MA23434 OPTIMIZATION TECHNIQUES FOR AI LABORATORY MANUAL

R-2023
IV SEMESTER

B.Tech. – Artificial Intelligence & Data Science and Artificial Intelligence & Machine Learning

NAME :--
REG. NO :------------------------------------
BRANCH :-------------------------------------

DEPARTMENT OF HUMANITIES AND SCIENCES - MATHEMATICS
RAJALAKSHMI ENGINEERING COLLEGE (AUTONOMOUS)
THANDALAM -602105

RAJALAKSHMI ENGINEERING COLLEGE (AUTONOMOUS)
RAJALAKSHMI NAGAR, THANDALAM - 602 105

Vision

- To be an institution of excellence in Engineering, Technology and Management Education & Research.

- To provide competent and ethical professional with a concern for society.

Mission

- To impart quality technical education imbibed with proficiency and humane values.

- To provide the right ambience and opportunities for the students to develop into creative, talented, and globally competent professionals.

- To promote research and development in technology and management for the benefit of the society.

RAJALAKSHMI ENGINEERING COLLEGE (AUTONOMOUS)
RAJALAKSHMI NAGAR, THANDALAM - 602 105

DEPARTMENT OF HUMANITIES AND SCIENCES - MATHEMATICS

Vision

To become a Department of Excellence in Mathematical education, Physical Sciences, Research and Development and Setting the standards of Excellence in Communication skills and preparing the learners to meet the challenging trends in the professional career.

Mission

- To impart high quality education in Applied Mathematics and Physical Sciences at various levels of programmes in engineering and technology.

- To build confidence in engineering students in using mathematical and scientific tools for designing industrial products and systems in a better, faster and effective way.

- To improve the Communication Skills of engineering students and make them suitable for employment.

SYLLABUS

Course Code	Course Title	Category	L	T	P	C
MA23434	**OPTIMIZATION TECHNIQUES FOR AI**	**BS**	3	0	2	4
Common to IV Sem. B.Tech. – Artificial Intelligence & Data Science and Artificial Intelligence & Machine Learning						

Objectives:
• To enumerate the fundamental knowledge of Linear Programming problems.
• To develop formulation skills in transportation and assignment models and finding solutions.
• To formulate and solve the pure integer, mixed integer or 0-1 integer linear programming models.
• To analyse the problems of unconstrained nonlinear programming and to know the necessary and sufficient conditions for the solution of unconstrained problems.
• To find the best ways to crash project schedule, shortening total project duration and the ways to save money by adjusting activity durations and optimizing resource requirements.

UNIT-I	**INTRODUCTION TO OPTIMIZATION**	**9**
Convex sets, Convex function-Linear Optimization: formulation, solution by graphical and simplex methods - Primal-Penalty- Two Phase –Principles of Duality.		
UNIT-II	**TRANSPORTATION AND ASSIGNMENT MODELS**	**9**
Transportation Models (Minimizing and Maximizing Problems) – Initial Basic feasible solution by Vogel's approximation methods- Check for optimality: Solution by MODI algorithm - Case of Degeneracy- Assignment Models -Solution by Hungarian method- Introduction to Bandit algorithm.		
UNIT-III	**INTEGER PROGRAMMING**	**9**
Cutting plane algorithm –Branch and bound methods - Multistage (Dynamic) programming.		
UNIT-IV	**NON – LINEAR OPTIMIZATION**	**9**
Unconstrained external problems -Newton–Raphson method – Equality constraints– Gradient Descent Method - Jacobian methods–Lagrangian method–Kuhn–Tucker conditions–Simple problems.		
UNIT-V	**PROJECT SCHEDULING**	**9**
Network diagram representation– Critical path method–Time charts and resource leveling–PERT.		
Total Contact Hours: 45		

Course Outcomes: On completion of the course, the students will be able to
• Solve Linear Programming problems using different methods.
• Formulate and solve transportation and assignment models arising in the field of engineering and technology.
• Set up and solve the pure integer, mixed integer or 0-1 integer linear programming problems in engineering and technology.
• Analyze the problems of unconstrained nonlinear programming and to know the necessary and sufficient conditions for the solution of unconstrained problems.
• Find the best ways to crash project schedule, shortening total project duration and the ways to save money by adjusting activity durations and optimizing resource requirements in real life problems.

SUGGESTED ACTIVITIES

- Usage of MP Solver wrapper to solve LPP.
- Problem solving sessions
- Smart Class room sessions

SUGGESTED EVALUATION METHODS

- Problem solving in Tutorial sessions
- Assignment problems
- Quizzes and class test
- Discussion in classroom

Text Book(s):	
1.	Hamdy A Taha, Operations Research: An Introduction, Prentice Hall India, Tenth Edition, 2019.
2.	Hwei Hsu, "Schaums Outline of Theory and Problems of Probability, Random Variables and Random Processes", Tata Mcgraw Hill Edition, New Delhi, 1997.
3.	S.Boyd and L.Vandenberghe, Convex optimization, Cambridge University press,2004.
4.	John Myles White , Bandit algorithm for website Optimization, O' Riley Media, 2012.

Reference Books(s) / Web links:	
	Paneerselvam R., Operations Research, Prentice Hall of India, Fourth Print,2008.
1.	G. Srinivasan, Operations Research – Principles and Applications, 2nd edition, PHI, 2011.
	F.S. Hiller and G.J. Lieberman, Introduction to Operations Research, McGraw-Hill, Year: 2001
2.	Katta G. Murty Linear Programming, John Wiley & Sons, 1983.

