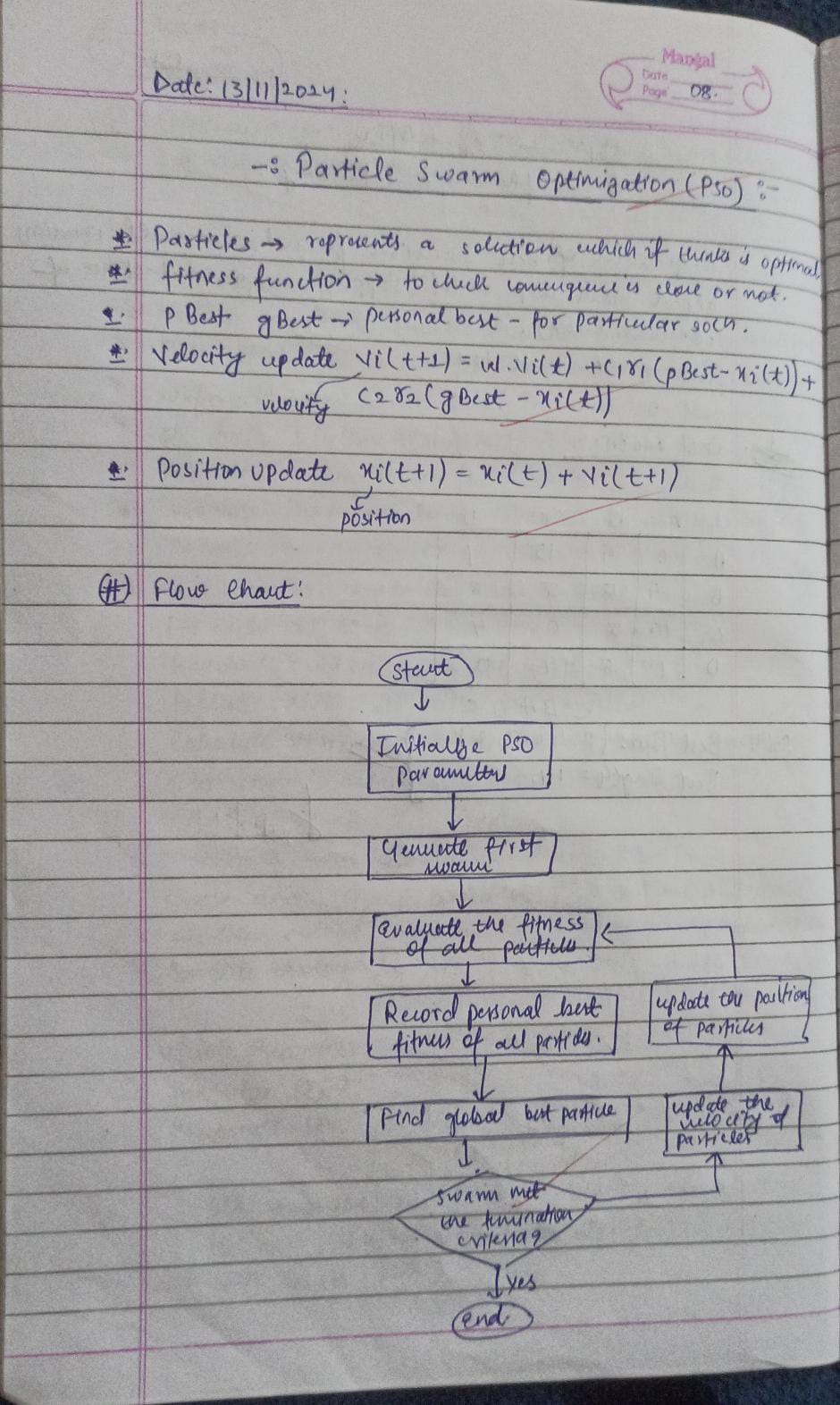


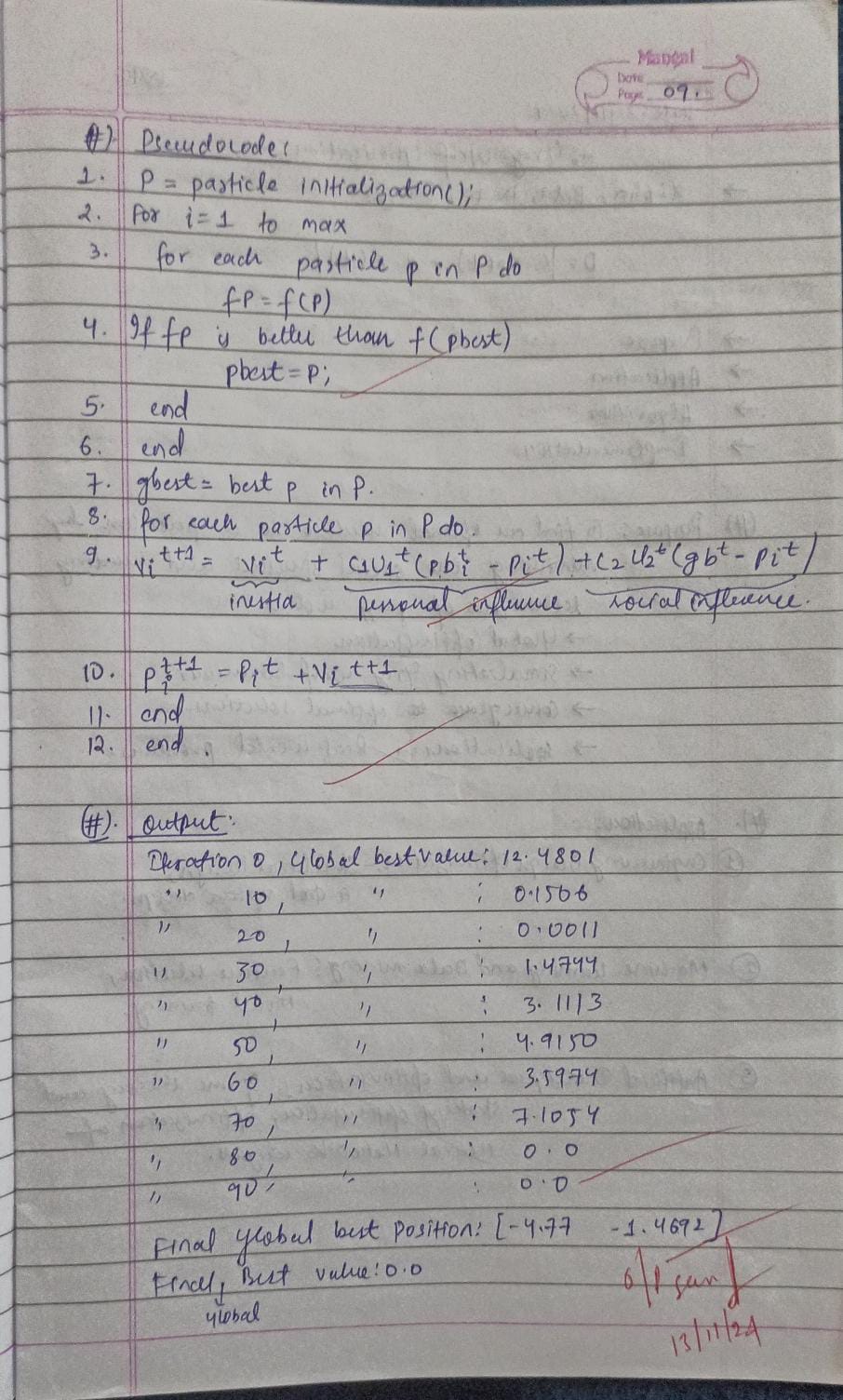
**Laboratory Program - 2**

**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





Code

import numpy as np

import matplotlib.pyplot as plt

# Objective function (Rastrigin function as an example)

def rastrigin(x):

    A = 10

    return A \* len(x) + sum(x\_i\*\*2 - A \* np.cos(2 \* np.pi \* x\_i) for x\_i in x)

# PSO Parameters

n\_particles = 30     # Number of particles

n\_dimensions = 2     # Number of dimensions (parameters to optimize)

n\_iterations = 100   # Number of iterations

w = 0.5              # Inertia weight

c1 = 1.5             # Cognitive coefficient (particle's own best position)

c2 = 1.5             # Social coefficient (global best position)

v\_max = 2.0          # Maximum velocity

# Initialize particles' positions and velocities

positions = np.random.uniform(-5.12, 5.12, (n\_particles, n\_dimensions))

velocities = np.random.uniform(-1, 1, (n\_particles, n\_dimensions))

# Initialize personal best positions and global best position

pbest\_positions = positions.copy()

pbest\_values = np.apply\_along\_axis(rastrigin, 1, pbest\_positions)

gbest\_position = pbest\_positions[np.argmin(pbest\_values)]

gbest\_value = np.min(pbest\_values)

# PSO Main Loop

for t in range(n\_iterations):

    # Evaluate fitness

    fitness\_values = np.apply\_along\_axis(rastrigin, 1, positions)

    # Update personal bests

    for i in range(n\_particles):

        if fitness\_values[i] < pbest\_values[i]:

            pbest\_positions[i] = positions[i]

            pbest\_values[i] = fitness\_values[i]

    # Update global best

    min\_fitness\_idx = np.argmin(pbest\_values)

    if pbest\_values[min\_fitness\_idx] < gbest\_value:

        gbest\_position = pbest\_positions[min\_fitness\_idx]

        gbest\_value = pbest\_values[min\_fitness\_idx]

    # Update velocity and position for each particle

    r1 = np.random.rand(n\_particles, n\_dimensions)

    r2 = np.random.rand(n\_particles, n\_dimensions)

    velocities = (w \* velocities +

                  c1 \* r1 \* (pbest\_positions - positions) +

                  c2 \* r2 \* (gbest\_position - positions))

    # Apply velocity limits (optional)

    velocities = np.clip(velocities, -v\_max, v\_max)

    # Update positions

    positions = positions + velocities

    # Print the current best solution only for multiples of 10

    if t % 10 == 0:

        print(f"Iteration {t}, Global Best Value: {gbest\_value:.5f}")

# Final output

print(f"\nFinal Global Best Position: {gbest\_position}")

print(f"Final Global Best Value: {gbest\_value:.5f}")

# Plotting the optimization process (visualization for 2D case)

x\_vals = np.linspace(-5.12, 5.12, 400)

y\_vals = np.linspace(-5.12, 5.12, 400)

X, Y = np.meshgrid(x\_vals, y\_vals)

Z = rastrigin([X, Y])

plt.contour(X, Y, Z, levels=np.linspace(0, 500, 50), cmap='jet')

plt.scatter(gbest\_position[0], gbest\_position[1], color='red', label='Global Best')

plt.title("PSO Optimization (Rastrigin Function)")

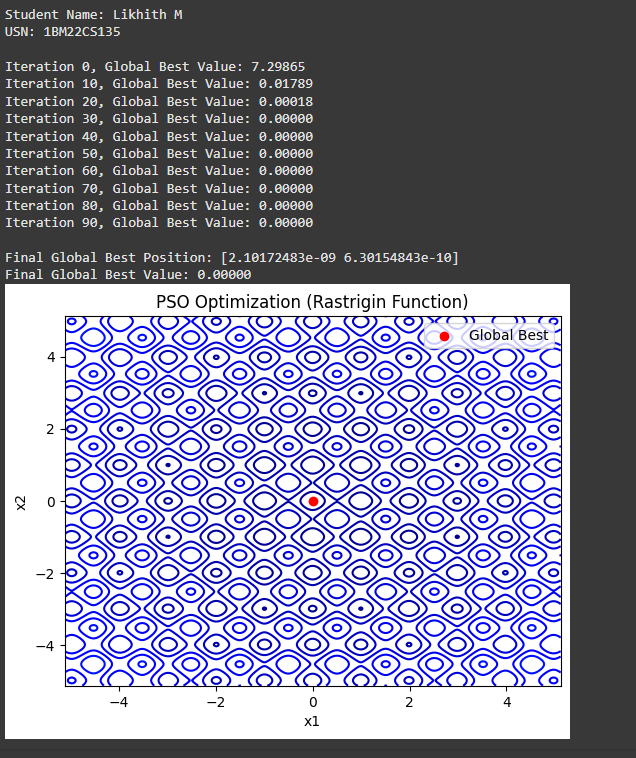
plt.xlabel('x1')

plt.ylabel('x2')

plt.legend()

plt.show()

