AM_Codes_Final

November 19, 2025

Practical No 1:- Perform matrix addition, multiplication, and scalar operations

[19]:
```
# Define two matrices 'a' and 'b'
a = [[7, 3],
     [2, 8]]

b = [[4, 6],
     [9, 2]]

# Define a scalar value
scalar_value = 7


# Matrix Addition Function

def add(x, y):
    # Create a result matrix filled with 0's (same size as x)
    result = [[0 for _ in range(len(x[0]))] for _ in range(len(x))]

    # Loop through rows
    for i in range(len(x)):
        # Loop through columns
        for j in range(len(x[0])):
            # Add corresponding elements
            result[i][j] = x[i][j] + y[i][j]

    return result



# Matrix Multiplication Function
def mul(A, B):
    result = []  # final result matrix

    # Loop through rows of A
    for i in range(len(A)):
        row = []
```

```
        # Loop through columns of B
        for j in range(len(B[0])):
            sum = 0

            # Multiply row of A by column of B
            for k in range(len(A[0])):
                sum += A[i][k] * B[k][j]

            row.append(sum)  # store result for one element

        result.append(row)  # add row to final matrix

    return result



# Scalar Multiplication Function

def scalar_mul(matrix, scalar):
    result = []  # final result matrix

    # Loop through rows
    for i in range(len(matrix)):
        row = []

        # Loop through columns
        for j in range(len(matrix[0])):
            # Multiply each element by scalar
            row.append(matrix[i][j] * scalar)

        result.append(row)  # add row to result

    return result

print("Addition is:", add(a, b))  # Element-wise addition
print("Multiplication is:", mul(a, b))  # Matrix multiplication
print(f"Scalar multiplication of 'a' with {scalar_value} is:", scalar_mul(a,␣
 ↪scalar_value))
print(f"Scalar multiplication of 'b' with {scalar_value} is:", scalar_mul(b,␣
 ↪scalar_value))
```

```
Addition is: [[11, 9], [11, 10]]
Multiplication is: [[55, 48], [80, 28]]
Scalar multiplication of 'a' with 7 is: [[49, 21], [14, 56]]
Scalar multiplication of 'b' with 7 is: [[28, 42], [63, 14]]
```

Practical No 2:- Compute eigenvalues/eigenvectors and diagonalize a matrix.

[20]:
```
import numpy as np


# Step 1: Define 2x2 matrix
A = np.array([[4, 1],
              [2, 3]])

print("Matrix A:")
print(A)


# Step 2: Compute eigenvalues manually
# For 2x2 matrix [[a, b],[c, d]]:
# Characteristic equation: |A - I| = 0 → ^2 - (a+d) + (ad-bc) = 0
a, b = A[0,0], A[0,1]
c, d = A[1,0], A[1,1]

trace = a + d
det = a*d - b*c

# Solve quadratic equation
lambda1 = (trace + np.sqrt(trace**2 - 4*det)) / 2
lambda2 = (trace - np.sqrt(trace**2 - 4*det)) / 2

print("\nEigenvalues:")
print(lambda1, lambda2)


# Step 3: Compute eigenvectors manually
# Solve (A - I)v = 0
# For 2x2: (a- )v1 + b*v2 = 0 → v2 = -((a- )/b)*v1

def eigenvector(A, lam):
    a, b = A[0,0], A[0,1]
    c, d = A[1,0], A[1,1]

    if b != 0:
        v1 = 1
        v2 = -((a - lam)/b)*v1
    elif c != 0:
        v2 = 1
        v1 = -((d - lam)/c)*v2
    else:  # diagonal matrix
        v1, v2 = 1, 0
    vec = np.array([v1, v2])
    return vec / np.linalg.norm(vec)  # normalize
```

```
v1 = eigenvector(A, lambda1)
v2 = eigenvector(A, lambda2)

print("\nEigenvectors (normalized):")
print("v1 =", v1)
print("v2 =", v2)
```

```
Matrix A:
[[4 1]
 [2 3]]

Eigenvalues:
5.0 2.0

Eigenvectors (normalized):
v1 = [0.70710678 0.70710678]
v2 = [ 0.4472136  -0.89442719]
```

Practical No 3:-Implement Gaussian elimination from scratch.

[21]:
```
# Step 1: Define the system of equations
# A is the coefficient matrix, B is the constants vector

A = [
    [1, 1, 1],
    [2, 3, 1],
    [1, 2, 3]
]
B = [6, 14, 14]
n = len(B)  # Number of equations/variables


# Step 2: Forward Elimination
# Convert A into an upper triangular matrix
for i in range(n):  # Loop over each pivot row
    for k in range(i+1, n):  # Loop over rows below pivot
        factor = A[k][i] / A[i][i]  # Divide by pivot element
        for j in range(i, n):  # Loop over columns (start from i for efficiency)
            A[k][j] -= factor * A[i][j]  # Subtract pivot row multiplied by␣
 ↪factor
        B[k] -= factor * B[i]  # Update the corresponding element in B

print("Matrix A after forward elimination (upper triangular):")
for row in A:
    print(row)
print("Modified B after forward elimination:")
print(B)
```