RAJALAKSHMI ENGINEERING COLLEGE (AUTONOMOUS), THANDALAM

MA23434– OPTIMIZATION TECHNIQUES FOR AI

CO - PO – PSO matrices of course

1: Slight (Low)

2: Moderate (Medium)

3: Substantial (High) If there is no correlation, put "-"

B.Tech. AI&DS

PO/PSO CO	PO1	PO2	PO3	PO4	PO5	PO6	PO7	PO8	PO9	PO10	PO11	PO12	PSO1	PSO2	PSO3
MA23434.1	2	2	2	-	-	-	-	-	-	-	-	-	-	-	-
MA23434.2	2	2	-	-	-	-	-	-	-	-	1	-	-	-	-
MA23434. 3	2	1	2	-	1	-	-	-	-	-	-	-	-	-	-
MA23434. 4	2	2	-	-	-	-	-	-	-	-	2	-	-	-	-
MA23434.5	2	1	-	-	-	-	-	-	-	-	1	-	-	-	-
Average	2	1.6	2	-	1	-	-	-	-	-	1.3	-	-	-	-

AIML

PO/PSO CO	PO1	PO2	PO3	PO4	PO5	PO6	PO7	PO8	PO9	PO10	PO11	PO12	PSO1	PSO2	PSO3
MA23434.1	2	2	2	-	-	-	-	-	-	-	-	-	-	-	-
MA23434. 2	2	2	-	-	-	-	-	-	-	-	1	-	-	-	-
MA23434. 3	2	1	2	-	2	-	-	-	-	-	-	-	-	-	-
MA23434. 4	2	2	-	-	-	-	-	-	-	-	2	-	-	-	-
MA23434.5	2	1	-	-	-	-	-	-	-	-	1	-	-	-	-
Average	2	1.6	2	-	2	-	-	-	-	-	1.3	-	-	-	-

S.No	List of Experiment (using Python Programming)	Total Contact Hours: 30
1	Linear programming Problem- Constraint Optimization	
2	Transportation Problem	
3	Assignment Problem	
4	Integer Programming Problem- Branch and bound method	
5	Dynamic programming – Knapsack problem, Subset sum problem, longest common subsequence problem	
6	Gradient Descent Method- Stochastic Gradient Descent Algorithm	
7	Unconstrained Optimization- Nonlinear Least squares	
8	Kuhn-Tucker conditions - Lagrangian Multiplier method	
9	CPM -Analysis	
10	PERT -Analysis	

EXP 1	LINEAR PROGRAMMING PROBLEM- CONSTRAINT OPTIMIZATION

AIM:

To solve a Linear Programming Problem (LPP) for constraint optimization using Python.

PROCEDURE:

1. **Install Required Libraries:**
   - Make sure you have numpy and scipy installed. You can install them using pip if not already installed.

     bash
     Copy code
     pip install numpy scipy

2. **Define the Problem:**
   - Identify the objective function to maximize or minimize.
   - Determine the constraints (equality and inequality).
3. **Set Up the Problem in Python:**
   - Use the scipy.optimize.linprog function to set up and solve the LPP.
4. **Run the Code:**
   - Execute the Python code to get the optimized result.
5. **Analyze the Output:**
   - Interpret the results and validate the solution against the problem constraints.

CODE:

Code to solve a Linear Programming Problem using Python.

```
import numpy as np
from scipy.optimize import linprog

# Define the coefficients of the objective function
# For example, maximize: z = 3x1 + 2x2
c = [-3, -2]
```

```
# Define the coefficients of the inequality constraints
# For example:
# 2x1 + x2 <= 20
# 4x1 - 5x2 <= 10
# x1, x2 >= 0 (non-negativity constraints)

A = [
    [2, 1],
    [4, -5]
]

# Define the right-hand side of the inequality constraints
b = [20, 10]

# Define bounds for variables (non-negativity constraints)
x_bounds = (0, None)
y_bounds = (0, None)

# Solve the linear programming problem
result = linprog(c, A_ub=A, b_ub=b, bounds=[x_bounds, y_bounds], method='highs')

# Print the result
print('Optimal value:', -result.fun)
print('Values of x:', result.x)
```

EXPECTED OUTPUT:

```
Optimal value: 14.0
Values of x: [10. 0.]
```

VIVA QUESTIONS:

1. What is a Linear Programming Problem (LPP)?
2. Explain the significance of the objective function in LPP.
3. How do you formulate constraints in an LPP?
4. What does the linprog function in SciPy do?
5. Why do we use negative coefficients for the objective function in the linprog function?
6. Can you explain the concept of feasible and optimal solutions in LPP?
7. What are some real-world applications of linear programming?
8. How do you interpret the results obtained from the linprog function?

EXP 2	TRANSPORTATION PROBLEM

AIM:

To solve the Transportation Problem for minimizing the cost of transporting goods from multiple sources to multiple destinations using Python.

PROCEDURE:

1. **Install Required Libraries:**
   - Make sure you have numpy and scipy installed. You can install them using pip if not already installed.

     bash
     Copy code
     pip install numpy scipy

2. **Define the Problem:**
   - Identify the cost matrix, supply array, and demand array.
3. **Set Up the Problem in Python:**
   - Use the scipy.optimize.linprog function to set up and solve the Transportation Problem.
4. **Run the Code:**
   - Execute the Python code to get the optimized result.
5. **Analyze the Output:**
   - Interpret the results and validate the solution against the problem constraints.

CODE:

```
import numpy as np
from scipy.optimize import linprog

# Define the cost matrix
# Example cost matrix where rows represent sources and columns represent destinations
cost = np.array([
    [4, 8, 8],
    [2, 7, 6],
    [3, 4, 2]
])

# Flatten the cost matrix to use in linprog
c = cost.flatten()

# Define the supply array
```

```
supply = np.array([50, 40, 60])

# Define the demand array
demand = np.array([30, 70, 50])

# Create inequality constraints matrix (A_ub) and right-hand side (b_ub)
# Constraints: each source supplies to multiple destinations (<= supply) and
# each destination is supplied from multiple sources (>= demand)
A_eq = []
b_eq = []

# Supply constraints
for i in range(len(supply)):
    constraint = [0] * len(c)
    for j in range(len(demand)):
        constraint[i * len(demand) + j] = 1
    A_eq.append(constraint)
    b_eq.append(supply[i])

# Demand constraints
for j in range(len(demand)):
    constraint = [0] * len(c)
    for i in range(len(supply)):
        constraint[i * len(demand) + j] = 1
    A_eq.append(constraint)
    b_eq.append(demand[j])

# Solve the transportation problem using linprog
result = linprog(c, A_eq=A_eq, b_eq=b_eq, bounds=(0, None), method='highs')

# Reshape the result to the original cost matrix shape
result_matrix = result.x.reshape(cost.shape)

# Print the result
print('Optimal value (total cost):', result.fun)
print('Optimal transport plan:')
print(result_matrix)
```

EXPECTED OUTPUT:

```
Optimal value (total cost): 630.0
Optimal transport plan:
[[ 0. 30. 20.]
 [30.  0. 10.]
 [ 0. 40. 20.]]
```

VIVA QUESTIONS:

1. What is the Transportation Problem in linear programming?
2. Explain the significance of the cost matrix in the Transportation Problem.
3. How do you formulate supply and demand constraints in the Transportation Problem?
4. What does the linprog function in SciPy do in the context of the Transportation Problem?
5. How do you interpret the optimal transport plan obtained from the linprog function?
6. Can the Transportation Problem be solved using other methods besides linear programming? If yes, name a few.
7. Why is it important to minimize the cost in the Transportation Problem?
8. What are some real-world applications of the Transportation Problem?

