Dr DY Patil Unitech society's

DR. D. Y. PATIL ARTS, COMMERCE & SCIENCE COLLEGE

Pimpri Pune-411 018

CERTIFICATE

Laboratory Certificate

This is to certify ______________________________ of **M.Sc. Data Science,** exam Seat No. _________ has successfully completed his/her the practical's in the Subject _________________________ as laid down by Savitribai Phule University for academic year 20__-202__.

Checked by: **Principal**

____________ ____________

Internal Examiner **External Examiner**

__________________ __________________

INDEX

Date	Topic	Page

Practical -1 Implementation of simplex algorithm

```
from scipy.optimize import linprog

# Objective function coefficients

c = [-3, -5]  # Maximization of 3x + 5y, so we use -3 and -5 for minimization


# Coefficients of the inequality constraints (Ax <= b)

A = [[1, 0], [0, 2], [3, 2]]

b = [4, 12, 18]


# Boundaries of the decision variables (x, y >= 0)

x_bounds = (0, None)

y_bounds = (0, None)


# Solving the linear programming problem using the simplex method

result = linprog(c, A_ub=A, b_ub=b, bounds=[x_bounds, y_bounds], method='simplex')


# Output the results

print(f"Optimal value: {-result.fun}")

print(f"Decision variables: {result.x}")
```

OUTPUT:

Optimal value: 15.0

Decision variables: [2. 6.]

Practical -2 Linear Programming using PuLP in Python

```
import pulp

# Create a linear programming problem (Maximization)

lp_problem = pulp.LpProblem("Maximize_Profit", pulp.LpMaximize)

# Decision variables

x = pulp.LpVariable('x', lowBound=0)  # x >= 0

y = pulp.LpVariable('y', lowBound=0)  # y >= 0

# Objective function

lp_problem += 3 * x + 5 * y, "Maximize profit"

# Constraints

lp_problem += x <= 4

lp_problem += 2 * y <= 12

lp_problem += 3 * x + 2 * y <= 18

# Solve the problem

lp_problem.solve()

# Output the results

print(f"Status: {pulp.LpStatus[lp_problem.status]}")

print(f"Optimal value: {pulp.value(lp_problem.objective)}")

print(f"Decision variables: x = {x.varValue}, y = {y.varValue}")
```

OUTPUT:

```
Status: Optimal

Optimal value: 15.0

Decision variables: x = 2.0, y = 6.0
```

Practical -3 LPP by calling solve() method

```
from scipy.optimize import linprog

# Objective function coefficients (for minimization)
c = [-1, -2]  # Maximization, so we use negative values for minimization

# Coefficients for inequality constraints (Ax <= b)
A = [[2, 1], [1, 1], [1, 0]]
b = [20, 16, 8]

# Boundaries for decision variables (x, y >= 0)
bounds = [(0, None), (0, None)]

# Solve the LPP using the default method (Simplex)
solution = linprog(c, A_ub=A, b_ub=b, bounds=bounds, method="simplex")

# Output the results
print(f"Optimal value: {-solution.fun}")
print(f"Decision variables: {solution.x}")
```

OUTPUT:

```
Optimal value: 14.0
Decision variables: [4.  8.]
```

Practical -4 PP model by declaring decision variables, list, objective function, and constraints

```
from pulp import LpMaximize, LpProblem, LpVariable

model = LpProblem("Maximize_Profit", LpMaximize)

x1 = LpVariable("x1", lowBound=0)  # x1 >= 0

x2 = LpVariable("x2", lowBound=0)  # x2 >= 0

# Objective function

model += 40 * x1 + 30 * x2, "Maximize Revenue"

# Constraints

model += 2 * x1 + x2 <= 60, "Material Constraint"

model += x1 + x2 <= 40, "Labor Constraint"

model += x1 <= 20, "Production Constraint"

# Solve the model

model.solve()

# Output results

print(f"Status: {pulp.LpStatus[model.status]}")

print(f"Optimal value: {pulp.value(model.objective)}")

print(f"x1 = {x1.varValue}, x2 = {x2.varValue}")
```