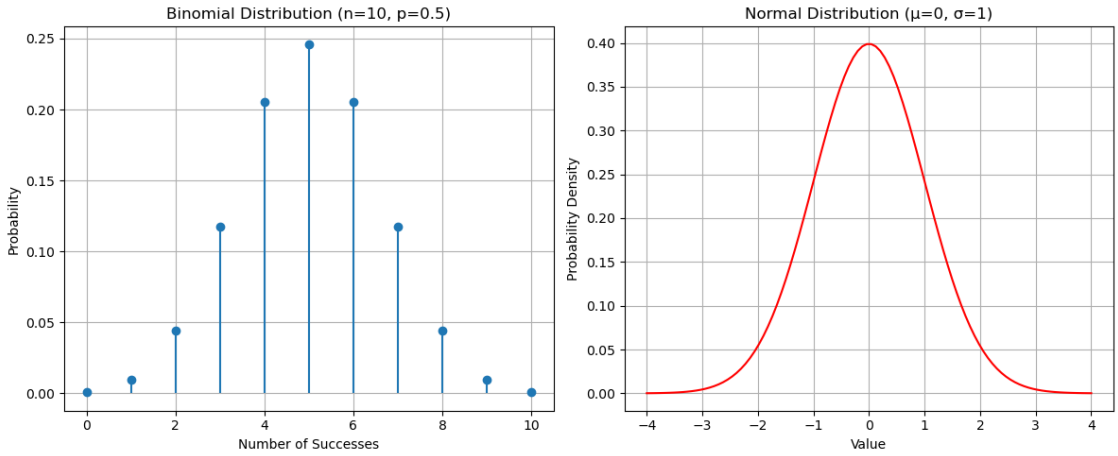
```
# Step 3: Back Substitution
# Solve for variables starting from the last row

x = [0] * n  # Initialize solution vector

for i in range(n-1, -1, -1):  # Start from last row and move upwards
    ax = 0
    for j in range(i+1, n):  # Sum of known variables multiplied by their␣
 ↪coefficients
        ax += A[i][j] * x[j]
    x[i] = (B[i] - ax) / A[i][i]  # Solve for current variable

print("Solution vector [x, y, z]:")
print(x)
```

```
Matrix A after forward elimination (upper triangular):
[1, 1, 1]
[0.0, 1.0, -1.0]
[0.0, 0.0, 3.0]
Modified B after forward elimination:
[6, 2.0, 6.0]
Solution vector [x, y, z]:
[0.0, 4.0, 2.0]
```

Practical No 4:-Demonstrate Discrete and Continuous Random Variables (Binomial and Normal Distributions)

[22]:
```
import numpy as np
import matplotlib.pyplot as plt
import scipy.stats as stats


# 1. BINOMIAL DISTRIBUTION


# Number of trials
n = 10
# Probability of success (here 0.25*2 = 0.5)
p = 0.25 * 2

# Possible values for number of successes (0, 1, ..., n)
x_bino = np.arange(0, n + 1)

# Probability Mass Function (PMF) for binomial distribution
# pmf_bino[i] = P(X = x_bino[i])
pmf_bino = stats.binom.pmf(x_bino, n, p)
```

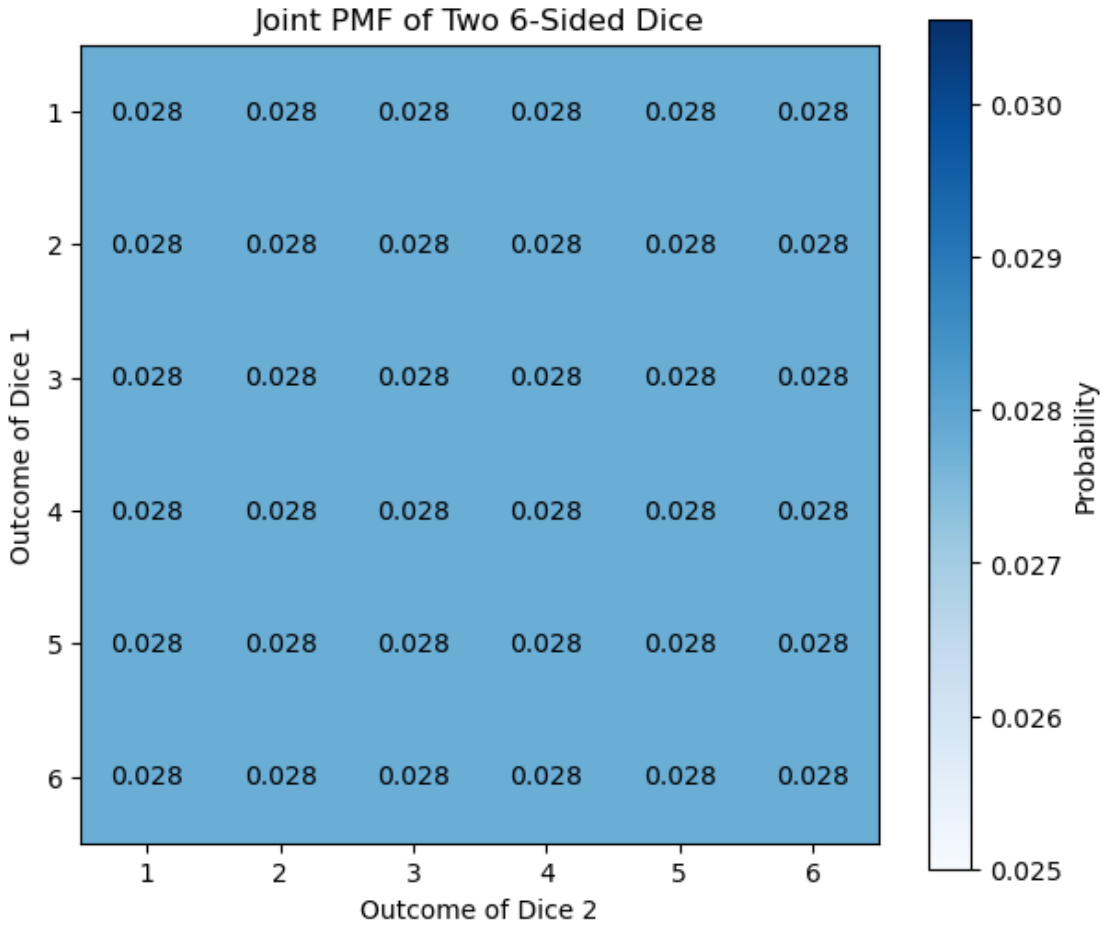
```
# 2. NORMAL DISTRIBUTION


# Mean ( ) and Standard Deviation ( )
mu = 0
sigma = 1

# Generate x values in range [-4, 4]
x_norm = np.linspace(-4, 4, 100)

# Probability Density Function (PDF) for normal distribution
# pdf_norm[i] = f(x_norm[i])
pmf_norm = stats.norm.pdf(x_norm, mu, sigma)



# 3. PLOTTING


plt.figure(figsize=(12, 5))   # Create a figure with width=12, height=5

# Binomial Distribution
plt.subplot(1, 2, 1)   # 1 row, 2 columns, 1st plot
plt.stem(x_bino, pmf_bino, basefmt=" ")  # Discrete stem plot
plt.title("Binomial Distribution (n=10, p=0.5)")
plt.xlabel("Number of Successes")
plt.ylabel("Probability")
plt.grid(True)

# Normal Distribution
plt.subplot(1, 2, 2)   # 1 row, 2 columns, 2nd plot
plt.plot(x_norm, pmf_norm, color="red")  # Continuous curve
plt.title("Normal Distribution ( =0,  =1)")
plt.xlabel("Value")
plt.ylabel("Probability Density")
plt.grid(True)

# Adjust layout so plots don't overlap
plt.tight_layout()
plt.show()
```