LAB 3	ASSIGNMENT PROBLEM

Assignment Problem-Assignment with team of workers-Assignment with task size

AIM:

To understand and solve the assignment problem using python ,assignment team of workers.

PROCEDURE:

1. Input Data: Define the cost matrix where each element represents the cost of assigning a particular task to a specific worker.
2. Algorithm: Use an optimization algorithm like the Hungarian algorithm (or the Munkres algorithm) to find the optimal assignment that minimizes the total cost.
3. Output: Display the optimal assignment along with the minimum total cost.

CODE:

```
from scipy.optimize import linear_sum_assignment
# Cost matrix where each element (i, j) represents the cost of assigning task i to worker j
cost_matrix = [
[4, 1, 3],
[2, 0, 5],
[3, 2, 2]
]
# Using the Munkres algorithm to solve the assignment problem
row_indices, col_indices = linear_sum_assignment(cost_matrix)
# Output the optimal assignment and minimum cost
```

```
total_cost = cost_matrix[row_indices, col_indices].sum()

print(&quot;Optimal Assignment:&quot;)

for i, worker in enumerate(col_indices):

print(f&quot;Task {i+1} assigned to Worker {worker+1}&quot;)

print(f&quot;\nMinimum Total Cost: {total_cost}&quot;)
```

OUTPUT:

```
OPTIMAL ASSIGNMENT:

Task 1 assigned to worker 2

Task 2 assigned to worker 1

Task 3 assigned to worker 3

Minimum Total cost:3
```

VIVA QUESTIONS:

1. What is the assignment problem?
2. What are the key components of the assignment problem?
3. How does the assignment problem differ from other optimization problems?
4. What is the Munkres (Hungarian) algorithm?
5. How does the Munkres algorithm find the optimal assignment?
6. What are the advantages of using the Munkres algorithm for solving the assignment problem?
7. Can you demonstrate the implementation of the assignment problem in Python using the Munkres algorithm?
8. How do you interpret the output of the Munkres algorithm in the context of the assignment problem?

EXP 4	INTEGER PROGRAMMING PROBLEM- BRANCH AND BOUND METHOD

Solving Integer Programming Problems using the Branch and Bound Method

AIM:

To solve any integer programming problem using the Branch and Bound method in Python.

PROCEDURE:

1. Formulate the integer programming problem.
2. Implement the Branch and Bound method in Python.
3. Solve the problem using the implemented method.
4. Verify the solution.

PYTHON CODE:

```
import numpy as np
from scipy.optimize import linprog
from queue import Queue

# Function to solve the linear programming relaxation
def solve_lp(c, A, b, bounds):
    res = linprog(c, A_ub=A, b_ub=b, bounds=bounds, method='highs')
    return res

# Branch and Bound method
def branch_and_bound(c, A, b, bounds):
    Q = Queue()
    Q.put((c, A, b, bounds))
    best_solution = None
    best_value = float('-inf')

    while not Q.empty():
        current_problem = Q.get()
        res = solve_lp(*current_problem)

        if res.success and -res.fun > best_value:
            solution = res.x
            if all(np.isclose(solution, np.round(solution))):
                value = -res.fun  # Objective function value
                if value > best_value:
```

```
                best_value = value
                best_solution = solution
        else:
            # Branching
            for i in range(len(solution)):
                if not np.isclose(solution[i], np.round(solution[i])):
                    lower_bound = current_problem[3].copy()
                    upper_bound = current_problem[3].copy()

                    lower_bound[i] = (lower_bound[i][0], np.floor(solution[i]))
                    upper_bound[i] = (np.ceil(solution[i]), upper_bound[i][1])

                    Q.put((current_problem[0], current_problem[1], current_problem[2],
lower_bound))
                    Q.put((current_problem[0], current_problem[1], current_problem[2],
upper_bound))
                    break

    return best_solution, best_value

# Example usage
c = [-4, -3]  # Coefficients for the objective function (maximize)
A = [[2, 1], [1, 2]]  # Coefficients for the constraints
b = [8, 6]  # RHS values for the constraints
bounds = [(0, None), (0, None)]  # Bounds for the variables

# Solve the integer programming problem
solution, value = branch_and_bound(c, A, b, bounds)
print(f"Optimal solution: {solution}")
print(f"Optimal value: {value}")
```

VIVA QUESTIONS:

1. What is the Branch and Bound method in the context of integer programming?
2. How does the Branch and Bound method work?
3. Explain the difference between linear programming and integer programming.
4. What are the advantages and limitations of the Branch and Bound method?
5. Can the Branch and Bound method be used for problems with non-integer solutions? Why or why not?
6. Describe a situation where the Branch and Bound method might not perform well.
7. How can you ensure that the solution obtained is an optimal integer solution?
8. What is the role of the relaxation step in the Branch and Bound method?
9. How do you determine the branching variables in the Branch and Bound method?
10. Can you modify the provided Python program to solve a minimization problem instead of a maximization problem?

EXP 5	DYNAMIC PROGRAMMING – KNAPSACK PROBLEM, SUBSET SUM PROBLEM, LONGEST COMMON SUBSEQUENCE PROBLEM

Dynamic programming

AIM:

1. To maximize the total value of items included in a knapsack without exceeding its capacity, given weights and values of the items.
2. To determine if a subset of a given set of integers sums up to a specified value.
3. To find the length of the longest subsequence common to two sequences, where the subsequence maintains the order of elements.