OUTPUT:

```
Status: Optimal

Optimal value: 1600.0

x1 = 20.0, x2 = 20.0
```

Practical -5 North-West corner method

```
def north_west_corner_method(c, b, A):
    # Initialize the transportation table
    T = [[0 for _ in range(len(b))] for _ in range(len(c))]

    # Initialize the remaining supply and demand
    remaining_supply = [c[i] for i in range(len(c))]
    remaining_demand = [b[i] for i in range(len(b))]

    # Start with the northwest corner
    i, j = 0, 0

    while i < len(c) and j < len(b):
        # Find the minimum of the remaining supply and demand
        min_remaining = min(remaining_supply[i], remaining_demand[j])

        # Update the transportation table
        T[i][j] = min_remaining

        # Update the remaining supply and demand
        remaining_supply[i] -= min_remaining
        remaining_demand[j] -= min_remaining

        # Move to the next cell
        if remaining_supply[i] == 0 and remaining_demand[j] == 0:
```

```
            i += 1
            j += 1
        elif remaining_supply[i] == 0:
            i += 1
        elif remaining_demand[j] == 0:
            j += 1

    return T

# Example input
supply = [20, 30, 25]
demand = [10, 40, 25]
cost = [[2, 3, 1], [5, 4, 8], [5, 6, 8]]

# Call the function
result = north_west_corner_method(supply, demand, cost)

# Print the result
for row in result:
    print(row)
```

OUTPUT

[10, 10, 0]

[0, 30, 0]

[0, 0, 25]

Practical -6 Least Cost Method

```
def least_cost_method(supply, demand, cost):
    rows = len(supply)
    cols = len(demand)
    allocation = [[0] * cols for _ in range(rows)]

    i = 0
    j = 0

    while i < rows and j < cols:
        if supply[i] < demand[j]:
            allocation[i][j] = supply[i]
            demand[j] -= supply[i]
            supply[i] = 0
            i += 1
        else:
            allocation[i][j] = demand[j]
            supply[i] -= demand[j]
            demand[j] = 0
            j += 1

    return allocation

def calculate_total_cost(allocation, cost):
    total_cost = 0
    for i in range(len(allocation)):
        for j in range(len(allocation[0])):
```

```
            total_cost += allocation[i][j] * cost[i][j]
    return total_cost


# Example usage
supply = [20, 30, 25]  # Supply for each source
demand = [30, 25, 20]  # Demand for each destination
cost = [[8, 6, 10],    # Cost matrix
        [9, 12, 13],
        [14, 9, 16]]


allocation = least_cost_method(supply, demand, cost)


print("Allocation Matrix:")
for row in allocation:
    print(row)


total_cost = calculate_total_cost(allocation, cost)
print(f"\nTotal Transportation Cost: {total_cost}")
```

OUTPUT:

```
Allocation Matrix:
[20, 0, 0]
[10, 20, 0]
[0, 5, 20]

Total Transportation Cost: 855
```

Practical -7 VAM

```
def vogels_approximation_method(supply, demand, cost):
    rows = len(supply)
    cols = len(demand)
    allocation = [[0] * cols for _ in range(rows)]

    while sum(supply) > 0 and sum(demand) > 0:
        # Calculate row and column penalties
        row_penalty = [max(cost[i]) - sorted(cost[i])[0] if supply[i] > 0 else float('inf') for i in
range(rows)]
        col_penalty = [max([cost[i][j] for i in range(rows)]) - min([cost[i][j] for i in range(rows)]) if
demand[j] > 0 else float('inf') for j in range(cols)]

        # Find the row/column with the highest penalty
        if min(row_penalty) <= min(col_penalty):
            i = row_penalty.index(min(row_penalty))
            j = cost[i].index(min(cost[i]))
        else:
            j = col_penalty.index(min(col_penalty))
            i = min(range(rows), key=lambda x: cost[x][j] if supply[x] > 0 else float('inf'))

        # Allocate as much as possible
        allocation_amount = min(supply[i], demand[j])
        allocation[i][j] = allocation_amount
        supply[i] -= allocation_amount
        demand[j] -= allocation_amount

    return allocation
```

```
def calculate_total_cost(allocation, cost):
    total_cost = 0
    for i in range(len(allocation)):
        for j in range(len(allocation[0])):
            total_cost += allocation[i][j] * cost[i][j]
    return total_cost

# Example usage
supply = [20, 30, 25]  # Supply for each source
demand = [30, 25, 20]  # Demand for each destination
cost = [[8, 6, 10],    # Cost matrix
        [9, 12, 13],
        [14, 9, 16]]

allocation = vogels_approximation_method(supply, demand, cost)

print("Allocation Matrix:")
for row in allocation:
    print(row)
total_cost = calculate_total_cost(allocation, cost)
print(f"\nTotal Transportation Cost: {total_cost}")
```

OUTPUT:

```
Allocation Matrix:
[0, 20, 0]
[30, 0, 0]
[0, 5, 20]

Total Transportation Cost: 755
```