Practical No 5:-Implement Joint Probability Mass Function

[23]:
```
import numpy as np
import matplotlib.pyplot as plt


# 1. Define outcomes of a single die

die_outcomes = [1, 2, 3, 4, 5, 6]


# 2. Construct sample space of 2 dice
# Each element is a tuple (x, y)

sample_space = [(x, y) for x in die_outcomes for y in die_outcomes]


# 3. Joint PMF (all pairs equally likely)
# Since fair dice -> probability = 1/36 for each (x, y)
joint_pmf = {}
for (x, y) in sample_space:
    joint_pmf[(x, y)] = 1 / 36


# 4. Store PMF values in a 6x6 matrix
# pmf_matrix[i, j] corresponds to P(X=i+1, Y=j+1)

pmf_matrix = np.zeros((6, 6))
for x in die_outcomes:
    for y in die_outcomes:
        pmf_matrix[x-1, y-1] = joint_pmf[(x, y)]
```

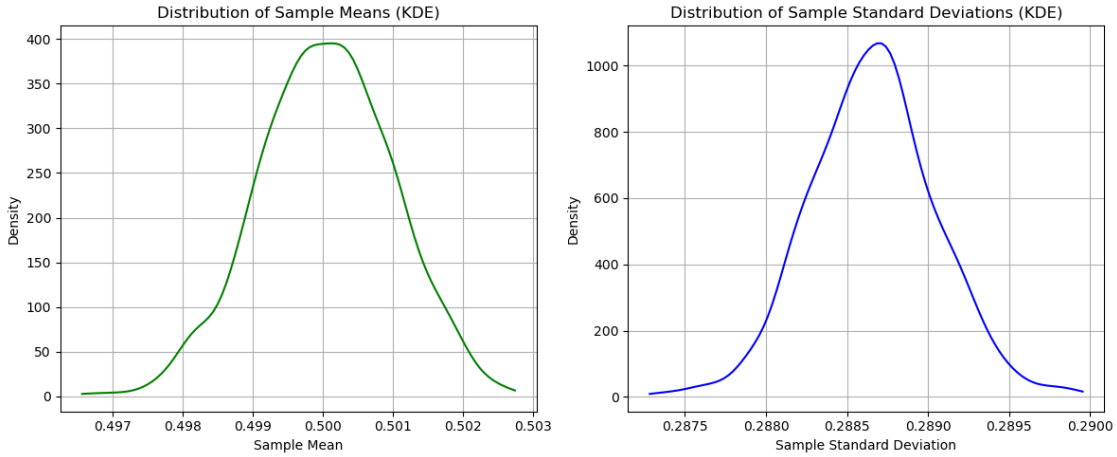
```python
# 5. Visualization (Heatmap with values)

plt.figure(figsize=(7, 6))
plt.imshow(pmf_matrix, cmap="Blues", interpolation="nearest")  # heatmap

# Overlay probability values on each cell
for i in range(6):
    for j in range(6):
        plt.text(j, i, f"{pmf_matrix[i, j]:.3f}",
                 ha="center", va="center", color="black")

plt.title("Joint PMF of Two 6-Sided Dice")
plt.xlabel("Outcome of Dice 2")
plt.ylabel("Outcome of Dice 1")
plt.colorbar(label="Probability")  # color bar scale
plt.xticks(np.arange(6), die_outcomes)
plt.yticks(np.arange(6), die_outcomes)
plt.show()
```



Practical No 6:- Implement central limit theorem and plot probability distribution

[24]:
```
import numpy as np
import matplotlib.pyplot as plt
import scipy.stats as stats


# Central Limit Theorem Demonstration


sample_size = 100000   # size of each sample (large number → better␣
 ↪approximation)
num_samples = 1000     # number of repeated samples

sample_means = []      # to store means of samples
sample_std_devs = []   # to store std. dev. of samples


# 1. Generate samples from Uniform(0,1) and compute their mean & std. dev.

for _ in range(num_samples):
    sample = np.random.uniform(0, 1, sample_size)  # uniform distribution
    sample_means.append(np.mean(sample))           # mean of sample
    sample_std_devs.append(np.std(sample))         # std dev of sample


# 2. Plot distribution of sample means
#    According to CLT -> should be Normal

plt.figure(figsize=(12, 5))

plt.subplot(1, 2, 1)
kde_means = stats.gaussian_kde(sample_means)              # Kernel Density␣
 ↪Estimate (smooth curve)
x_means = np.linspace(min(sample_means), max(sample_means), 100)
plt.plot(x_means, kde_means(x_means), color='g')
plt.title('Distribution of Sample Means (KDE)')
plt.xlabel('Sample Mean')
plt.ylabel('Density')
plt.grid(True)


# 3. Plot distribution of sample std devs
#  (less "normal", more spread-out)
```