PROCEDURE:

1. Knapsack Problem

Given weights and values of n items, put these items in a knapsack of capacity W to get the maximum total value in the knapsack.

```
def knapsack(weights, values, W):
    n = len(weights)
    dp = [[0 for _ in range(W + 1)] for _ in range(n + 1)]

    for i in range(n + 1):
        for w in range(W + 1):
            if i == 0 or w == 0:
                dp[i][w] = 0
            elif weights[i-1] <= w:
                dp[i][w] = max(values[i-1] + dp[i-1][w-weights[i-1]], dp[i-1][w])
            else:
                dp[i][w] = dp[i-1][w]

    return dp[n][W]

# Example usage:
weights = [1, 3, 4, 5]
values = [1, 4, 5, 7]
W = 7
print(knapsack(weights, values, W))
  # Output: 9
```

2. Subset Sum Problem

Determine if there is a subset of the given set with a sum equal to a given sum.

```
def subset_sum(arr, sum):
    n = len(arr)
    dp = [[False for _ in range(sum + 1)] for _ in range(n + 1)]

    for i in range(n + 1):
        dp[i][0] = True

    for i in range(1, n + 1):
        for s in range(1, sum + 1):
            if arr[i-1] <= s:
                dp[i][s] = dp[i-1][s] or dp[i-1][s-arr[i-1]]
            else:
                dp[i][s] = dp[i-1][s]

    return dp[n][sum]

# Example usage:
arr = [3, 34, 4, 12, 5, 2]
sum = 9
print(subset_sum(arr, sum))
 # Output: True
```

3. Longest Common Subsequence Problem

Given two sequences, find the length of the longest subsequence present in both of them.

```
def lcs(X, Y):
    m = len(X)
    n = len(Y)
    dp = [[0 for _ in range(n + 1)] for _ in range(m + 1)]

    for i in range(m + 1):
        for j in range(n + 1):
            if i == 0 or j == 0:
                dp[i][j] = 0
            elif X[i-1] == Y[j-1]:
                dp[i][j] = dp[i-1][j-1] + 1
            else:
                dp[i][j] = max(dp[i-1][j], dp[i][j-1])

    return dp[m][n]

# Example usage:
X = "AGGTAB"
Y = "GXTXAYB"
print(lcs(X, Y))
# Output: 4
```

VIVA QUESTIONS:

1. What is the Knapsack Problem?
2. What is dynamic programming and how is it applied in the Knapsack Problem?
3. What is the Subset Sum Problem?
4. What are the base cases in the dynamic programming approach to the Subset Sum Problem?
5. What is the Longest Common Subsequence (LCS) Problem?

LAB 6	GRADIENT DESCENT METHOD- STOCHASTIC GRADIENT DESCENT ALGORITHM

Gradient Descent method

AIM:

To iteratively adjust the parameters of a model in order to minimize the error between the predicted and actual values by using the Stochastic Gradient Descent (SGD) algorithm. SGD updates the model parameters based on each training example, allowing for faster convergence and handling larger datasets efficiently compared to batch gradient descent.

PROCEDURE:

1. **Initialization**:
    1. We initialize the weights to zeros.
    2. X is the feature matrix, and y is the target vector.
    3. learning_rate controls the step size of each update.
    4. epochs is the number of iterations over the entire dataset.
2. **Gradient Computation**:
    1. For each sample in the dataset, we compute the gradient of the loss with respect to the weights.
    2. The gradient is calculated as (prediction - target) * feature.
3. **Weights Update**:
    1. We update the weights by moving them in the opposite direction of the gradient, scaled by the learning rate.

Note:

- The data X is a matrix where each row is a sample and each column is a feature.
- The target values y are reshaped to ensure they are in the correct format for the computations.
- In the example, we generate some synthetic data for linear regression with added noise.

```
import numpy as np

def stochastic_gradient_descent(X, y, learning_rate=0.01, epochs=1000):
    """
    Perform Stochastic Gradient Descent to learn the weights for linear regression.

    Parameters:
    X: numpy array, shape (n_samples, n_features)
      Training data.
    y: numpy array, shape (n_samples,)
      Target values.
    learning_rate: float
      The learning rate.
    epochs: int
      Number of iterations over the training data.

    Returns:
    weights: numpy array, shape (n_features,)
      The learned weights.
    """
    n_samples, n_features = X.shape
    weights = np.zeros(n_features)

    for epoch in range(epochs):
        for i in range(n_samples):
            gradient = (np.dot(X[i], weights) - y[i]) * X[i]
            weights -= learning_rate * gradient

    return weights

# Example usage:
if __name__ == "__main__":
    # Generating some example data
    np.random.seed(0)
    X = 2 * np.random.rand(100, 1)
    y = 4 + 3 * X + np.random.randn(100, 1)
    X_b = np.c_[np.ones((100, 1)), X]  # Add x0 = 1 to each instance

    # Reshape y
    y = y.ravel()

    # Parameters
    learning_rate = 0.01
    epochs = 1000

    # Running stochastic gradient descent
    weights = stochastic_gradient_descent(X_b, y, learning_rate, epochs)
    print(f"Weights: {weights}")
```

VIVA QUESTIONS:

1. What is Gradient Descent?
2. What is Stochastic Gradient Descent (SGD)?
3. How does SGD differ from Batch Gradient Descent?
4. What are the key hyperparameters in the SGD algorithm?
5. What is the role of the cost function in Gradient Descent?

EXP 7	UNCONSTRAINED OPTIMIZATION- NONLINEAR LEAST SQUARES

AIM:

The objective of this lab is to understand and implement optimization techniques for solving nonlinear least squares problems without constraints.

PROCEDURE:

a. **Problem Formulation:**

- Define the nonlinear least squares objective function f(x)f(x)f(x).
- Specify the data points and the model that relates to the data.

b. **Choose an Optimization Library:**

- Select an appropriate optimization library (e.g., scipy.optimize in Python).

c. **Implementing the Optimization:**

- Set up the objective function to be minimized.
- Define any constraints (if the problem extends to constrained NLS).

d. **Running Optimization:**

- Execute the chosen optimization algorithm.
- Monitor convergence and iterate as necessary.

e. **Post-optimization Analysis:**

- Evaluate and interpret the results obtained from the optimization.
- Compare with initial assumptions and data.

EXAMPLES:

```
import numpy as np
from scipy.optimize import least_squares
def model(x, t):
   # Define the model function to be fit (e.g. exponential decay)
   return x[0] * np.exp(-x[1] * t)
def residual(x, t, y):
   # Define the residual function (e.g. squared error)
```

```
    return model(x, t) - y
# Generate some data with noise
np.random.seed(123)
t = np.linspace(0, 1, 50)
y_true = [2.0, 0.5]  # true parameters
y = model(y_true, t) + 0.1 * np.random.randn(50)
# Fit the model to the data using nonlinear least squares
x0 = [1.0, 1.0]  # initial guess for parameters
result = least_squares(residual, x0, args=(t, y))
# Print the optimized parameters
print("Optimized parameters:", result.x)
```