Output

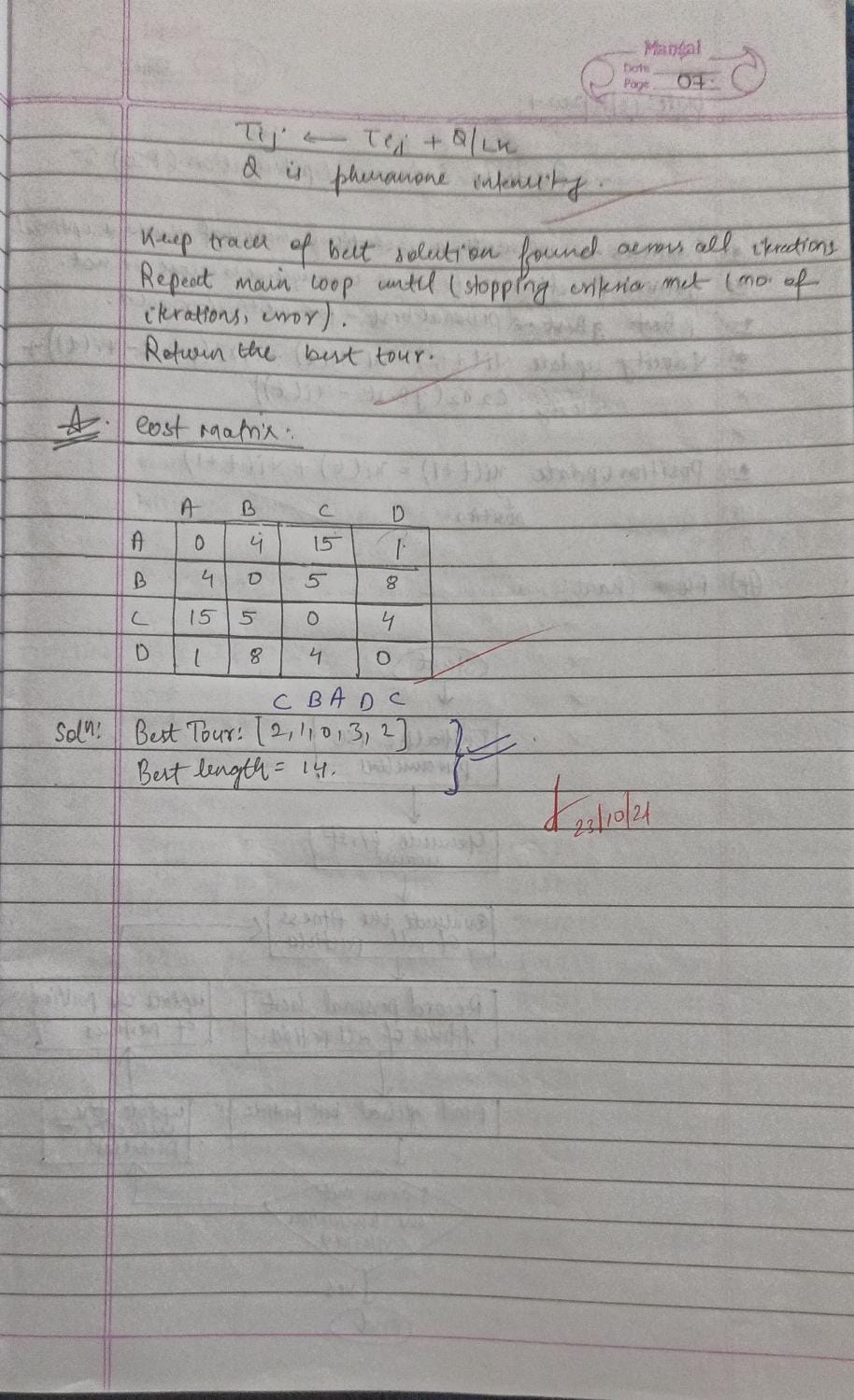
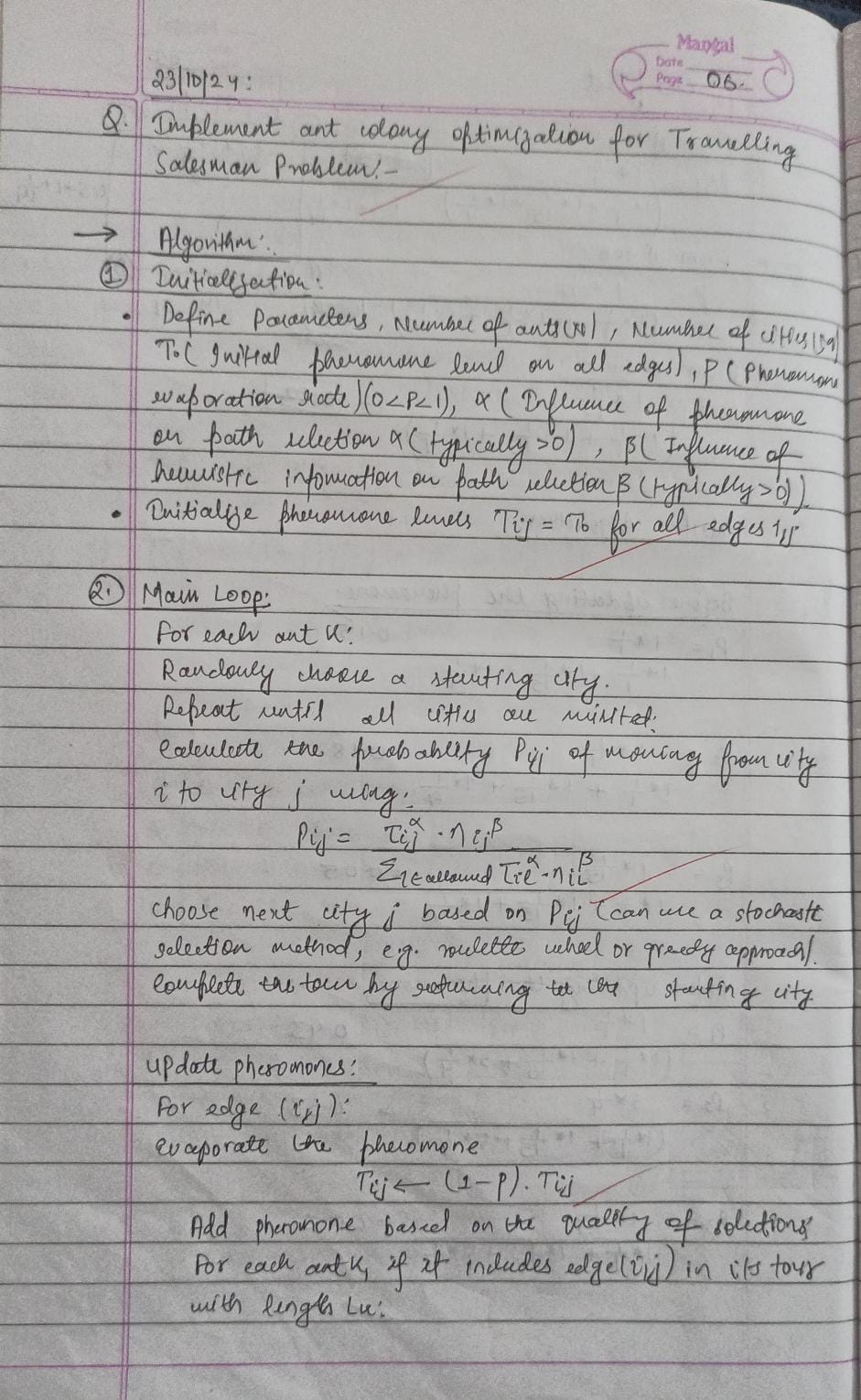
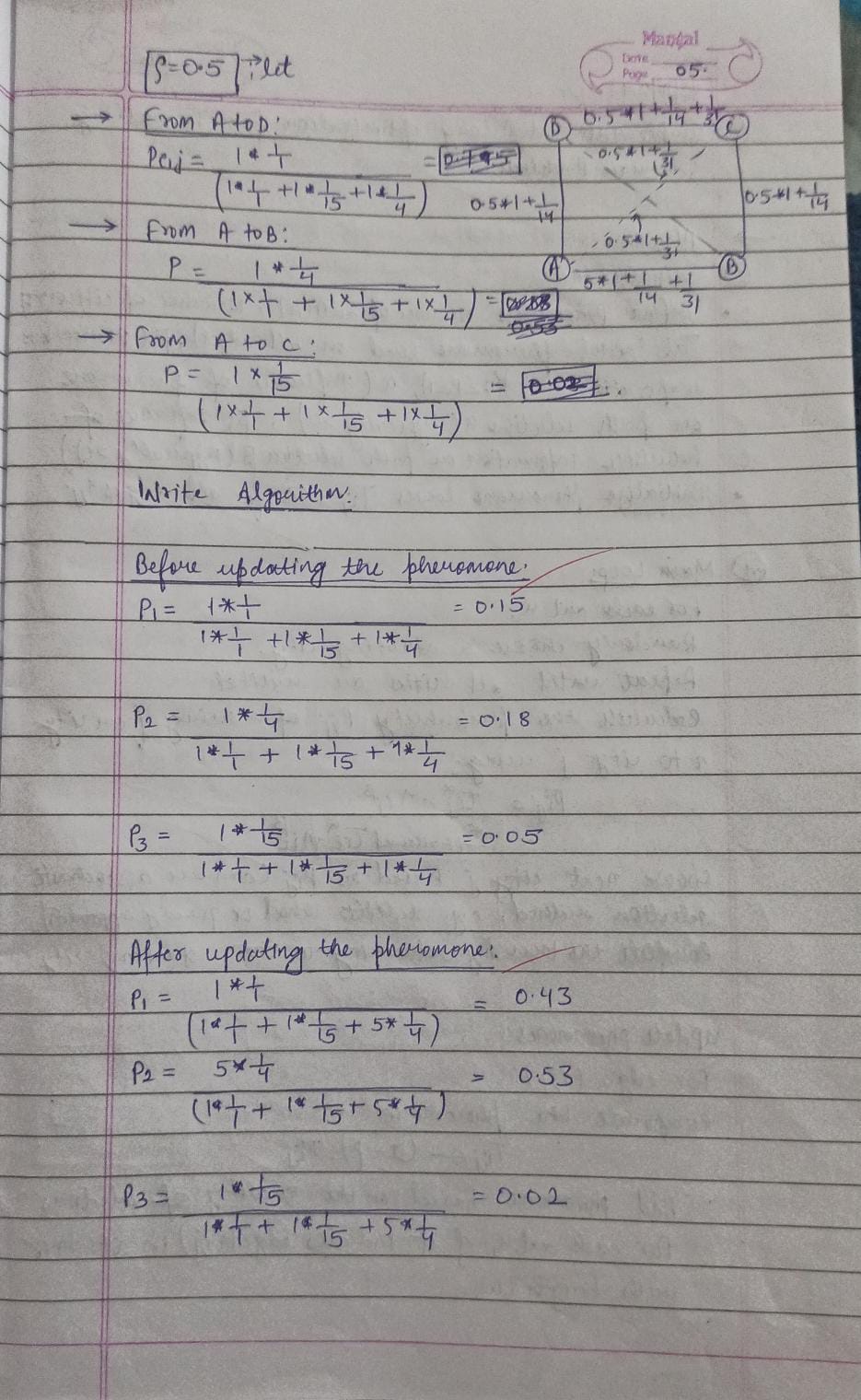
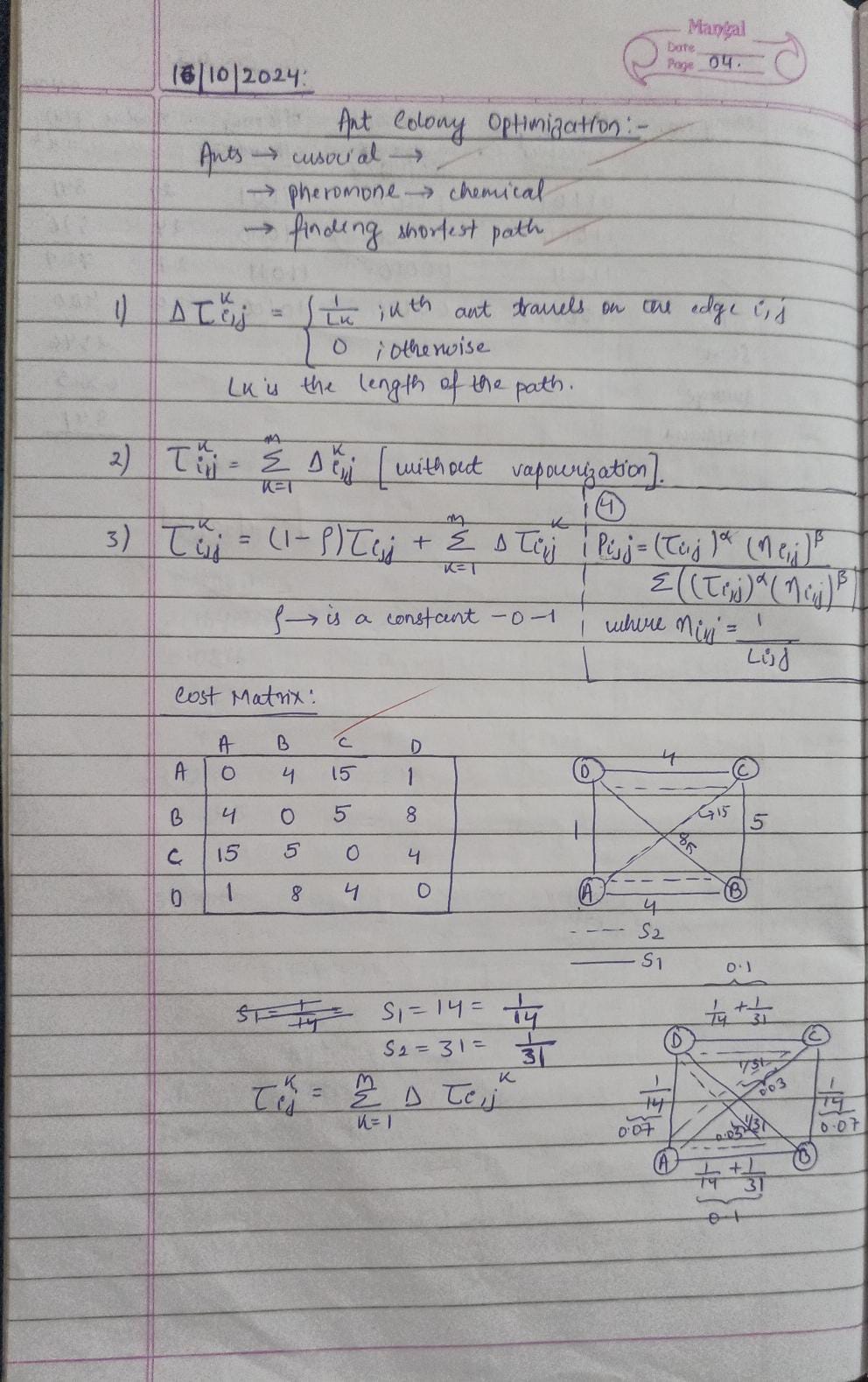