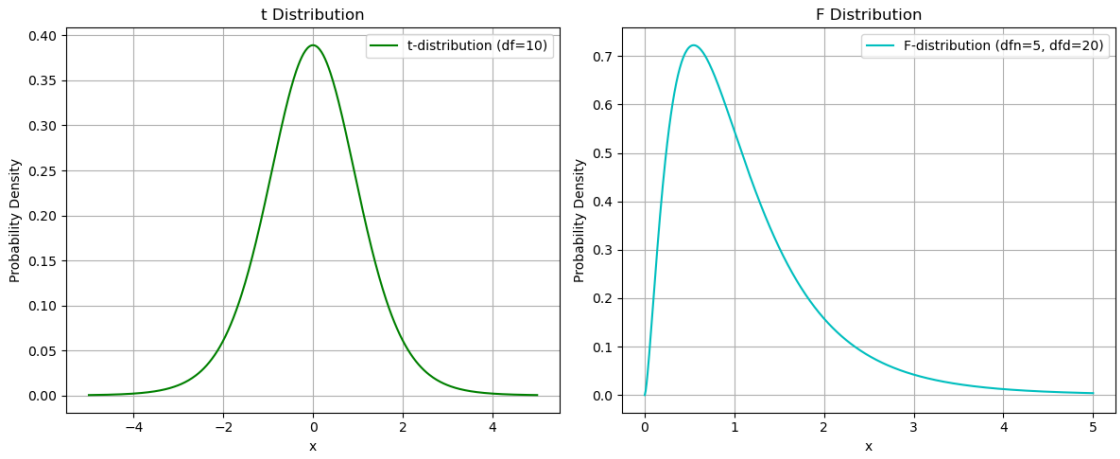
```
plt.subplot(1, 2, 2)
kde_std_devs = stats.gaussian_kde(sample_std_devs)
x_std_devs = np.linspace(min(sample_std_devs), max(sample_std_devs), 100)
plt.plot(x_std_devs, kde_std_devs(x_std_devs), color='b')
plt.title('Distribution of Sample Standard Deviations (KDE)')
plt.xlabel('Sample Standard Deviation')
plt.ylabel('Density')
plt.grid(True)

plt.tight_layout()
plt.show()
```



Practical No 7:- Generate t & f distribution

[25]:
```
import numpy as np
import matplotlib.pyplot as plt
from scipy.stats import t, f    # Import t and F distributions


# t-distribution

df_t = 10                                  # degrees of freedom for t-distribution
x_t = np.linspace(-5, 5, 500)              # x values (range for t-distribution)
pdf_t = t.pdf(x_t, df_t)                   # compute Probability Density Function␣
 ↪(PDF)


# F-distribution

dfn, dfd = 5, 20                           # numerator and denominator degrees of␣
 ↪freedom
```

```
x_f = np.linspace(0, 5, 500)              # x values (F distribution is >= 0)
pdf_f = f.pdf(x_f, dfn, dfd)              # compute PDF of F-distribution


# Plot both distributions

plt.figure(figsize=(12, 5))

# Plot t-distribution
plt.subplot(1, 2, 1)
plt.plot(x_t, pdf_t, 'g', label=f't-distribution (df={df_t})')
plt.title('t Distribution')
plt.xlabel('x')
plt.ylabel('Probability Density')
plt.legend()
plt.grid(True)

# Plot F-distribution
plt.subplot(1, 2, 2)
plt.plot(x_f, pdf_f, 'c', label=f'F-distribution (dfn={dfn}, dfd={dfd})')
plt.title('F Distribution')
plt.xlabel('x')
plt.ylabel('Probability Density')
plt.legend()
plt.grid(True)

plt.tight_layout()
plt.show()
```



Practical No 8:- Perform Large Sample Test for Single Mean (Z-test)

[26]:
```
import numpy as np
from scipy.stats import norm

# ----------------------------------------
# Function to perform one-sample Z-test for population mean
# ----------------------------------------
def z_test_single_mean(sample, mu0, sigma=None, alpha=0.05,␣
 ↪alternative="two-sided"):
    sample = np.array(sample)
    n = len(sample)  # Sample size
    x_bar = np.mean(sample)  # Sample mean

    # If population standard deviation (sigma) not given, use sample standard␣
 ↪deviation
    if sigma is None:
        sigma = np.std(sample, ddof=1)

    # Compute Z-statistic
    z_stat = (x_bar - mu0) / (sigma / np.sqrt(n))

    # Compute p-value based on test type
    if alternative == "two-sided":
        p_value = 2 * (1 - norm.cdf(abs(z_stat)))
    elif alternative == "greater":
        p_value = 1 - norm.cdf(z_stat)
    elif alternative == "less":
        p_value = norm.cdf(z_stat)
    else:
        raise ValueError("alternative must be 'two-sided', 'greater', or␣
 ↪'less'")

    # Decision: reject or fail to reject the null hypothesis
    reject = p_value <= alpha

    # Return test summary as a dictionary
    return {
        "Sample Mean": x_bar,
        "Hypothesized Mean": mu0,
        "Z-statistic": z_stat,
        "P-value": p_value,
        "Conclusion": "Reject H " if reject else "Fail to Reject H "
    }

# ----------------------------------------
# Example: Testing whether mean = 50
# ----------------------------------------
data = [52, 50, 53, 49, 48, 51, 54, 50, 52, 49]
```

```
mu0 = 50  # Hypothesized mean
alpha = 0.05  # Significance level

# Perform Z-test
result = z_test_single_mean(data, mu0, alpha=alpha, sigma=None,␣
 ↪alternative='two-sided')

# Display result
print(result)
```

```
{'Sample Mean': np.float64(50.8), 'Hypothesized Mean': 50, 'Z-statistic':
np.float64(1.3093073414159497), 'P-value': np.float64(0.19043026382552575),
'Conclusion': 'Fail to Reject H '}
```