OUTPUT:

```
Optimized parameters: [1.98254019 0.47917343]
```

2. **Nonlinear least squares for fitting a logistic growth model to data**

```
import numpy as np
from scipy.optimize import least_squares
def model(x, t):
    # Define the logistic growth model function to be fit
    return x[0] / (1 + np.exp(-x[1] * (t - x[2])))
def residual(x, t, y):
    # Define the residual function (e.g. squared error)
    return model(x, t) - y
# Generate some data with noise
np.random.seed(456)
t = np.linspace(0, 5, 100)
y_true = [5.0, 0.8, 3.0]  # true parameters
y = model(y_true, t) + 0.1 * np.random.randn(100)
# Fit the model to the data using nonlinear least squares
x0 = [1.0, 1.0, 1.0]  # initial guess for parameters
result = least_squares(residual, x0, args=(t, y))
# Print the optimized parameters
print("Optimized parameters:", result.x)
```

OUTPUT:

```
Optimized parameters: [1.98254019 0.47917343]
```

3. **Nonlinear least squares for fitting a sum of sine waves to data**

```
import numpy as np
from scipy.optimize import least_squares
```

```
def model(x, t):
    # Define the sum of sine waves model function to be fit
    return x[0]*np.sin(x[1]*t + x[2]) + x[3]*np.sin(x[4]*t + x[5])

def residual(x, t, y):
    # Define the residual function (e.g. squared error)
    return model(x, t) - y
# Generate some data with noise
np.random.seed(789)
t = np.linspace(0, 10, 200)
y_true = [2.0, 0.5, 1.0, 1.0, 2.0, 2.5]  # true parameters
y = model(y_true, t) + 0.1 * np.random.randn(200)
# Fit the model to the data using nonlinear least squares
x0 = [1.0, 1.0, 1.0, 1.0, 1.0, 1.0]  # initial guess for parameters
result = least_squares(residual, x0, args=(t, y))
# Print the optimized parameters
print("Optimized parameters:", result.x)
```

OUTPUT:

```
Optimized parameters: [0.97587131 0.54470271 0.77416988 0.97586095 0.54475034
0.77408607]
```

VIVA QUESTIONS:

1. What is the difference between linear least squares and nonlinear least squares?
2. Can you define the objective function in the context of nonlinear least squares?
3. How does the Gauss-Newton method differ from the Levenberg-Marquardt method in solving nonlinear least squares problems?
4. How do you choose an appropriate model for a given set of data in nonlinear least squares?
5. What are the criteria for evaluating the goodness of fit in nonlinear least squares?
6. How can overfitting or underfitting occur in the context of nonlinear least squares?

EXP 8	KUHN-TUCKER CONDITIONS LAGRANGIAN MULTIPLIER METHOD

AIM:

The objective of this lab is to introduce and implement the Kuhn-Tucker conditions for constrained optimization using Python. Participants will learn the theoretical basis of the Kuhn-Tucker conditions and apply them to solve optimization problems with inequality constraints.

PROCEDURE:

a. **Objective Function and Constraints:**
   - Define the objective function and inequality constraints in Python.

b. **Kuhn-Tucker Conditions:**
   - Implement the Kuhn-Tucker conditions using Python functions.
   - Compute the gradients of the objective function and constraints.

c. **Optimization Setup:**
   - Use scipy.optimize.minimize or another suitable optimization function.
   - Incorporate the Kuhn-Tucker conditions as additional constraints or through custom callback functions.

d. **Running the Optimization:**
   - Execute the optimization procedure and monitor convergence.
   - Validate results against theoretical expectations.

Examples:

Source code 1:

```
from scipy.optimize import minimize
# Define the objective function
def objective(x):
   return -(2*x[0]**2 + x[1]**2 + x[0]*x[1] + x[0] + x[1])
# Define the inequality constraint function
def constraint(x):
   return x[0]**2 + x[1]**2 - 1
# Kuhn-Tucker conditions
def kuhn_tucker_conditions(solution, gradient, constraint_value, constraint_derivative):
   # Stationarity condition
   stationarity = all([g * s == 0 for g, s in zip(gradient, solution)])
   # Primal feasibility
```

```
    primal_feasibility = constraint_value <= 0
    # Dual feasibility
    dual_feasibility = all([l >= 0 for l in constraint_derivative])
    # Complementary slackness
    complementary_slackness = all([l * c == 0 for l, c in zip(constraint_derivative,
solution)])
    return stationarity, primal_feasibility, dual_feasibility, complementary_slackness
# Solve the optimization problem using scipy minimize
initial_guess = [0.5, 0.5]
bounds = [(None, None), (None, None)]
constraint_obj = {'type': 'ineq', 'fun': constraint}
result = minimize(objective, initial_guess, bounds=bounds, constraints=constraint_obj)
opt_solution = result.x
opt_value = result.fun
# Calculate gradients and constraint derivatives
from scipy.optimize import approx_fprime
gradient = approx_fprime(opt_solution, objective, epsilon=1e-6)
constraint_value = constraint(opt_solution)
constraint_derivative = approx_fprime(opt_solution, constraint, epsilon=1e-6)
# Check Kuhn-Tucker conditions
stationarity, primal_feasibility, dual_feasibility, complementary_slackness =
kuhn_tucker_conditions(
    opt_solution, gradient, constraint_value, constraint_derivative)

print("Optimal Solution:", opt_solution)
print("Optimal Objective Value:", opt_value)
print("Stationarity:", stationarity)
print("Primal Feasibility:", primal_feasibility)
print("Dual Feasibility:", dual_feasibility)
print("Complementary Slackness:", complementary_slackness)
```

Output:

```
Optimal Solution: [5.68355079e+160 3.08302943e+160]
Optimal Objective Value: -inf
Stationarity: False
Primal Feasibility: False
Dual Feasibility: False
Complementary Slackness: False
<ipython-input-1-4051017acb99>:5: RuntimeWarning: overflow encountered in scalar
power
  return -(2*x[0]**2 + x[1]**2 + x[0]*x[1] + x[0] + x[1])
<ipython-input-1-4051017acb99>:5: RuntimeWarning: overflow encountered in scalar
multiply
```

```
  return -(2*x[0]**2 + x[1]**2 + x[0]*x[1] + x[0] + x[1])
<ipython-input-1-4051017acb99>:9: RuntimeWarning: overflow encountered in scalar
power
  return x[0]**2 + x[1]**2 - 1
/usr/local/lib/python3.10/dist-packages/scipy/optimize/_numdiff.py:576:
RuntimeWarning: invalid value encountered in subtract
  df = fun(x) - f0
```