**Laboratory Program - 3**

**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



Code

import numpy as np

np.random.seed(0)

# Problem Parameters

num\_cities = 10

num\_ants = 20

alpha = 1.0

beta = 5.0

rho = 0.5

initial\_pheromone = 0.1

num\_iterations = 100

# Generate Random Cities

cities = np.random.rand(num\_cities, 2)

# Distance Matrix Calculation

def calculate\_distance\_matrix(cities):

num\_cities = len(cities)

distance\_matrix = np.zeros((num\_cities, num\_cities))

for i in range(num\_cities):

for j in range(i + 1, num\_cities):

distance = np.linalg.norm(cities[i] - cities[j])

distance\_matrix[i][j] = distance

distance\_matrix[j][i] = distance

return distance\_matrix

distance\_matrix = calculate\_distance\_matrix(cities)

# Initialize Pheromone Matrix

pheromone = np.ones((num\_cities, num\_cities)) \* initial\_pheromone

# City Selection

def select\_next\_city(probabilities):

return np.random.choice(len(probabilities), p=probabilities)

# Probability Calculation

def calculate\_probabilities(ant\_path, pheromone, distance\_matrix, alpha, beta):

current\_city = ant\_path[-1]

probabilities = np.zeros(len(distance\_matrix))

for city in range(len(distance\_matrix)):

if city not in ant\_path:

probabilities[city] = (pheromone[current\_city][city] \*\* alpha) \* ((1.0 / distance\_matrix[current\_city][city]) \*\* beta)

probabilities /= probabilities.sum()

return probabilities

# Solution Construction

def construct\_solution(pheromone, distance\_matrix, alpha, beta):

solution = []

for \_ in range(num\_ants):

ant\_path = [np.random.randint(num\_cities)]

while len(ant\_path) < num\_cities:

probabilities = calculate\_probabilities(ant\_path, pheromone, distance\_matrix, alpha, beta)

next\_city = select\_next\_city(probabilities)

ant\_path.append(next\_city)

solution.append(ant\_path + [ant\_path[0]])

return solution

# Pheromone Update

def update\_pheromones(pheromone, solutions, distance\_matrix, rho):

pheromone \*= (1 - rho)

for solution in solutions:

path\_length = sum(distance\_matrix[solution[i], solution[i+1]] for i in range(len(solution) - 1))

pheromone\_delta = 1.0 / path\_length

for i in range(len(solution) - 1):

pheromone[solution[i]][solution[i + 1]] += pheromone\_delta

pheromone[solution[i + 1]][solution[i]] += pheromone\_delta

return pheromone

# Optimization Process

best\_solution = None

best\_path\_length = float('inf')

for iteration in range(num\_iterations):

solutions = construct\_solution(pheromone, distance\_matrix, alpha, beta)

pheromone = update\_pheromones(pheromone, solutions, distance\_matrix, rho)

for solution in solutions:

path\_length = sum(distance\_matrix[solution[i], solution[i + 1]] for i in range(len(solution) - 1))

if path\_length < best\_path\_length:

best\_path\_length = path\_length

best\_solution = solution

# Display output only for multiples of 10

if (iteration + 1) % 10 == 0:

print(f"Iteration {iteration + 1}: Path length = {best\_path\_length:.5f}")

# Check for convergence

if iteration > 0 and best\_path\_length == previous\_best\_path\_length:

print(f"Convergence reached at iteration {iteration + 1}. Best solution found.")

break

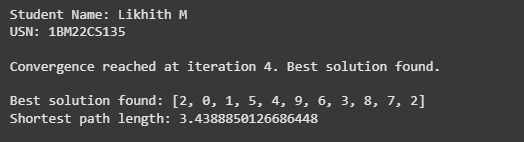
previous\_best\_path\_length = best\_path\_length

# Final Output

print("\nBest solution found:", best\_solution)

print("Shortest path length:", best\_path\_length)