Practical -8 Algebraic method for 2 by 2 game

```
import numpy as np
def solve_2x2_game(payoff_matrix):
    # Get elements from the payoff matrix
    a11, a12 = payoff_matrix[0]
    a21, a22 = payoff_matrix[1]

    # Solving for player A's strategy
    # Player A's probabilities: p1 for row 1 (A1) and p2 for row 2 (A2)
    # To find the optimal strategies, solve:
    # p1 * a11 + p2 * a21 = v (expected payoff)
    # p1 * a12 + p2 * a22 = v (expected payoff)

    # Solve for p1 and p2 using the algebraic method
    v = (a11 * a22 - a12 * a21) / (a11 + a22 - a12 - a21)
    p1 = (a22 - a12) / (a11 + a22 - a12 - a21)
    p2 = 1 - p1  # Since p1 + p2 = 1

    # Solving for player B's strategy
    q1 = (a22 - a21) / (a11 + a22 - a12 - a21)
    q2 = 1 - q1  # Since q1 + q2 = 1

    return {
        "Player A's strategy": (p1, p2),
        "Player B's strategy": (q1, q2),
        "Value of the game (v)": v
    }
```

```
# Example usage
payoff_matrix = [[3, 2],  # Payoffs for player A
                 [1, 4]]

result = solve_2x2_game(payoff_matrix)

print("Optimal Strategies:")
print(result)
```

OUTPUT:

```
Optimal Strategies:
{"Player A's strategy": (0.5, 0.5), "Player B's strategy":
(0.75, 0.25), 'Value of the game (v)': 2.5}
```

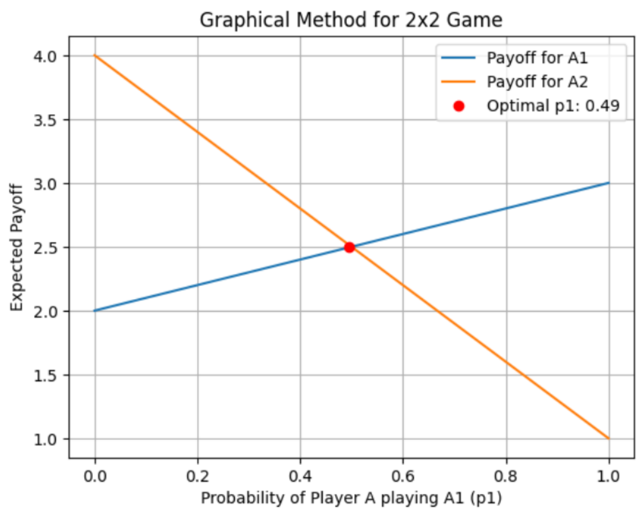
Practical -9 Graphical method for solving 2 by 2 game

```
import numpy as np
import matplotlib.pyplot as plt
def graphical_method(payoff_matrix):
    # Get payoffs from the matrix
    a11, a12 = payoff_matrix[0]
    a21, a22 = payoff_matrix[1]
    # Create an array of probabilities for Player A's first strategy (p1)
    p1_values = np.linspace(0, 1, 100)
    # Calculate expected payoffs for Player A's strategies A1 and A2
    payoff_A1 = p1_values * a11 + (1 - p1_values) * a12
    payoff_A2 = p1_values * a21 + (1 - p1_values) * a22
    # Find the intersection point (where the difference between payoffs is minimum)
    optimal_p1_index = np.argmin(abs(payoff_A1 - payoff_A2))
    optimal_p1 = p1_values[optimal_p1_index]
    optimal_value = payoff_A1[optimal_p1_index]
    # Plot the payoffs
    plt.plot(p1_values, payoff_A1, label="Payoff for A1")
    plt.plot(p1_values, payoff_A2, label="Payoff for A2")
    # Mark the optimal point
    plt.plot(optimal_p1, optimal_value, 'ro', label=f'Optimal p1: {optimal_p1:.2f}')

    # Add labels and legend
    plt.xlabel("Probability of Player A playing A1 (p1)")
    plt.ylabel("Expected Payoff")
    plt.title("Graphical Method for 2x2 Game")
    plt.legend()
```

```
    plt.grid(True)
    plt.show()
    # Calculate the optimal probability for Player B (complementary)
    optimal_p2 = 1 - optimal_p1
    return {
        "Player A's optimal strategy (p1)": optimal_p1,
        "Player A's complementary strategy (p2)": optimal_p2,
        "Value of the game (v)": optimal_value
    }
# Example usage
payoff_matrix = [[3, 2],  # Payoffs for Player A
                 [1, 4]]
result = graphical_method(payoff_matrix)
print("Optimal Strategy and Game Value:")
print(result)
```