Source code:

```
from scipy.optimize import minimize
import numpy as np
# define the objective function and constraints
def objective_function(x):
    return -x[0] * x[1]
def constraint1(x):
    return x[0]**2 + x[1]**2 - 10
def constraint2(x):
    return x[0] - x[1]
# set the initial guess for the optimization
x0 = [1, 1]
# define the Lagrangian function
def lagrangian(x, c1, c2, l1, l2):
    return objective_function(x) + l1 * c1(x) + l2 * c2(x)
# set the initial Lagrange multipliers to zero
l1 = 0
l2 = 0
# perform the optimization using the Lagrangian Multiplier Method
result = minimize(lambda x: lagrangian(x, constraint1, constraint2, l1, l2), x0,
method='SLSQP', constraints=[{'type': 'ineq', 'fun': constraint1}, {'type': 'eq', 'fun':
constraint2}])
# print the optimized values and Lagrange multipliers
print('Optimized Values:', result.x)
print('Lagrange Multipliers:', result['jac'])
```

Output:

```
Optimized Values: [3.13313166e+147 3.13313166e+147]
Lagrange Multipliers: [-3.13313166e+147 -3.13313166e+147]
```

Source Code :

```
from scipy.optimize import minimize
          # Define the objective function
```

```
def objective(x):
    return -(2*x[0]**2 + x[1]**2 + x[0]*x[1] + x[0] + x[1])
# Define the inequality constraint function
def constraint(x):
    return x[0]**2 + x[1]**2 - 1
# Function to compute the Lagrangian
def lagrangian(x, lambda_):
    return objective(x) + lambda_ * constraint(x)
# Function to compute gradients of the Lagrangian
def lagrangian_gradients(x, lambda_):
    grad_objective = [4*x[0] + x[1] + 1, 2*x[1] + x[0] + 1]
    grad_constraint = [2*x[0], 2*x[1]]
    return [g + lambda_ * gc for g, gc in zip(grad_objective, grad_constraint)]
# Function to check Kuhn-Tucker conditions using Lagrangian multipliers
def kuhn_tucker_lagrangian(solution, lambda_):
    # Compute gradients of the Lagrangian
    gradients = lagrangian_gradients(solution, lambda_)
        # Stationarity condition
    stationarity = all([g == 0 for g in gradients])
        # Primal feasibility
    primal_feasibility = constraint(solution) <= 0
        # Dual feasibility
    dual_feasibility = lambda_ >= 0

    # Complementary slackness
    complementary_slackness = lambda_ * constraint(solution) == 0
        return stationarity, primal_feasibility, dual_feasibility,
complementary_slackness
# Solve the optimization problem using scipy minimize with constraints
initial_guess = [0.5, 0.5]
bounds = [(None, None), (None, None)]
constraint_obj = {'type': 'ineq', 'fun': constraint}
result = minimize(lambda x: lagrangian(x, 0.0), initial_guess, bounds=bounds,
constraints=constraint_obj)
opt_solution = result.x
opt_value = result.fun
# Compute Lagrange multiplier (lambda)
lambda_ = result['fun']
# Check Kuhn-Tucker conditions using Lagrangian multipliers
stationarity, primal_feasibility, dual_feasibility, complementary_slackness =
kuhn_tucker_lagrangian(opt_solution, lambda_)
print("Optimal Solution:", opt_solution)
print("Optimal Objective Value:", opt_value)
```

```
    print("Stationarity:", stationarity)
    print("Primal Feasibility:", primal_feasibility)
    print("Dual Feasibility:", dual_feasibility)
    print("Complementary Slackness:", complementary_slackness)
```

Output:

```
Optimal Solution: [5.68355092e+153 3.08302950e+153]
Optimal Objective Value: -9.163312799108927e+307
Stationarity: False
Primal Feasibility: False
Dual Feasibility: False
Complementary Slackness: False
<ipython-input-6-8e1b69516b56>:5: RuntimeWarning: overflow encountered in scalar
power
  return -(2*x[0]**2 + x[1]**2 + x[0]*x[1] + x[0] + x[1])
<ipython-input-6-8e1b69516b56>:5: RuntimeWarning: overflow encountered in scalar
multiply
  return -(2*x[0]**2 + x[1]**2 + x[0]*x[1] + x[0] + x[1])
<ipython-input-6-8e1b69516b56>:9: RuntimeWarning: overflow encountered in scalar
power
  return x[0]**2 + x[1]**2 - 1
<ipython-input-6-8e1b69516b56>:13: RuntimeWarning: invalid value encountered in
scalar multiply
  return objective(x) + lambda_ * constraint(x)
<ipython-input-6-8e1b69516b56>:19: RuntimeWarning: overflow encountered in scalar
multiply
  return [g + lambda_ * gc for g, gc in zip(grad_objective, grad_constraint)]
<ipython-input-6-8e1b69516b56>:36: RuntimeWarning: overflow encountered in scalar
multiply
  complementary_slackness = lambda_ * constraint(solution) == 0
```

Source code: Non linear programming

```
from scipy.optimize import minimize
# Define the objective function
def objective(x):
   return -(2*x[0]**2 + x[1]**2 + x[0]*x[1] + x[0] + x[1])
# Define the inequality constraint function
def constraint(x):
   return x[0]**2 + x[1]**2 - 1
# Function to compute the Lagrangian for nonlinear programming
def lagrangian(x, lambda_):
   return objective(x) + lambda_ * constraint(x)
# Function to compute gradients of the Lagrangian for nonlinear programming
```

```
def lagrangian_gradients(x, lambda_):
    grad_objective = [4*x[0] + x[1] + 1, 2*x[1] + x[0] + 1]
    grad_constraint = [2*x[0], 2*x[1]]
    return [g + lambda_ * gc for g, gc in zip(grad_objective, grad_constraint)]
# Function to check Kuhn-Tucker conditions using Lagrangian for nonlinear
programming
def kuhn_tucker_lagrangian(solution, lambda_):
    # Compute gradients of the Lagrangian
    gradients = lagrangian_gradients(solution, lambda_)
        # Stationarity condition
    stationarity = all([g == 0 for g in gradients])
        # Primal feasibility
    primal_feasibility = constraint(solution) <= 0
        # Dual feasibility
    dual_feasibility = lambda_ >= 0
        # Complementary slackness
    complementary_slackness = lambda_ * constraint(solution) == 0
        return stationarity, primal_feasibility, dual_feasibility, complementary_slackness
# Solve the optimization problem using scipy minimize with constraints
initial_guess = [0.5, 0.5]
bounds = [(None, None), (None, None)]
constraint_obj = {'type': 'ineq', 'fun': constraint}

result = minimize(lambda x: lagrangian(x, 0.0), initial_guess, bounds=bounds,
constraints=constraint_obj)
opt_solution = result.x
opt_value = result.fun
# Compute Lagrange multiplier (lambda)
lambda_ = result['fun']
# Check Kuhn-Tucker conditions using Lagrangian for nonlinear programming
stationarity, primal_feasibility, dual_feasibility, complementary_slackness =
kuhn_tucker_lagrangian(opt_solution, lambda_)
print("Optimal Solution:", opt_solution)
print("Optimal Objective Value:", opt_value)
print("Stationarity:", stationarity)
print("Primal Feasibility:", primal_feasibility)
print("Dual Feasibility:", dual_feasibility)
print("Complementary Slackness:", complementary_slackness)
```