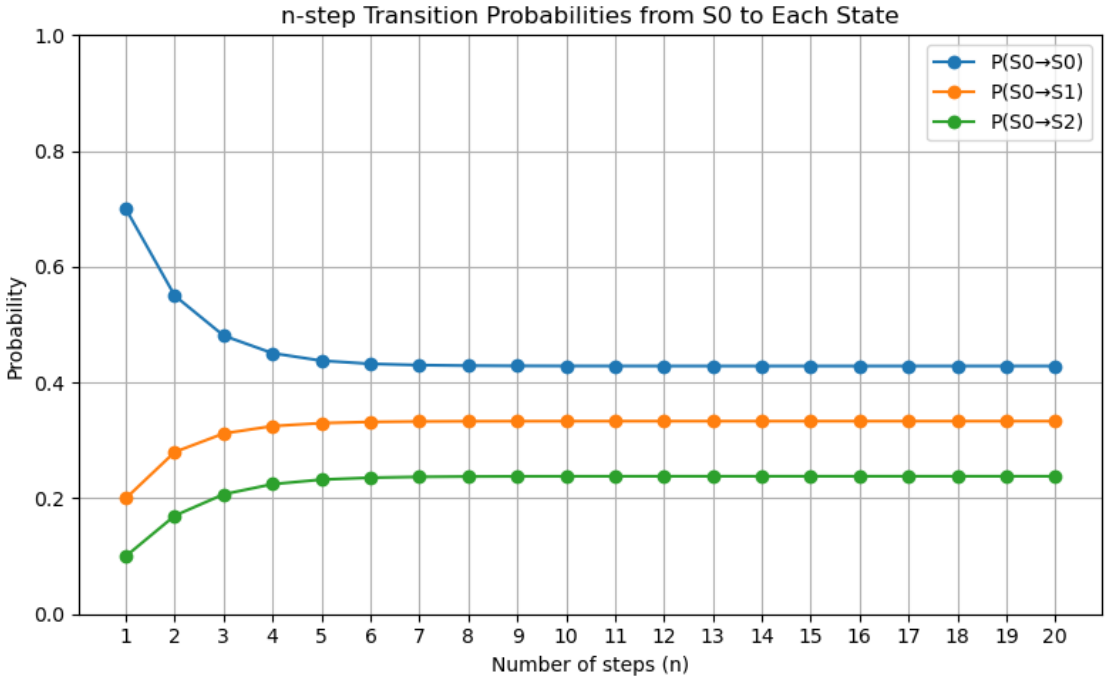
Practical No 9 :- Calculate n-Step Transition Probability

[27]:
```
import numpy as np
import matplotlib.pyplot as plt

# Function to validate a transition matrix
def validate_transition_matrix(P):
    P = np.array(P, dtype=float)  # Convert input to a float numpy array

    # Check if matrix is square
    if P.ndim != 2 or P.shape[0] != P.shape[1]:
        raise ValueError("P must be a square matrix")

    # Check for negative entries (not allowed in transition matrices)
    if np.any(P < 0):
        raise ValueError("Transition matrix cannot have negative entries.")

    # Check that each row sums to 1 (probabilities must add up to 1)
    if not np.allclose(P.sum(axis=1), 1.0):
        raise ValueError("Rows of P must sum to 1")

    return P


# Function to compute n-step transition probabilities
def n_step_transition_probability(P, n):
    P = validate_transition_matrix(P)  # Validate first
    return np.linalg.matrix_power(P, n)  # Compute P^n (matrix exponentiation)

# Main Program-
if __name__ == "__main__":
    # Define the transition matrix for a 3-state Markov chain
    P = np.array([
        [0.7, 0.2, 0.1],
```

```
        [0.1, 0.6, 0.3],
        [0.4, 0.2, 0.4]
    ])

    steps = [1, 2, 5, 10, 20]  # Number of steps to evaluate
    results = {}

    # Compute and print n-step transition probability matrices
    for n in steps:
        Pn = n_step_transition_probability(P, n)
        results[n] = Pn
        print(f"\n{n}-step transition probability matrix:\n{Pn}")

    # Track evolution of probabilities starting from state S0
    state_names = ["S0", "S1", "S2"]
    max_n = 20
    # Calculate transition probabilities from S0 to all other states for each n
    evolution = [n_step_transition_probability(P, n)[0] for n in range(1, max_n␣
↪+ 1)]
    evolution = np.array(evolution)

    # ------------------------------
    # Plot evolution of probabilities
    # ------------------------------
    plt.figure(figsize=(8, 5))
    for i in range(P.shape[0]):
        plt.plot(range(1, max_n + 1), evolution[:, i], marker='o',␣
↪label=f"P(S0→S{i})")

    plt.xlabel("Number of steps (n)")
    plt.ylabel("Probability")
    plt.title("n-step Transition Probabilities from S0 to Each State")
    plt.xticks(range(1, max_n + 1))
    plt.ylim(0, 1)
    plt.grid(True)
    plt.legend()
    plt.tight_layout()
    plt.show()
```

```
1-step transition probability matrix:
[[0.7 0.2 0.1]
 [0.1 0.6 0.3]
 [0.4 0.2 0.4]]

2-step transition probability matrix:
[[0.55 0.28 0.17]
```

```
 [0.25 0.44 0.31]
 [0.46 0.28 0.26]]

5-step transition probability matrix:
[[0.43777 0.32992 0.23231]
 [0.41191 0.34016 0.24793]
 [0.43534 0.32992 0.23474]]

10-step transition probability matrix:
[[0.42867376 0.33329838 0.23802786]
 [0.42837099 0.33340324 0.23822577]
 [0.42866785 0.33329838 0.23803377]]

20-step transition probability matrix:
[[0.42857144 0.33333333 0.23809523]
 [0.42857141 0.33333334 0.23809525]
 [0.42857144 0.33333333 0.23809523]]
```