Output

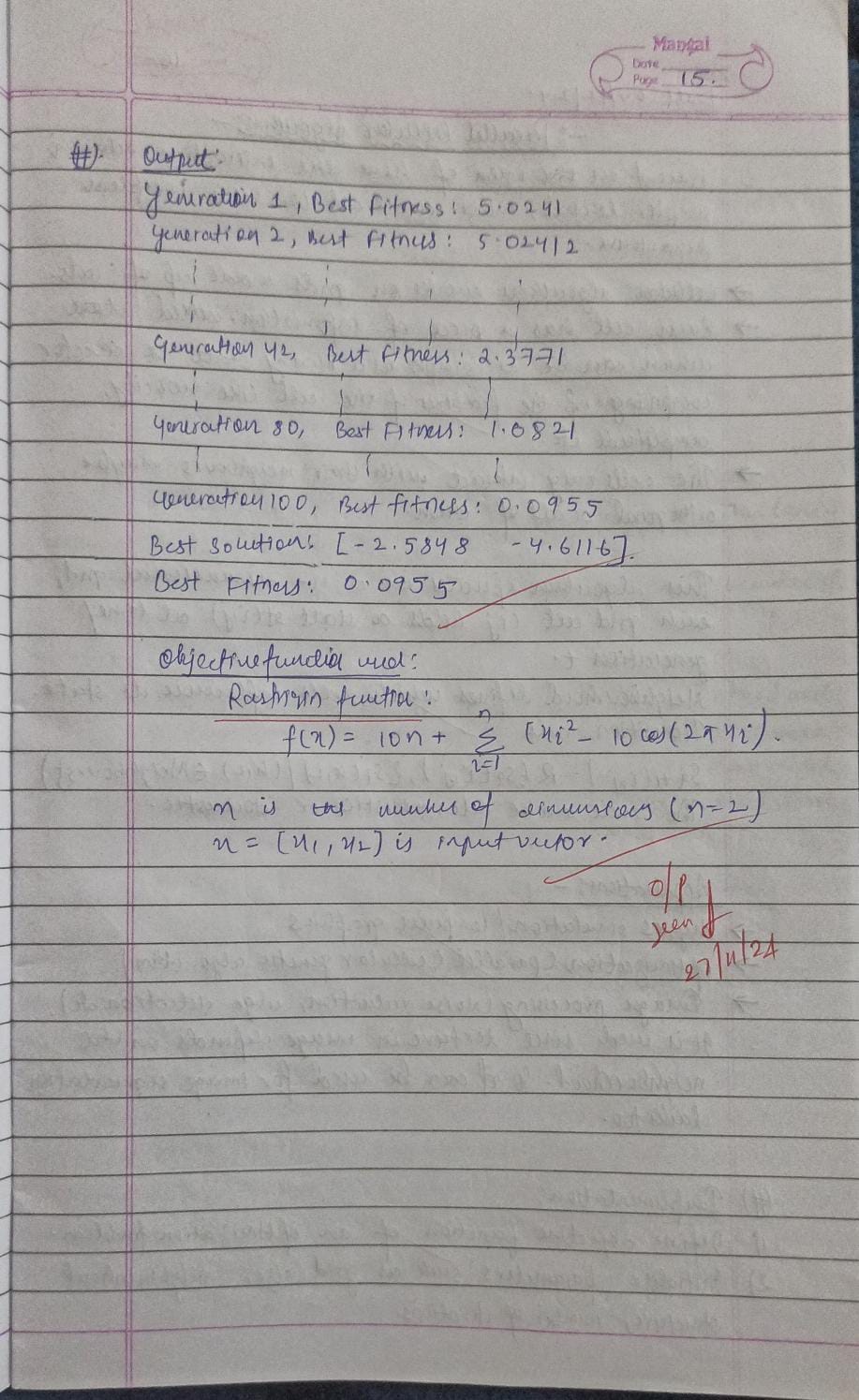
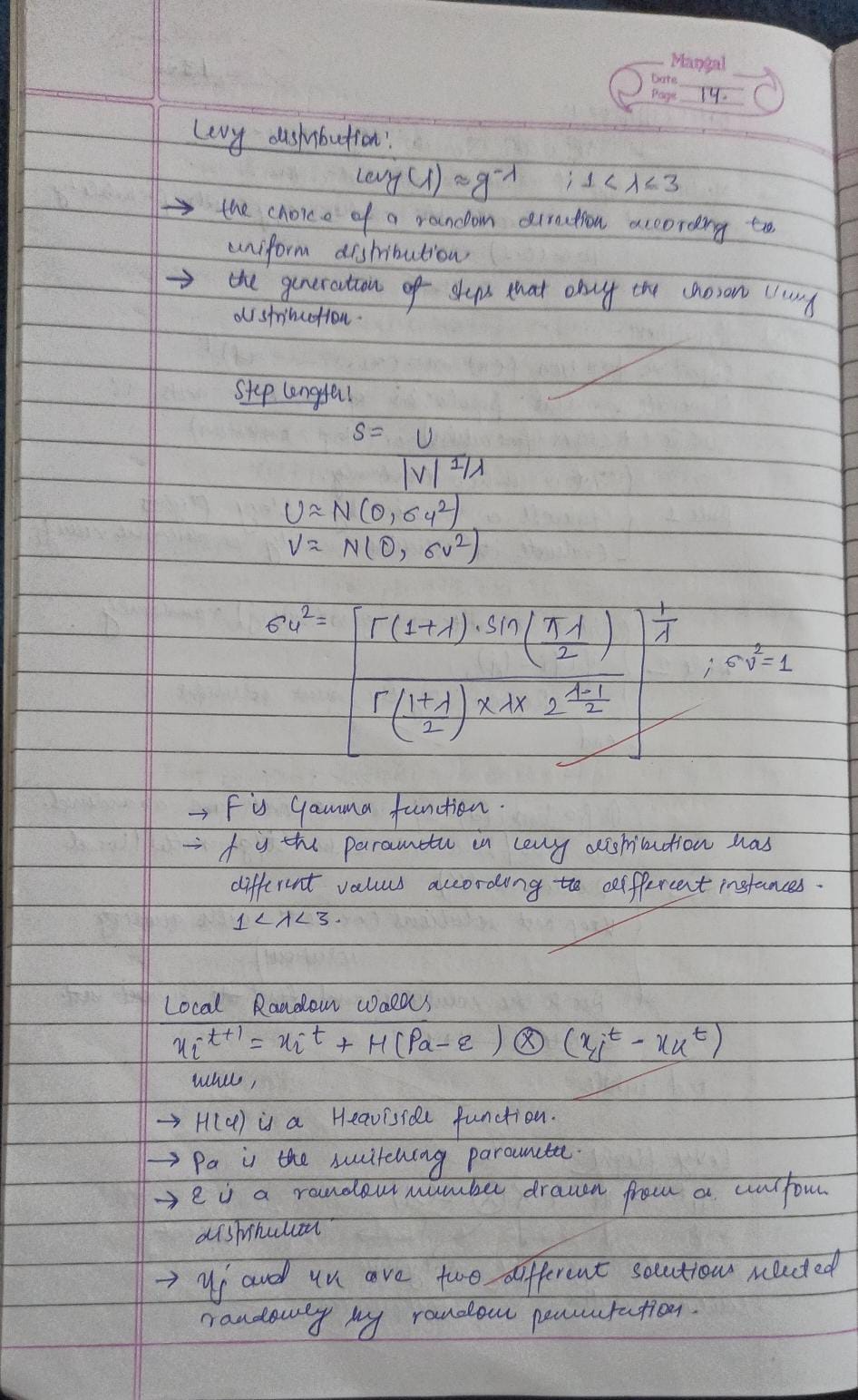
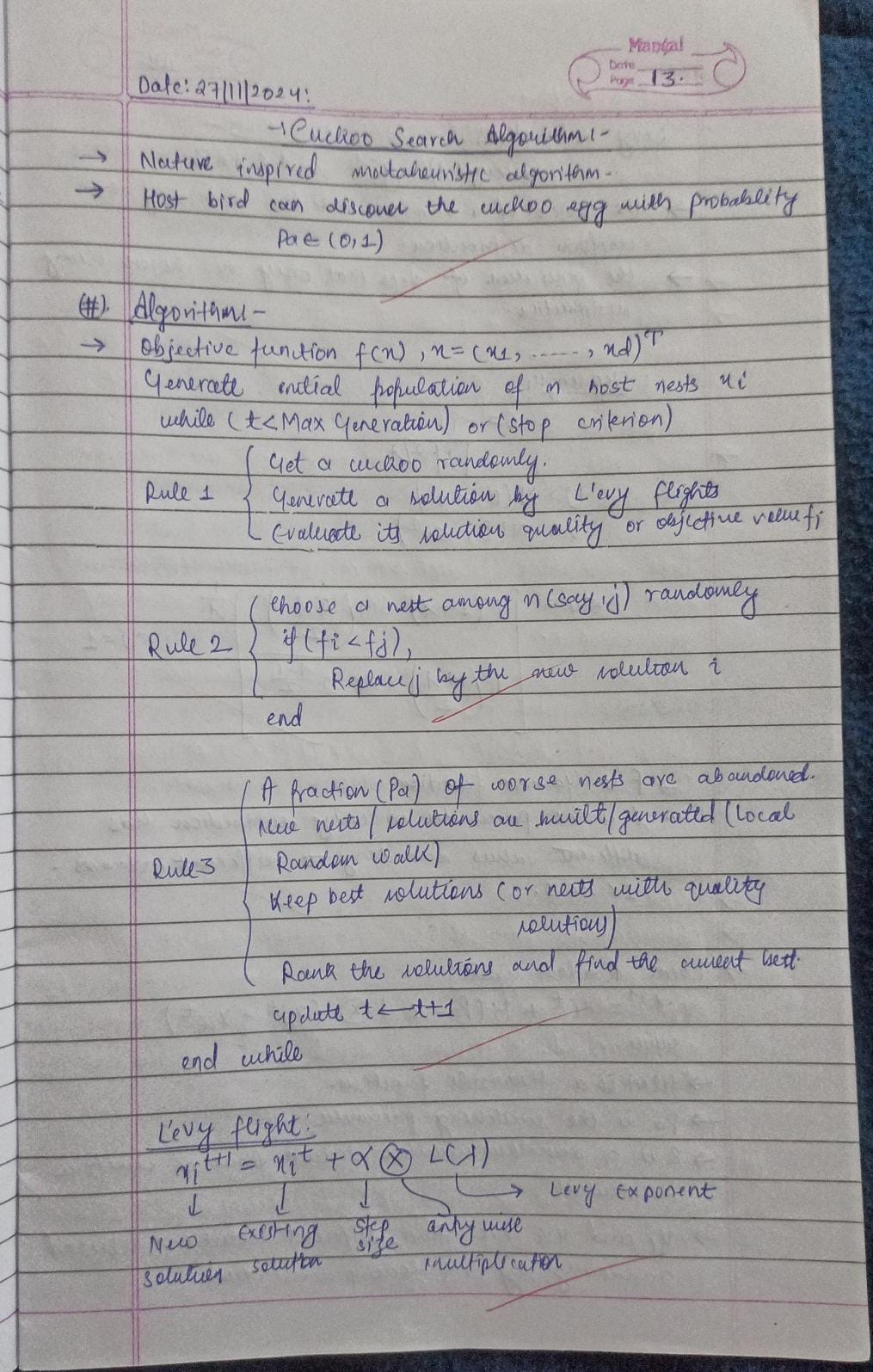


**Laboratory Program - 4**

**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



Code

import random

import math

# Objective Function

def objective\_function(x):

return sum(xi \*\* 2 for xi in x)

# Levy Flight Step

def levy\_flight(Lambda, dim):

sigma = (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) for \_ in range(dim)]

v = [random.gauss(0, 1) for \_ in range(dim)]

step = [ui / abs(vi) \*\* (1 / Lambda) for ui, vi in zip(u, v)]

return step

# Generate New Solution

def generate\_new\_solution(current\_solution, alpha, Lambda, lower\_bound, upper\_bound):

step = levy\_flight(Lambda, len(current\_solution))

new\_solution = [

max(min(current\_solution[i] + alpha \* step[i], upper\_bound), lower\_bound)

for i in range(len(current\_solution))

]

return new\_solution

# Initialize Nests

def initialize\_nests(n\_nests, dim, lower\_bound, upper\_bound):

return [[random.uniform(lower\_bound, upper\_bound) for \_ in range(dim)] for \_ in range(n\_nests)]

# Cuckoo Search Algorithm

def cuckoo\_search(objective, n\_nests, max\_iter, alpha, pa, lower\_bound=-10, upper\_bound=10, Lambda=1.5):

nests = initialize\_nests(n\_nests, 2, lower\_bound, upper\_bound)

fitness = [objective(nest) for nest in nests]

best\_nest = min(nests, key=objective)

best\_fitness = objective(best\_nest)

for iteration in range(max\_iter):

random\_index = random.randint(0, n\_nests - 1)

cuckoo\_solution = generate\_new\_solution(nests[random\_index], alpha, Lambda, lower\_bound, upper\_bound)

cuckoo\_fitness = objective(cuckoo\_solution)

random\_nest\_index = random.randint(0, n\_nests - 1)

if cuckoo\_fitness < fitness[random\_nest\_index]:

nests[random\_nest\_index] = cuckoo\_solution

fitness[random\_nest\_index] = cuckoo\_fitness

num\_to\_abandon = int(pa \* n\_nests)

worst\_indices = sorted(range(n\_nests), key=lambda i: fitness[i], reverse=True)[:num\_to\_abandon]

for idx in worst\_indices:

nests[idx] = [random.uniform(lower\_bound, upper\_bound) for \_ in range(2)]

fitness[idx] = objective(nests[idx])

current\_best\_index = min(range(n\_nests), key=lambda i: fitness[i])

if fitness[current\_best\_index] < best\_fitness:

best\_nest = nests[current\_best\_index]

best\_fitness = fitness[current\_best\_index]

# Output every 100 iterations

if iteration % 100 == 0 or iteration == max\_iter - 1:

print(f"Iteration {iteration}, Best Fitness: {best\_fitness:.5f}, Best Solution: {best\_nest}")

return best\_nest, best\_fitness

# Main Function

if \_\_name\_\_ == "\_\_main\_\_":

n\_nests = 25 # Number of nests

max\_iter = 1000 # Maximum iterations

alpha = 0.1 # Step size

pa = 0.25 # Probability of abandoning worse nests

best\_solution, best\_value = cuckoo\_search(

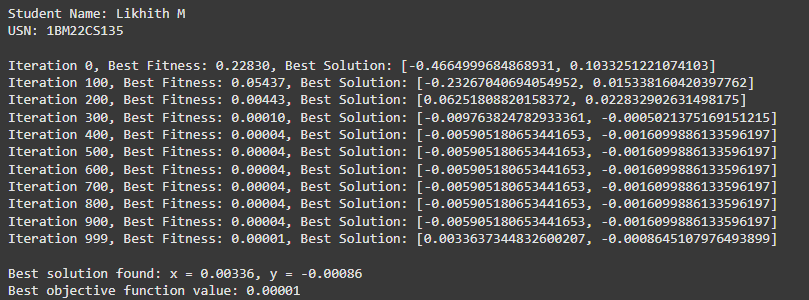
objective\_function, n\_nests=n\_nests, max\_iter=max\_iter, alpha=alpha, pa=pa

)

print(f"\nBest solution found: x = {best\_solution[0]:.5f}, y = {best\_solution[1]:.5f}")

print(f"Best objective function value: {best\_value:.5f}")