Output:

```
Optimal Solution: [5.68355092e+153 3.08302950e+153]
Optimal Objective Value: -9.163312799108927e+307
Stationarity: False
```

```
Primal Feasibility: False
Dual Feasibility: False
Complementary Slackness: False
<ipython-input-7-0093525e22ad>:5: RuntimeWarning: overflow encountered in scalar
power
  return -(2*x[0]**2 + x[1]**2 + x[0]*x[1] + x[0] + x[1])
<ipython-input-7-0093525e22ad>:5: RuntimeWarning: overflow encountered in scalar
multiply
  return -(2*x[0]**2 + x[1]**2 + x[0]*x[1] + x[0] + x[1])
<ipython-input-7-0093525e22ad>:9: RuntimeWarning: overflow encountered in scalar
power
  return x[0]**2 + x[1]**2 - 1
<ipython-input-7-0093525e22ad>:13: RuntimeWarning: invalid value encountered in
scalar multiply
  return objective(x) + lambda_ * constraint(x)
<ipython-input-7-0093525e22ad>:19: RuntimeWarning: overflow encountered in scalar
multiply
  return [g + lambda_ * gc for g, gc in zip(grad_objective, grad_constraint)]
<ipython-input-7-0093525e22ad>:36: RuntimeWarning: overflow encountered in scalar
multiply
  complementary_slackness = lambda_ * constraint(solution) == 0
```

VIVA QUESTIONS:

1. What are the Kuhn-Tucker conditions? Provide a brief explanation.
2. How do the Kuhn-Tucker conditions extend the ideas of the Lagrange multiplier method in optimization?
3. Explain the significance of each component of the Kuhn-Tucker conditions (complementary slackness, feasibility, gradient equality).
4. How are inequality constraints incorporated into the Kuhn-Tucker conditions framework?
5. What happens if the constraints are non-linear or non-convex in the context of the Kuhn-Tucker conditions?
6. Explain the role of Lagrange multipliers in the Kuhn-Tucker conditions. How are they interpreted geometrically?

EXP 9	**CPM -ANALYSIS**

AIM:

To perform Critical Path Method (CPM) Analysis for project scheduling and management using Python.

PROCEDURE:

1. **Install Required Libraries:**
   - Make sure you have `pandas`, `networkx`, and `matplotlib` installed. You can install them using pip if not already installed.

     ```
     bash
     Copy code
     pip install pandas networkx matplotlib
     ```

2. **Define the Problem:**
   - Identify the tasks, their durations, and dependencies.
3. **Set Up the Problem in Python:**
   - Use NetworkX to represent the project tasks and their dependencies.
   - Use the topological sort to identify the critical path.
4. **Run the Code:**
   - Execute the Python code to get the critical path and project duration.
5. **Analyze the Output:**
   - Interpret the results and validate the solution against the project schedule.

CODE:

```
python
Copy code
import pandas as pd
import networkx as nx
import matplotlib.pyplot as plt

# Define the tasks, durations, and dependencies
tasks = {
    'A': {'duration': 3, 'dependencies': []},
    'B': {'duration': 2, 'dependencies': ['A']},
    'C': {'duration': 4, 'dependencies': ['A']},
    'D': {'duration': 2, 'dependencies': ['B']},
    'E': {'duration': 3, 'dependencies': ['C']},
    'F': {'duration': 1, 'dependencies': ['D', 'E']}
}

# Create a directed graph
G = nx.DiGraph()

# Add tasks to the graph
for task, data in tasks.items():
    G.add_node(task, duration=data['duration'])
```

```
    for dep in data['dependencies']:
        G.add_edge(dep, task)

# Perform topological sort to determine the order of tasks
topological_order = list(nx.topological_sort(G))

# Initialize earliest start and finish times
earliest_start = {task: 0 for task in topological_order}
earliest_finish = {task: 0 for task in topological_order}

# Calculate earliest start and finish times
for task in topological_order:
    earliest_finish[task] = earliest_start[task] + G.nodes[task]['duration']
    for successor in G.successors(task):
        earliest_start[successor] = max(earliest_start[successor],
earliest_finish[task])

# Initialize latest start and finish times
latest_finish = {task: earliest_finish[task] for task in topological_order}
latest_start = {task: earliest_start[task] for task in topological_order}

# Calculate latest start and finish times
for task in reversed(topological_order):
    for predecessor in G.predecessors(task):
        latest_finish[predecessor] = min(latest_finish[predecessor],
latest_start[task])
        latest_start[predecessor] = latest_finish[predecessor] -
G.nodes[predecessor]['duration']

# Determine the critical path
critical_path = [task for task in topological_order if earliest_start[task]
== latest_start[task]]

# Print the results
print('Earliest start times:', earliest_start)
print('Earliest finish times:', earliest_finish)
print('Latest start times:', latest_start)
print('Latest finish times:', latest_finish)
print('Critical path:', critical_path)

# Plot the graph
pos = nx.spring_layout(G)
nx.draw(G, pos, with_labels=True, node_size=3000, node_color='lightblue')
labels = nx.get_node_attributes(G, 'duration')
nx.draw_networkx_edge_labels(G, pos, edge_labels={(u, v): f"{u} -> {v}" for
u, v in G.edges})
plt.show()
```

Expected Output:

```
Earliest start times: {'A': 0, 'B': 3, 'C': 3, 'D': 5, 'E': 7, 'F': 10}
Earliest finish times: {'A': 3, 'B': 5, 'C': 7, 'D': 7, 'E': 10, 'F': 11}
```

```
Latest start times: {'A': 0, 'B': 3, 'C': 3, 'D': 5, 'E': 7, 'F': 10}
Latest finish times: {'A': 3, 'B': 5, 'C': 7, 'D': 7, 'E': 10, 'F': 11}
Critical path: ['A', 'C', 'E', 'F']
```

VIVA QUESTIONS:

1. What is the Critical Path Method (CPM)?
2. Explain the significance of the critical path in project management.
3. How do you determine the earliest start and finish times for tasks in CPM?
4. What are the latest start and finish times, and why are they important?
5. How do you identify the critical path in a project schedule?
6. What are the benefits of using CPM for project scheduling?
7. Can a project have more than one critical path? Explain.
8. How do dependencies between tasks affect the project schedule in CPM?
9. What tools and software are commonly used for CPM analysis besides Python?
10. How does CPM help in resource allocation and project management?