Practical No 10:- Use any dataset and solve Simple Linear Regression

[28]:
```
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.linear_model import LinearRegression
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error, r2_score
```

```
# Create the dataset
data = pd.DataFrame({
    'YearsExperience': [1.1, 1.3, 1.5, 2.0, 2.2, 2.9, 3.0, 3.2, 3.2, 3.7, 3.9,␣
 ↪4.0, 4.0, 4.1, 4.5, 4.9,
                        5.1, 5.3, 5.9, 6.0, 6.8, 7.1, 7.9, 8.2, 8.7, 9.0, 9.5,␣
 ↪9.6, 10.3, 10.5, 11.0, 11.2],
    'Salary': [39343, 46205, 37731, 43525, 39891, 56642, 60150, 54445, 64445,␣
 ↪57189, 63218, 55794, 56957,
               57081, 61111, 67938, 66029, 83088, 81363, 93940, 91738, 98273,␣
 ↪101302, 113812, 109431,
               105582, 116969, 112635, 122391, 121872, 123000, 124000]
})

# Save to CSV
data.to_csv("Salary_dataset.csv", index=False)

# Prepare features and target
X = data[['YearsExperience']]
y = data['Salary']

# Split into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,␣
 ↪random_state=42)

# Train the model
model = LinearRegression()
model.fit(X_train, y_train)

# Predict on test set
y_pred = model.predict(X_test)

# Evaluate the model
mse = mean_squared_error(y_test, y_pred)
r2 = r2_score(y_test, y_pred)

print("MSE:", mse)
print("R^2 Score:", r2)

# Visualize the results
plt.scatter(X_train, y_train, color='blue', label='Actual')
plt.plot(X_train, model.predict(X_train), color='red', label='Regression Line')
plt.xlabel('Years of Experience')
plt.ylabel('Salary')
plt.title('Salary vs Years of Experience')
plt.legend()
plt.show()
```

```
MSE: 28784943.286985014
R^2 Score: 0.957537154577728
```

Practical No 11:- Use any dataset and solve Multiple Linear Regression

[29]:
```python
# Multiple Linear Regression Example: House Prices

import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error, r2_score

# Step 1: Create Synthetic House Price Dataset
np.random.seed(42)
n = 120  # number of records

# Generate features
size = np.random.randint(500, 4000, n)        # Size in square feet
bedrooms = np.random.randint(1, 6, n)         # Number of bedrooms
```

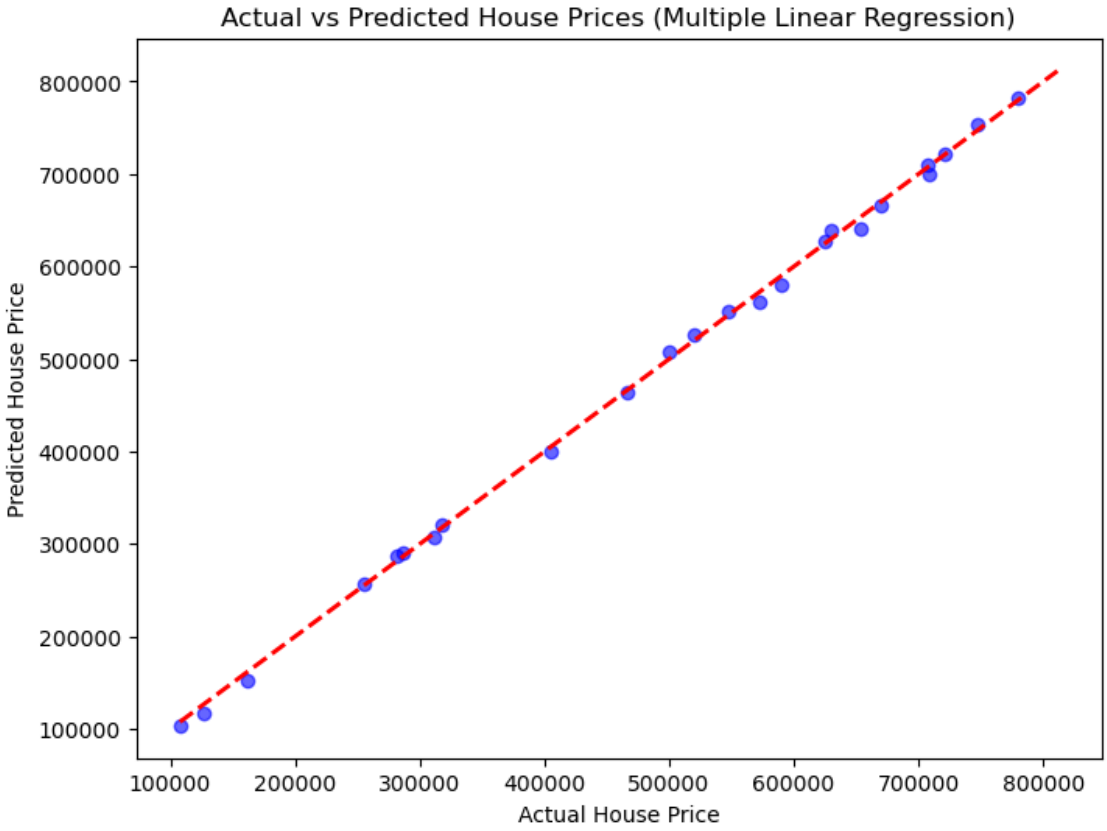
```
bathrooms = np.random.randint(1, 4, n)        # Number of bathrooms
distance = np.random.randint(1, 30, n)        # Distance to city center (in km)

# Create target variable (Price) with some random noise
price = (
    3000*bedrooms +
    5000*bathrooms +
    200*size -
    1500*distance +
    np.random.randint(-10000, 10000, n)  # adds randomness (noise)
)

# Combine into a DataFrame
house_data = pd.DataFrame({
    "Size_sqft": size,
    "Bedrooms": bedrooms,
    "Bathrooms": bathrooms,
    "Distance_km": distance,
    "Price": price
})

# Save dataset to CSV
house_data.to_csv("house_prices.csv", index=False)
print("Dataset saved as house_prices.csv")
print(house_data.head())

# Step 2: Prepare Features and Target
X = house_data[["Size_sqft", "Bedrooms", "Bathrooms", "Distance_km"]]  #␣
 ↪Independent variables
y = house_data["Price"]                                                #␣
 ↪Dependent variable

# Split the dataset into training (80%) and testing (20%)
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=42
)

# Step 3: Train the Model
model = LinearRegression()
model.fit(X_train, y_train)

# Step 4: Make Predictions and Evaluate Model
y_pred = model.predict(X_test)

# Evaluate model performance
mse = mean_squared_error(y_test, y_pred)  # Mean Squared Error
r2 = r2_score(y_test, y_pred)             # R^2 score
```