Output

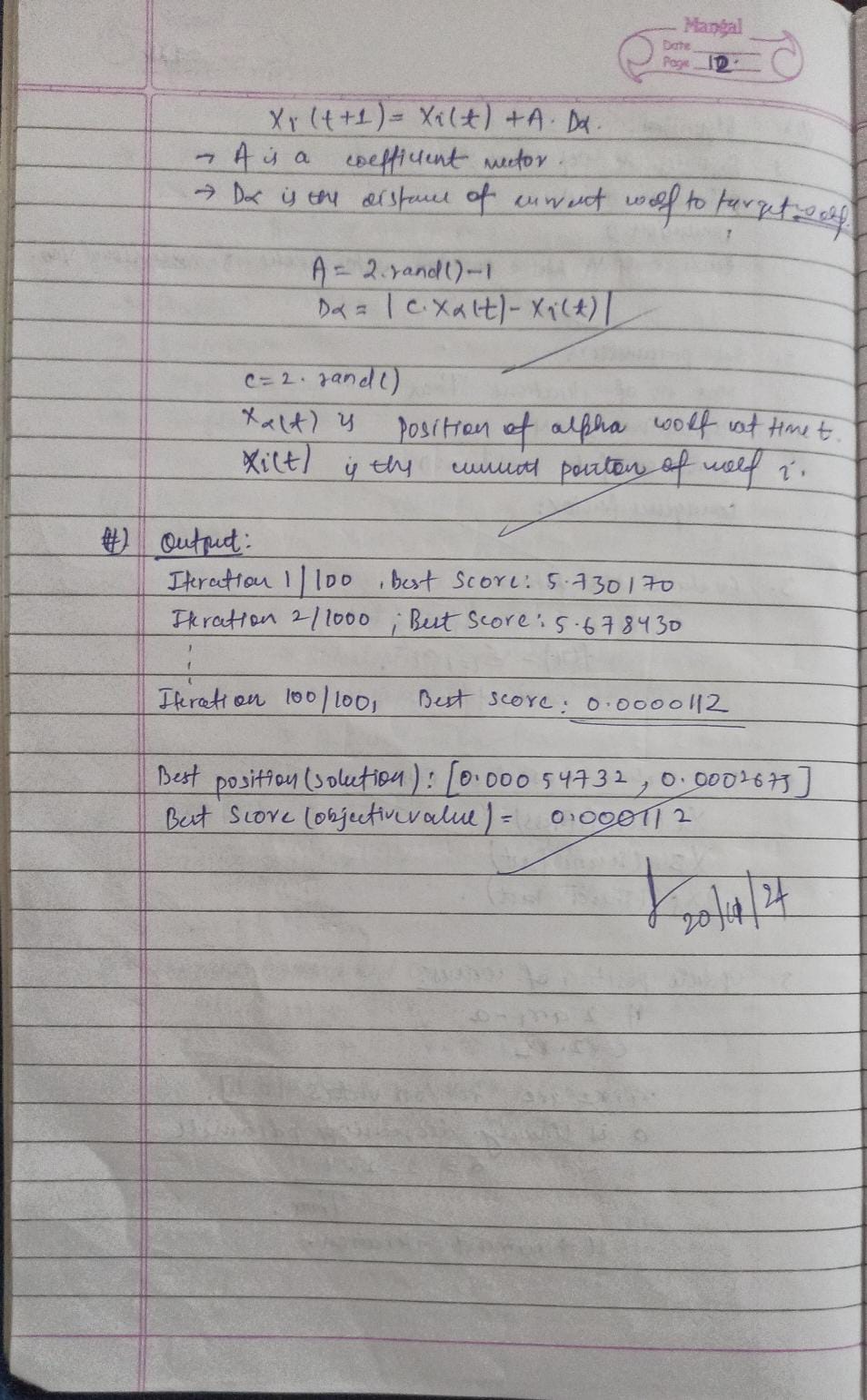
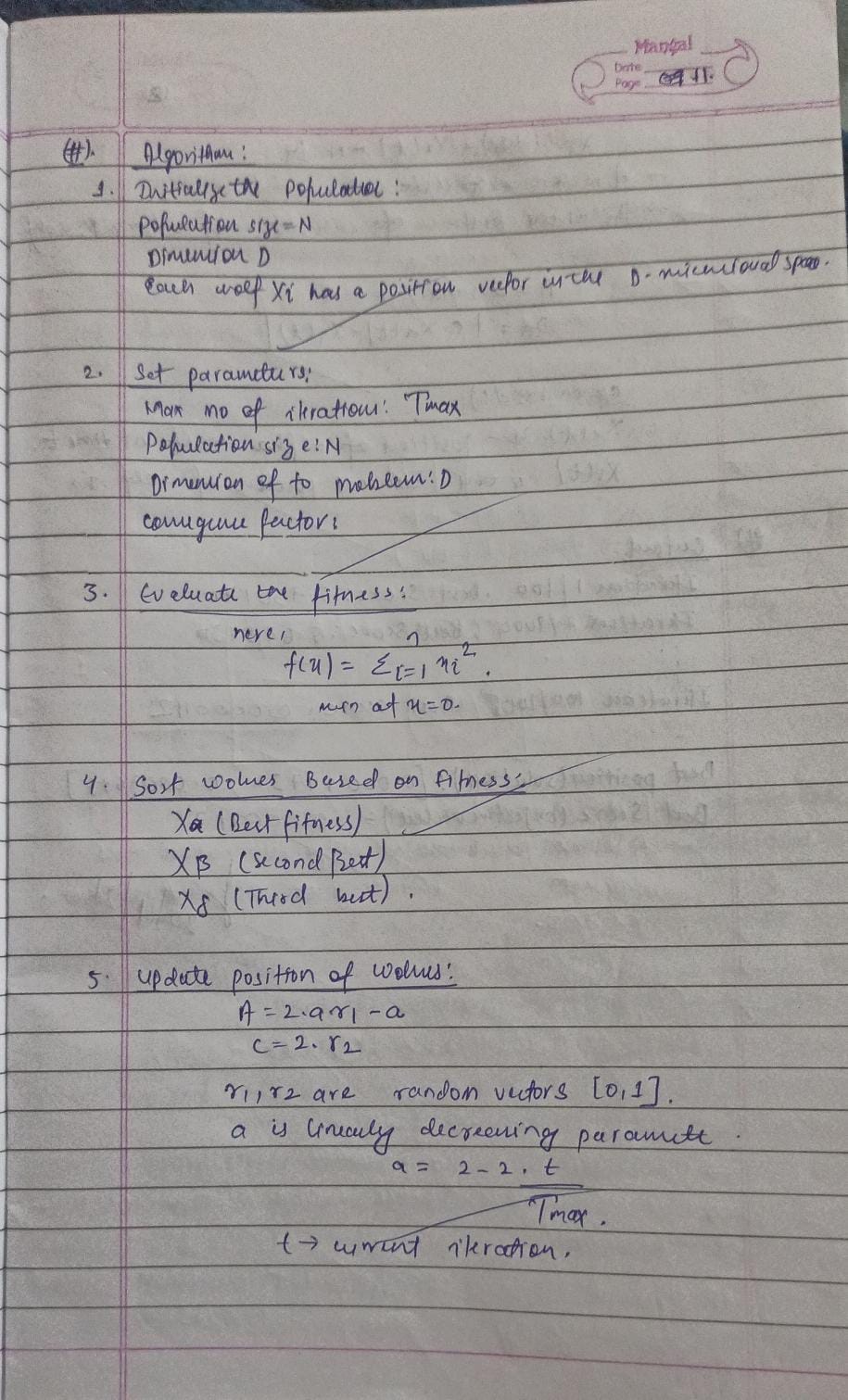
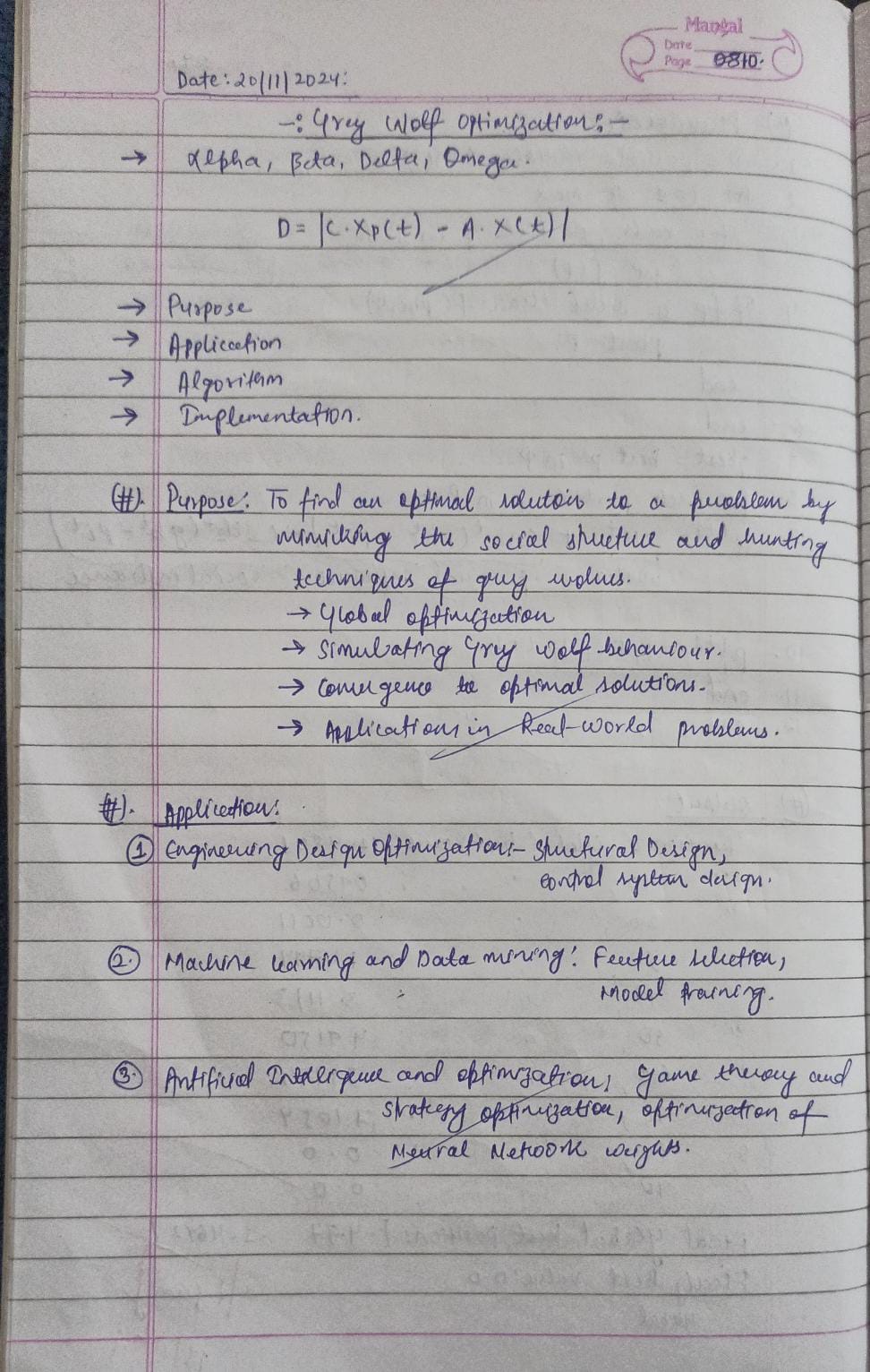


**Laboratory Program - 5**

**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



Code

import random

# Grey Wolf Optimizer Function

def grey\_wolf\_optimizer(objective\_function, lower\_bound, upper\_bound, dim, num\_wolves, max\_iter):

alpha\_pos = [0] \* dim

beta\_pos = [0] \* dim

delta\_pos = [0] \* dim

alpha\_score = float('inf')

beta\_score = float('inf')

delta\_score = float('inf')

wolves = [[random.uniform(lower\_bound, upper\_bound) for \_ in range(dim)] for \_ in range(num\_wolves)]

for iteration in range(max\_iter):

for i in range(num\_wolves):

fitness = objective\_function(wolves[i])

if fitness < alpha\_score:

delta\_score, delta\_pos = beta\_score, beta\_pos[:]

beta\_score, beta\_pos = alpha\_score, alpha\_pos[:]

alpha\_score, alpha\_pos = fitness, wolves[i][:]

elif fitness < beta\_score:

delta\_score, delta\_pos = beta\_score, beta\_pos[:]

beta\_score, beta\_pos = fitness, wolves[i][:]

elif fitness < delta\_score:

delta\_score, delta\_pos = fitness, wolves[i][:]

a = 2 - iteration \* (2 / max\_iter)

for i in range(num\_wolves):

for j in range(dim):

r1 = random.random()

r2 = random.random()

A1 = a \* (2 \* r1 - 1)

C1 = 2 \* r2

D\_alpha = abs(C1 \* alpha\_pos[j] - wolves[i][j])

X1 = alpha\_pos[j] - A1 \* D\_alpha

r1 = random.random()

r2 = random.random()

A2 = a \* (2 \* r1 - 1)

C2 = 2 \* r2

D\_beta = abs(C2 \* beta\_pos[j] - wolves[i][j])

X2 = beta\_pos[j] - A2 \* D\_beta

r1 = random.random()

r2 = random.random()

A3 = a \* (2 \* r1 - 1)

C3 = 2 \* r2

D\_delta = abs(C3 \* delta\_pos[j] - wolves[i][j])

X3 = delta\_pos[j] - A3 \* D\_delta

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

if wolves[i][j] < lower\_bound:

wolves[i][j] = lower\_bound

elif wolves[i][j] > upper\_bound:

wolves[i][j] = upper\_bound

# Output every 10 iterations

if iteration % 10 == 0 or iteration == max\_iter - 1:

print(f"Iteration {iteration}: Best Score = {alpha\_score:.5f}, Best Position = {alpha\_pos}")

return alpha\_pos, alpha\_score

# Sphere Function (Objective Function)

def sphere\_function(position):

return sum(x \*\* 2 for x in position)