EXP 10	PERT -ANALYSIS

AIM:

To perform Program Evaluation Review Technique (PERT) Analysis for project scheduling and management using Python.

PROCEDURE:

1. **Install Required Libraries:**
   - Make sure you have `pandas`, `networkx`, `matplotlib`, and `numpy` installed. You can install them using pip if not already installed.

     ```
     bash
     Copy code
     pip install pandas networkx matplotlib numpy
     ```

2. **Define the Problem:**
   - Identify the tasks, their optimistic, most likely, and pessimistic durations, and dependencies.
3. **Set Up the Problem in Python:**
   - Use NetworkX to represent the project tasks and their dependencies.
   - Calculate the expected duration and variance for each task.
   - Use topological sort to identify the critical path and perform PERT analysis.
4. **Run the Code:**
   - Execute the Python code to get the critical path, project duration, and associated uncertainty.
5. **Analyze the Output:**
   - Interpret the results and validate the solution against the project schedule.

CODE:

```
import numpy as np
import pandas as pd
import networkx as nx
import matplotlib.pyplot as plt

# Define the tasks, durations (optimistic, most likely, pessimistic), and
dependencies
tasks = {
    'A': {'optimistic': 2, 'most_likely': 4, 'pessimistic': 6,
'dependencies': []},
    'B': {'optimistic': 1, 'most_likely': 2, 'pessimistic': 3,
'dependencies': ['A']},
    'C': {'optimistic': 1, 'most_likely': 3, 'pessimistic': 5,
'dependencies': ['A']},
    'D': {'optimistic': 2, 'most_likely': 2, 'pessimistic': 4,
'dependencies': ['B']},
```

```
    'E': {'optimistic': 3, 'most_likely': 5, 'pessimistic': 7,
'dependencies': ['C']},
    'F': {'optimistic': 1, 'most_likely': 4, 'pessimistic': 6,
'dependencies': ['D', 'E']}
}

# Calculate expected duration and variance for each task
for task, data in tasks.items():
    data['expected_duration'] = (data['optimistic'] + 4 * data['most_likely']
+ data['pessimistic']) / 6
    data['variance'] = ((data['pessimistic'] - data['optimistic']) / 6) ** 2

# Create a directed graph
G = nx.DiGraph()

# Add tasks to the graph
for task, data in tasks.items():
    G.add_node(task, duration=data['expected_duration'],
variance=data['variance'])
    for dep in data['dependencies']:
        G.add_edge(dep, task)

# Perform topological sort to determine the order of tasks
topological_order = list(nx.topological_sort(G))

# Initialize earliest start and finish times
earliest_start = {task: 0 for task in topological_order}
earliest_finish = {task: 0 for task in topological_order}

# Calculate earliest start and finish times
for task in topological_order:
    earliest_finish[task] = earliest_start[task] + G.nodes[task]['duration']
    for successor in G.successors(task):
        earliest_start[successor] = max(earliest_start[successor],
earliest_finish[task])

# Initialize latest start and finish times
latest_finish = {task: earliest_finish[task] for task in topological_order}
latest_start = {task: earliest_start[task] for task in topological_order}

# Calculate latest start and finish times
for task in reversed(topological_order):
    for predecessor in G.predecessors(task):
        latest_finish[predecessor] = min(latest_finish[predecessor],
latest_start[task])
        latest_start[predecessor] = latest_finish[predecessor] -
G.nodes[predecessor]['duration']

# Determine the critical path
critical_path = [task for task in topological_order if earliest_start[task]
== latest_start[task]]

# Calculate project duration and standard deviation
project_duration = earliest_finish[topological_order[-1]]
project_variance = sum(G.nodes[task]['variance'] for task in critical_path)
project_std_dev = np.sqrt(project_variance)
```

```
# Print the results
print('Earliest start times:', earliest_start)
print('Earliest finish times:', earliest_finish)
print('Latest start times:', latest_start)
print('Latest finish times:', latest_finish)
print('Critical path:', critical_path)
print('Project duration (expected):', project_duration)
print('Project standard deviation:', project_std_dev)

# Plot the graph
pos = nx.spring_layout(G)
nx.draw(G, pos, with_labels=True, node_size=3000, node_color='lightblue')
labels = nx.get_node_attributes(G, 'duration')
nx.draw_networkx_edge_labels(G, pos, edge_labels={(u, v): f"{u} -> {v}" for
u, v in G.edges})
plt.show()
```

EXPECTED OUTPUT:

```
Earliest start times: {'A': 0, 'B': 4, 'C': 4, 'D': 6, 'E': 7, 'F': 10}
Earliest finish times: {'A': 4, 'B': 6, 'C': 7, 'D': 8, 'E': 12, 'F': 14}
Latest start times: {'A': 0, 'B': 4, 'C': 4, 'D': 6, 'E': 7, 'F': 10}
Latest finish times: {'A': 4, 'B': 6, 'C': 7, 'D': 8, 'E': 12, 'F': 14}
Critical path: ['A', 'C', 'E', 'F']
Project duration (expected): 14.0
Project standard deviation: 1.247219128924647
```

VIVA QUESTIONS:

1. What is the Program Evaluation Review Technique (PERT)?
2. Explain the significance of the three time estimates (optimistic, most likely, pessimistic) in PERT.
3. How do you calculate the expected duration and variance for each task in PERT?
4. What is the critical path, and why is it important in PERT analysis?
5. How do you interpret the project duration and standard deviation obtained from PERT analysis?
6. What are the benefits of using PERT for project scheduling?
7. How does PERT handle uncertainty in project schedules?
8. Can a project have multiple critical paths in PERT analysis? Explain.
9. What tools and software are commonly used for PERT analysis besides Python?
10. How does PERT help in project management and decision-making?