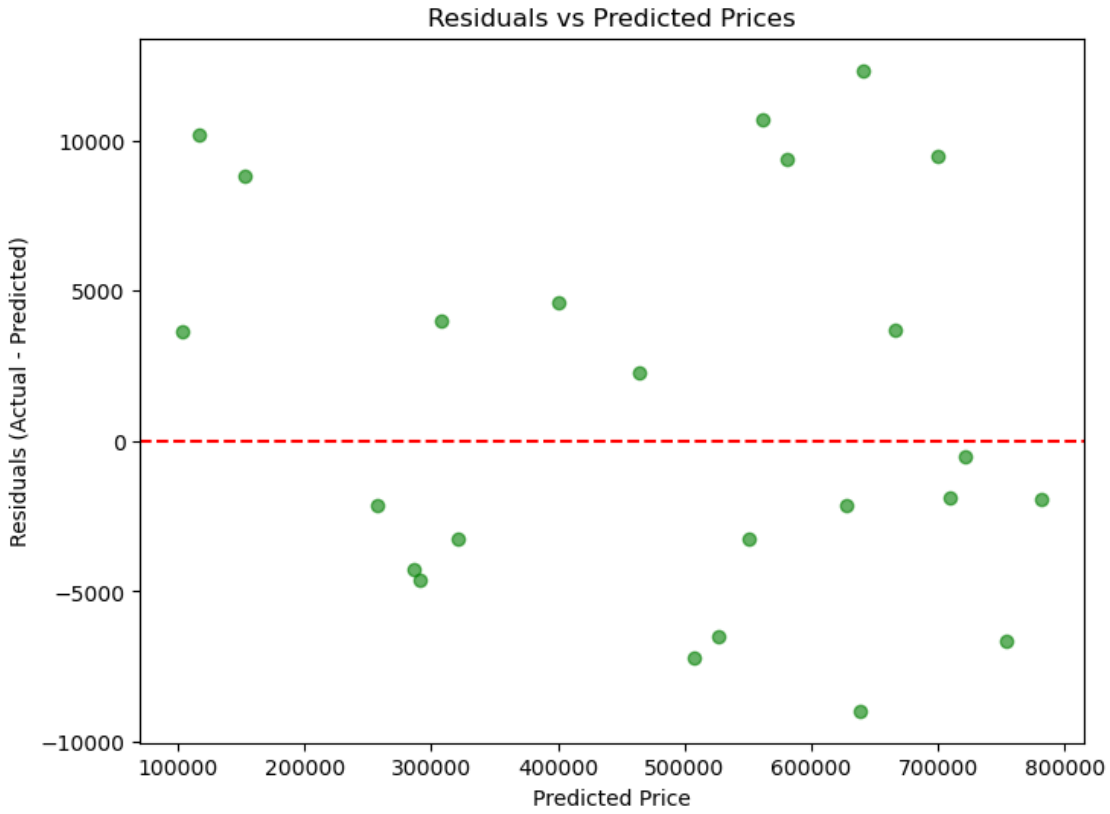
```python
# Print model results
print("\nModel Coefficients (Weights):", model.coef_)
print("Intercept ( ):", model.intercept_)
print("Mean Squared Error (MSE):", mse)
print("R² Score:", r2)

# Display regression equation
print("\nEquation:")
print(f"Price = {model.intercept_:.2f} "
      f"+ ({model.coef_[0]:.2f} * Size_sqft) "
      f"+ ({model.coef_[1]:.2f} * Bedrooms) "
      f"+ ({model.coef_[2]:.2f} * Bathrooms) "
      f"+ ({model.coef_[3]:.2f} * Distance_km)")

# Step 5: Visualization

# Plot actual vs predicted prices
plt.figure(figsize=(8,6))
plt.scatter(y_test, y_pred, color="blue", alpha=0.6)
plt.plot([y.min(), y.max()], [y.min(), y.max()], 'r--', lw=2)
plt.xlabel("Actual House Price")
plt.ylabel("Predicted House Price")
plt.title("Actual vs Predicted House Prices (Multiple Linear Regression)")
plt.show()

# Plot residuals (difference between actual and predicted)
residuals = y_test - y_pred
plt.figure(figsize=(8,6))
plt.scatter(y_pred, residuals, color="green", alpha=0.6)
plt.axhline(0, color="red", linestyle="--")
plt.xlabel("Predicted Price")
plt.ylabel("Residuals (Actual - Predicted)")
plt.title("Residuals vs Predicted Prices")
plt.show()
```

```
Dataset saved as house_prices.csv
   Size_sqft  Bedrooms  Bathrooms  Distance_km   Price
0       3674         3          3           24  720930
1       1360         3          3           18  278531
2       1794         1          2           15  340979
3       1630         3          3           22  310506
4       1595         3          3           23  311784

Model Coefficients (Weights): [  199.63493784  2905.34423713  4464.78588106
-1400.54505448]
Intercept ( ): -513.5268167564063
Mean Squared Error (MSE): 41255013.98912258
```