# Problem Parameters

lower\_bound = -10

upper\_bound = 10

dim = 3

num\_wolves = 25

max\_iter = 50

# Run Grey Wolf Optimizer

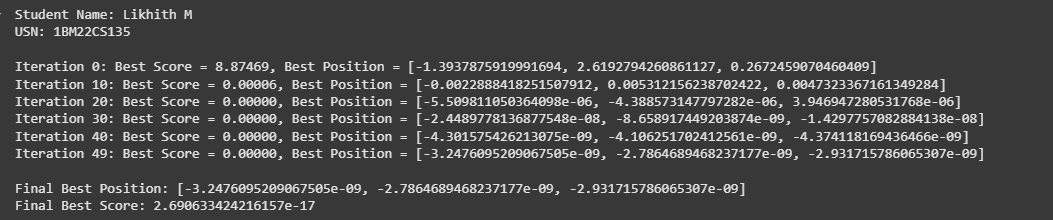
best\_position, best\_score = grey\_wolf\_optimizer(sphere\_function, lower\_bound, upper\_bound, dim, num\_wolves, max\_iter)

# Final Output

print("\nFinal Best Position:", best\_position)

print("Final Best Score:", best\_score)

Output

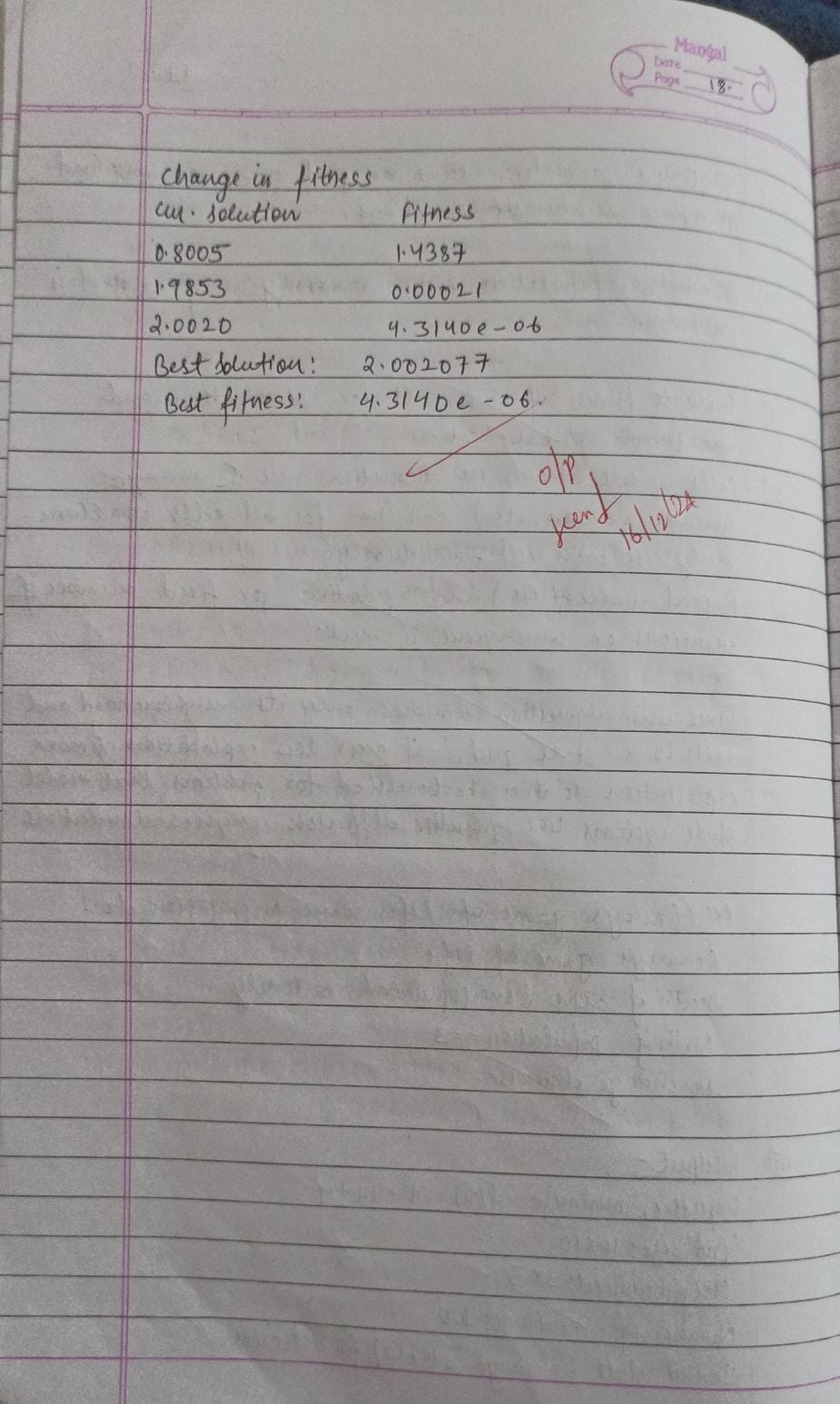
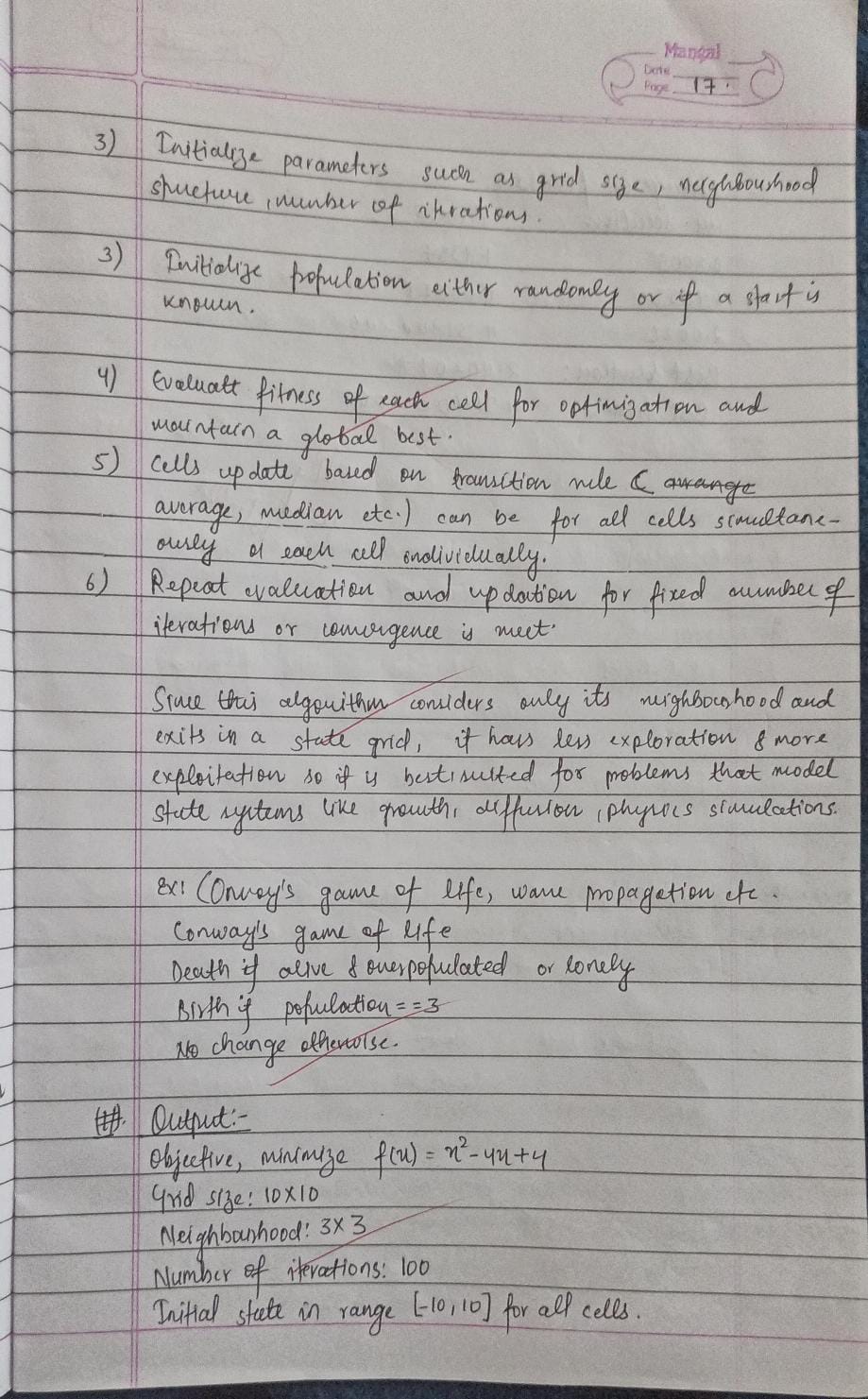
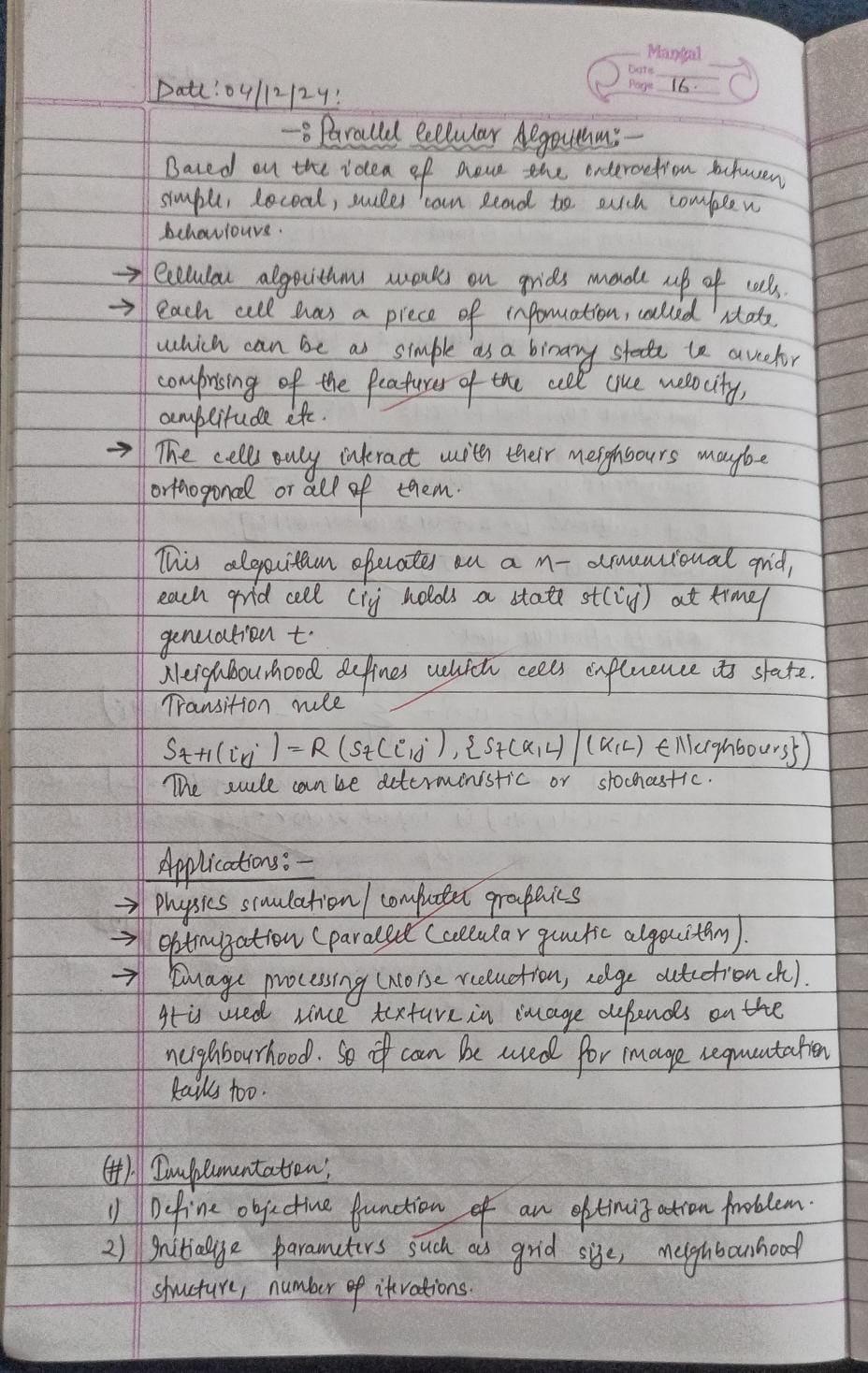