OUTPUT:




```
Optimal Strategy and Game Value:
{"Player A's optimal strategy (p1)": 0.494949494949495, "Player A's complementary strategy (p2)": 0.505050505050505, 'Value of the game (v)': 2.49494949494
9495}
```

Practical -10 Game without saddle point

```
def solve_game_without_saddle_point(payoff_matrix):
    a11, a12 = payoff_matrix[0]
    a21, a22 = payoff_matrix[1]
    # Solve for Player A's strategy probabilities (p1 for A1, p2 for A2)
    # p1 * a11 + (1 - p1) * a12 = p1 * a21 + (1 - p1) * a22
    # Simplify the equation: (a11 - a12) * p1 + a12 = (a21 - a22) * p1 + a22
    p1 = (a22 - a12) / ((a11 - a12) - (a21 - a22))
    p2 = 1 - p1
    # Solve for Player B's strategy probabilities (q1 for B1, q2 for B2)
    # Similar logic as for Player A
    q1 = (a22 - a21) / ((a11 - a21) - (a12 - a22))
    q2 = 1 - q1
    return {
        "Player A's strategy": (p1, p2),
        "Player B's strategy": (q1, q2),
    }
# Example usage
payoff_matrix = [[1, 4],  # Payoff for Player A
                 [3, 2]]
result = solve_game_without_saddle_point(payoff_matrix)
print("Optimal Mixed Strategies:")
print(result)
```

OUTPUT:

```
Optimal Mixed Strategies:
{"Player A's strategy": (0.5, 0.5), "Player B's strategy": (0.25, 0.75)}
```

Practical -11 Two person zero sum game with saddle point

```
def find_saddle_point(payoff_matrix):
    # Find the minimum of each row (Player A's best worst-case scenario)
    row_minima = [min(row) for row in payoff_matrix]

    # Find the maximum of each column (Player B's worst best-case scenario)
    column_maxima = [max(col) for col in zip(*payoff_matrix)]

    # The saddle point exists if any of the row minima equals any of the column maxima
    saddle_points = []

    for i in range(len(payoff_matrix)):
        for j in range(len(payoff_matrix[i])):
            if payoff_matrix[i][j] == max(row_minima) and payoff_matrix[i][j] ==
min(column_maxima):
                saddle_points.append((i, j, payoff_matrix[i][j]))

    if saddle_points:
        return saddle_points
    else:
        return "No saddle point"

# Example usage
payoff_matrix = [[3, 2],  # Payoff for Player A
                 [1, 4]]

result = find_saddle_point(payoff_matrix)
```

```
if result != "No saddle point":
    print("Saddle Point(s) Found:")
    for point in result:
        print(f"Saddle point at row {point[0]+1}, column {point[1]+1} with value {point[2]}")
else:
    print(result)
```

OUTPUT:

No saddle point

Practical -12 Two person zero sum game without saddle point

```
def solve_zero_sum_game(payoff_matrix):
    # Extract the payoff matrix
    a11, a12 = payoff_matrix[0]
    a21, a22 = payoff_matrix[1]

    # Calculate Player A's mixed strategy probabilities
    p1 = (a22 - a12) / ((a11 - a12) + (a22 - a21))
    p2 = 1 - p1

    # Calculate Player B's mixed strategy probabilities
    q1 = (a22 - a21) / ((a11 - a21) + (a22 - a12))
    q2 = 1 - q1

    # Calculate the value of the game
    v = p1 * a11 + p2 * a21

    return {
        "Player A's strategy": (p1, p2),
        "Player B's strategy": (q1, q2),
        "Value of the game": v
    }

# Example usage
payoff_matrix = [[1, 4],  # Payoff for Player A
                 [3, 2]]
```

```
result = solve_zero_sum_game(payoff_matrix)

print("Optimal Mixed Strategies and Game Value:")

print(f"Player A's strategy: {result['Player A\'s strategy']}")

print(f"Player B's strategy: {result['Player B\'s strategy']}")

print(f"Value of the game: {result['Value of the game']:.2f}")
```

OUTPUT:

```
Optimal Mixed Strategies and Game Value:
Player A's strategy: (0.5, 0.5)
Player B's strategy: (0.25, 0.75)
Value of the game: 2.0
```