**Laboratory Program - 6**

**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



Code

import random

# Objective Function

def objective\_function(x):

return -(x \*\* 2) + 4 \* x

# Initialize Parameters

def initialize\_parameters():

grid\_size = 10 # Grid size

num\_iterations = 50 # Number of iterations

lower\_bound, upper\_bound = -10, 10 # Bounds for the grid values

return grid\_size, num\_iterations, lower\_bound, upper\_bound

# Initialize Population Grid

def initialize\_population(grid\_size, lower\_bound, upper\_bound):

grid = [[random.uniform(lower\_bound, upper\_bound) for \_ in range(grid\_size)] for \_ in range(grid\_size)]

return grid

# Evaluate Fitness Grid

def evaluate\_fitness(grid):

fitness\_grid = [[objective\_function(cell) for cell in row] for row in grid]

return fitness\_grid

# Update Grid States Based on Neighbor Averages

def update\_states(grid, fitness\_grid):

grid\_size = len(grid)

updated\_grid = [[0] \* grid\_size for \_ in range(grid\_size)]

for i in range(grid\_size):

for j in range(grid\_size):

neighbors = []

for di in [-1, 0, 1]:

for dj in [-1, 0, 1]:

if di == 0 and dj == 0:

continue

ni, nj = i + di, j + dj

if 0 <= ni < grid\_size and 0 <= nj < grid\_size:

neighbors.append(grid[ni][nj])

if neighbors:

updated\_grid[i][j] = sum(neighbors) / len(neighbors)

else:

updated\_grid[i][j] = grid[i][j]

return updated\_grid

# Print Grid State

def print\_grid(grid, label="Grid"):

print(f"{label}")

for row in grid:

print([f"{value: .4f}" for value in row])

print()

# Parallel Cellular Algorithm

def parallel\_cellular\_algorithm():

grid\_size, num\_iterations, lower\_bound, upper\_bound = initialize\_parameters()

grid = initialize\_population(grid\_size, lower\_bound, upper\_bound)

print\_grid(grid, label="Initial Grid")

best\_solution = None

best\_fitness = float('-inf')

for iteration in range(num\_iterations):

fitness\_grid = evaluate\_fitness(grid)

for i in range(grid\_size):

for j in range(grid\_size):

if fitness\_grid[i][j] > best\_fitness:

best\_fitness = fitness\_grid[i][j]

best\_solution = grid[i][j]

# Output progress at multiples of 10 iterations

if iteration % 10 == 0 or iteration == num\_iterations - 1:

print(f"Iteration {iteration}: Best Solution = {best\_solution:.4f}, Best Fitness = {best\_fitness:.4f}")

grid = update\_states(grid, fitness\_grid)

return best\_solution, best\_fitness

# Main Function

if \_\_name\_\_ == "\_\_main\_\_":

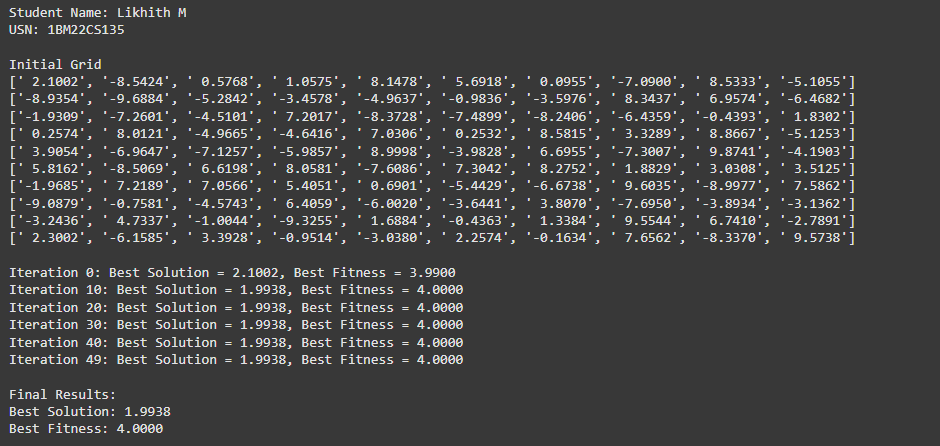
best\_solution, best\_fitness = parallel\_cellular\_algorithm()

print("\nFinal Results:")

print(f"Best Solution: {best\_solution:.4f}")

print(f"Best Fitness: {best\_fitness:.4f}")

Output

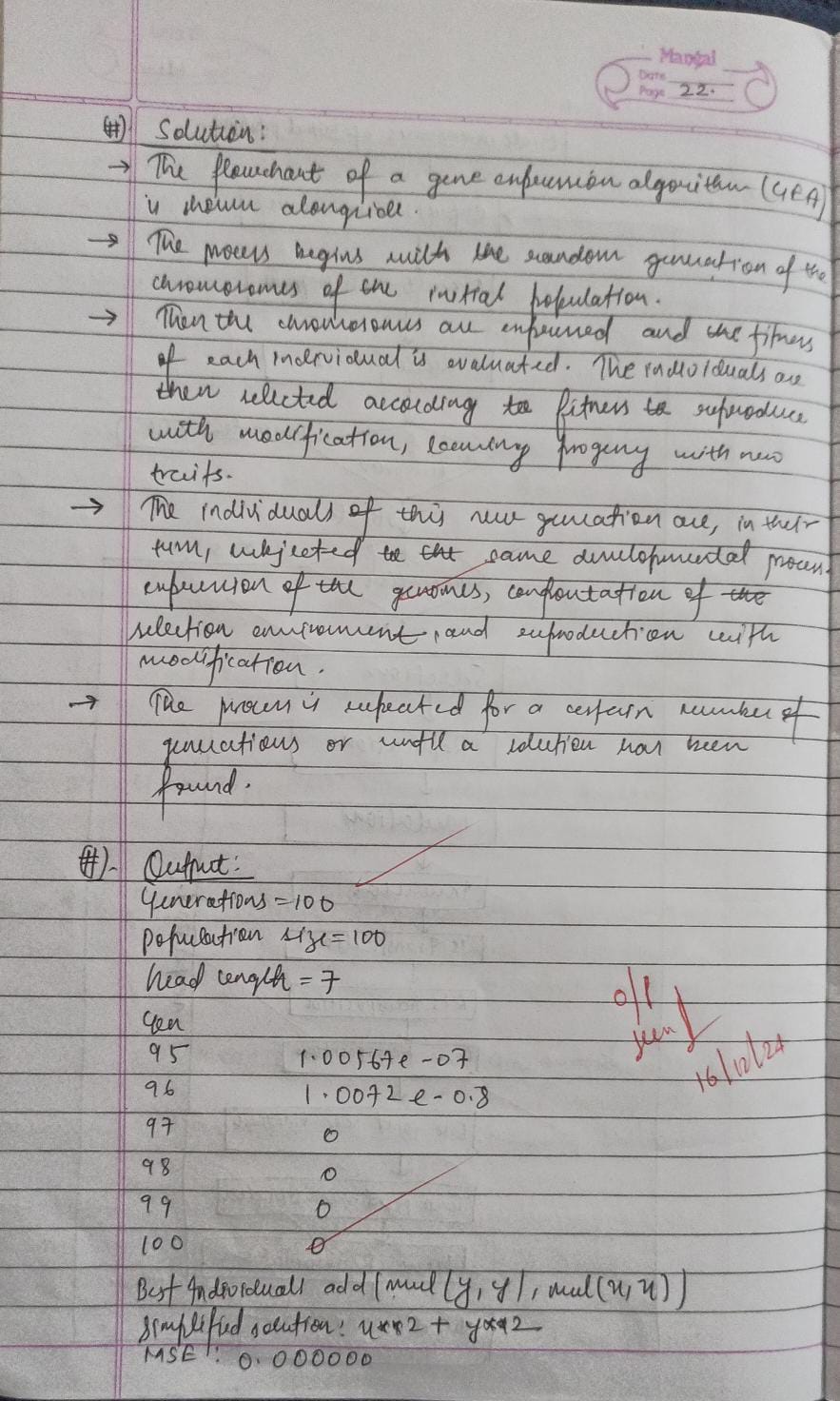
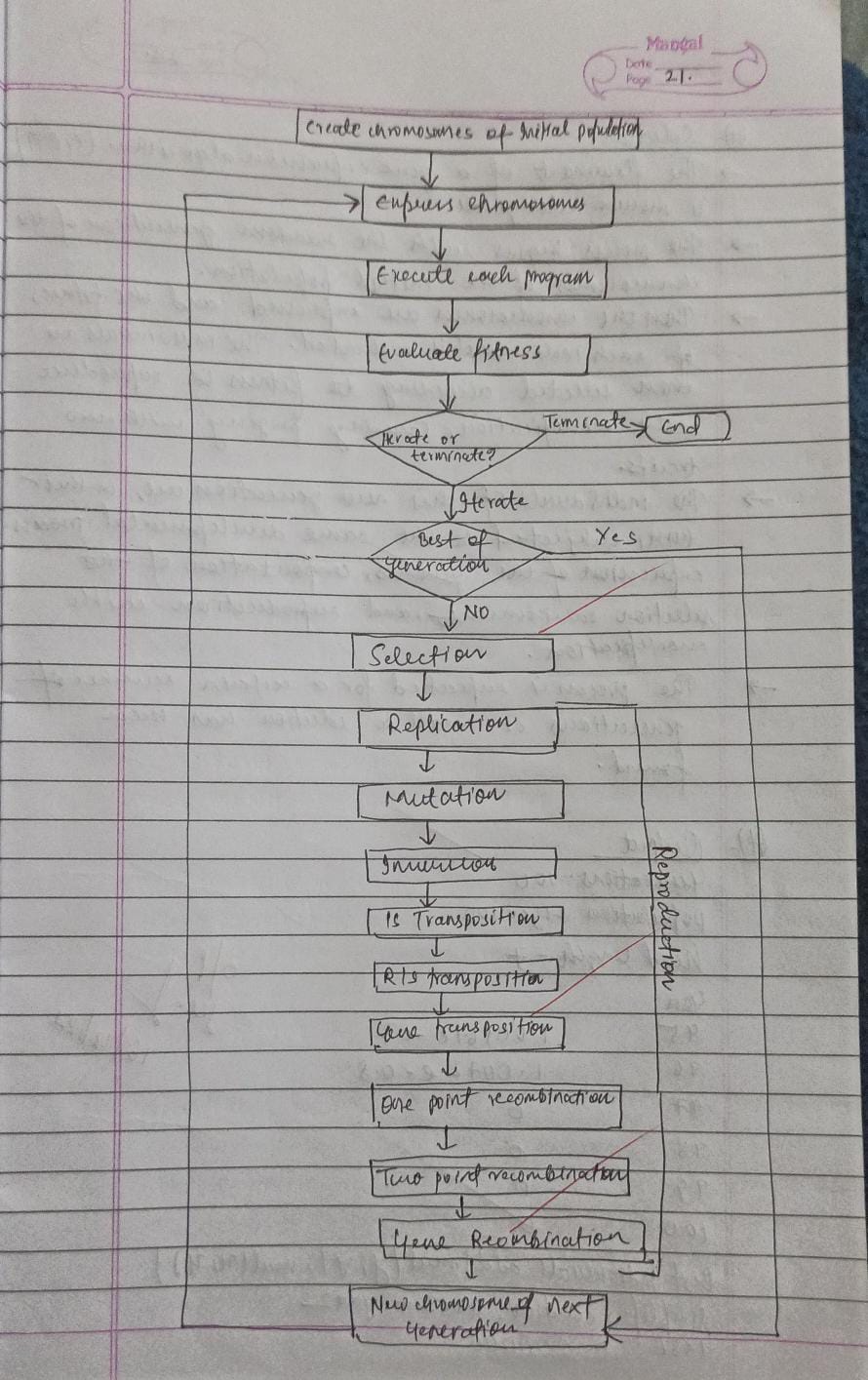
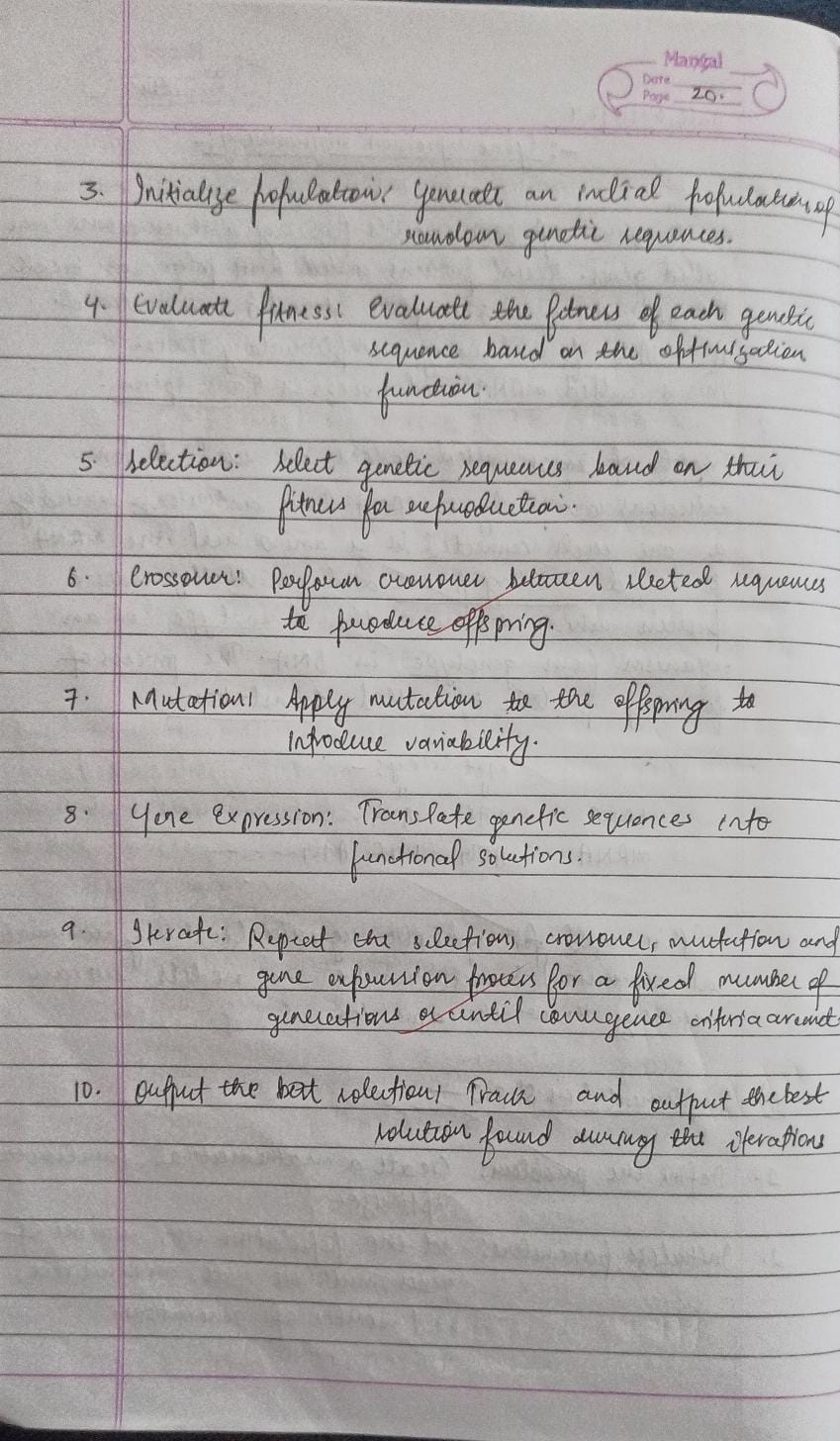
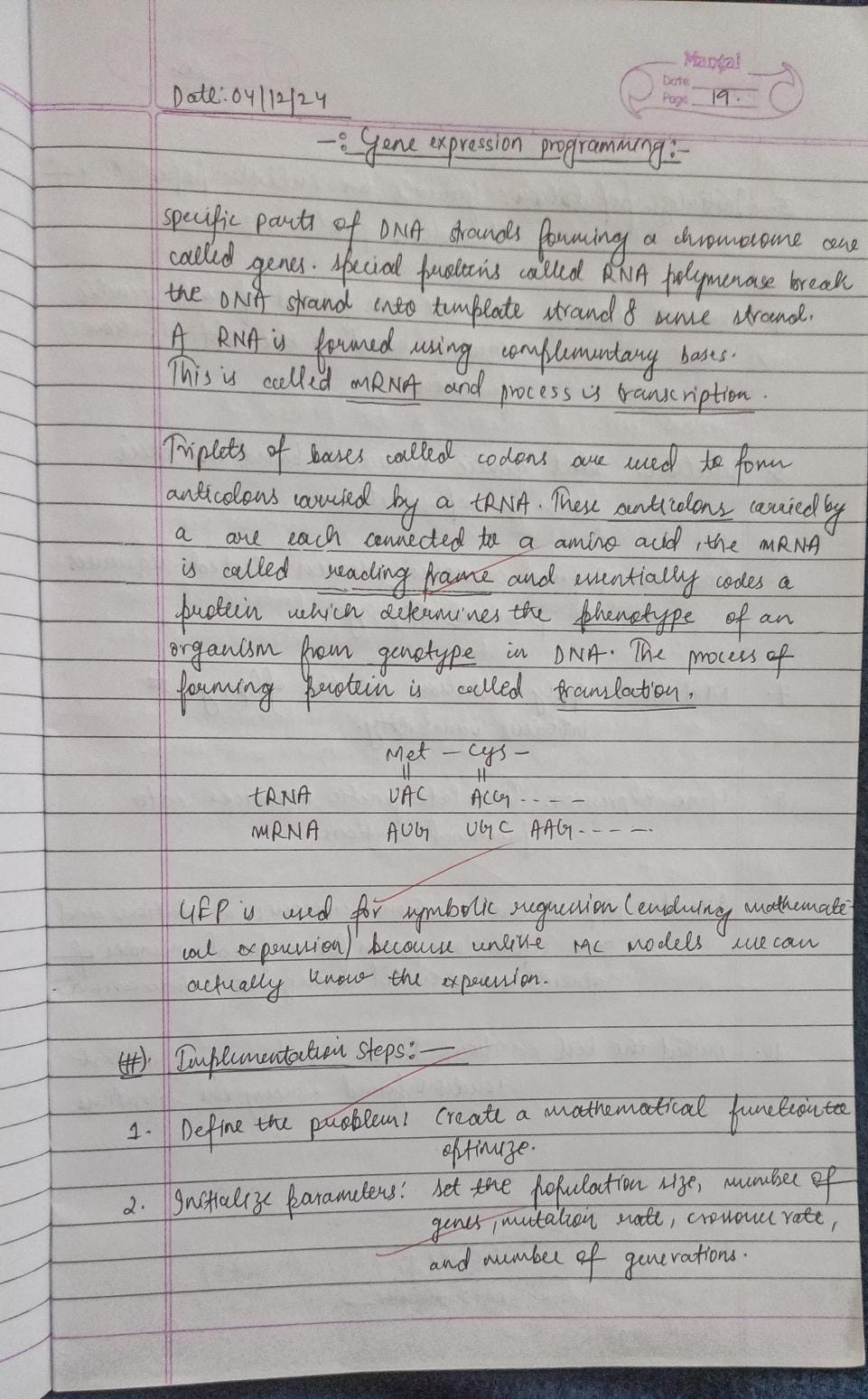


**Laboratory Program - 7**

**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 optimizatsion 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



Code

import operator

import numpy as np

import geppy as gep

from deap import creator, base, tools

# Step 1: Define the target function

def target\_function(x, y):

return x\*\*2 + y\*\*2

# Step 2: Define the dataset

x\_data = np.linspace(-10, 10, 50)

y\_data = np.linspace(-10, 10, 50)

X, Y = np.meshgrid(x\_data, y\_data)

Z = target\_function(X, Y) # Target outputs

# Flatten the data for evaluation

inputs = np.array([X.ravel(), Y.ravel()]).T

outputs = Z.ravel()

# Step 3: Define the GEP primitive set

pset = gep.PrimitiveSet('main', input\_names=['x', 'y'])

pset.add\_function(operator.add, 2)

pset.add\_function(operator.mul, 2)

pset.add\_constant\_terminal(3)

# Step 4: Define the fitness and individual

if not hasattr(creator, "FitnessMax"):

creator.create("FitnessMax", base.Fitness, weights=(1.0,))

if not hasattr(creator, "Individual"):

creator.create('Individual', gep.Chromosome, fitness=creator.FitnessMax)

# Define head length and number of genes

h = 10 # Set head length to a suitable value

n\_genes = 2 # Adjusted to ensure compatibility with the linker

# Step 5: Define the toolbox

toolbox = gep.Toolbox()

# Register chromosome, population, and compile function

toolbox.register('gene\_gen', gep.Gene, pset=pset, head\_length=h)

toolbox.register('individual', creator.Individual, gene\_gen=toolbox.gene\_gen, n\_genes=n\_genes, linker=operator.add)

toolbox.register('population', tools.initRepeat, list, toolbox.individual)

toolbox.register('compile', gep.compile\_, pset=pset)

# Define the fitness evaluation function

def evaluate(individual):

func = toolbox.compile(individual)

predictions = np.array([func(\*input\_pair) for input\_pair in inputs])

fitness = -np.mean((outputs - predictions)\*\*2) # Negative MSE

return fitness,

toolbox.register('evaluate', evaluate)

# Register selection, mutation, and crossover operators

toolbox.register('select', tools.selRoulette)

toolbox.register('mut\_uniform', gep.mutate\_uniform, pset=pset, ind\_pb=0.1)

toolbox.register('mut\_invert', gep.invert, pb=0.1)

toolbox.register('mut\_is\_ts', gep.is\_transpose, pb=0.1)

toolbox.register('mut\_ris\_ts', gep.ris\_transpose, pb=0.1)

toolbox.register('mut\_gene\_ts', gep.gene\_transpose, pb=0.1)

toolbox.register('cx\_1p', gep.crossover\_one\_point, pb=0.4)

toolbox.register('cx\_2p', gep.crossover\_two\_point, pb=0.2)

toolbox.register('cx\_gene', gep.crossover\_gene, pb=0.1)

# Explicitly set probabilities for the operators in Toolbox.pbs

toolbox.pbs['mut\_uniform'] = 0.1 # Set the probability for mut\_uniform

# Step 6: Define statistics and Hall of Fame

stats = tools.Statistics(key=lambda ind: ind.fitness.values[0])

stats.register("avg", np.mean)

stats.register("std", np.std)

stats.register("min", np.min)

stats.register("max", np.max)

hof = tools.HallOfFame(3)

# Step 7: Set population size and generations

n\_pop = 100

n\_gen = 5

pop = toolbox.population(n=n\_pop)

print("Starting Genetic Programming Evolution...\n")

# Start evolution

pop, log = gep.gep\_simple(pop, toolbox, n\_generations=n\_gen, n\_elites=1,

stats=stats, hall\_of\_fame=hof, verbose=True)

# Step 8: Output the best individual

best\_individual = hof[0]

simplified\_solution = gep.simplify(best\_individual)

print("\nBest Individual (Chromosome):")

print(best\_individual)

print("\nSimplified Solution:")

print(simplified\_solution)

# Evaluate the error of the solution

best\_func = toolbox.compile(best\_individual)

predictions = np.array([best\_func(\*input\_pair) for input\_pair in inputs])

mse = np.mean((outputs - predictions)\*\*2)

print(f"\nMean Squared Error of the Best Solution: {mse:.6f}")

# Export the expression tree

rename\_labels = {'add': '+', 'mul': '\*'}

gep.export\_expression\_tree(best\_individual, rename\_labels, file='tree.png')

print("\nExpression tree exported to 'tree.png'.")

Output