R^2 Score: 0.9990211626451463

Equation:
Price = -513.53 + (199.63 * Size_sqft) + (2905.34 * Bedrooms) + (4464.79 * Bathrooms) + (-1400.55 * Distance_km)



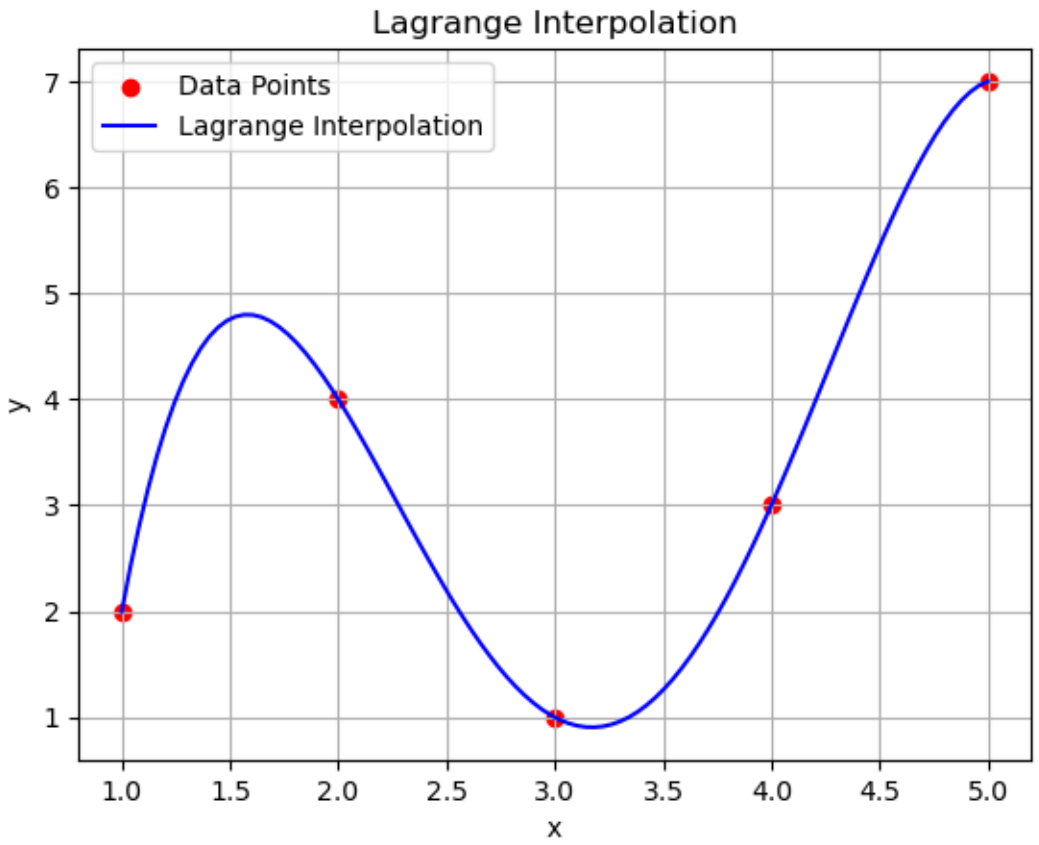


Practical No 12:- Implement Lagrange's Interpolation method

[30]:
```python
# Lagrange Interpolation in Python
import numpy as np
import matplotlib.pyplot as plt

# Sample data points (x, y)
x = np.array([1, 2, 3, 4, 5])        # x-coordinates
y = np.array([2, 4, 1, 3, 7])        # y-coordinates

# Function to perform Lagrange interpolation
def lagrange_interpolation(x_points, y_points, x_val):
    """
    x_points: array of x data points
    y_points: array of y data points
    x_val: the x-value at which interpolation is to be calculated
    Returns: interpolated y-value at x_val
    """
    n = len(x_points)
    result = 0.0
```

```
    # Iterate through each data point
    for i in range(n):
        # Compute the Lagrange basis polynomial L_i(x)
        term = y_points[i]
        for j in range(n):
            if i != j:
                term *= (x_val - x_points[j]) / (x_points[i] - x_points[j])
        result += term  # Add current term to result
    return result

# Interpolate at a new value
x_new = 2.5
y_new = lagrange_interpolation(x, y, x_new)
print(f"Interpolated value at x = {x_new} is y = {y_new}")

# Plotting the data points and interpolation curve
x_range = np.linspace(min(x), max(x), 100)  # Smooth curve
y_range = [lagrange_interpolation(x, y, xi) for xi in x_range]

plt.scatter(x, y, color='red', label='Data Points')  # Original data points
plt.plot(x_range, y_range, color='blue', label='Lagrange Interpolation')  #␣
 ↪Interpolated curve
plt.xlabel('x')
plt.ylabel('y')
plt.title("Lagrange Interpolation")
plt.legend()
plt.grid(True)
plt.show()
```

```
Interpolated value at x = 2.5 is y = 2.1953125
```



Practical No 13:- Implement Newton's Interpolation method

[31]:
```
# Newton's Interpolation Method in Python

# Function to calculate the divided difference table
def divided_diff(x, y):
    """
    Calculates the divided difference table for Newton's Interpolation.
    x: list of x values
    y: list of y values (f(x))
    returns: a 2D list representing the divided difference table
    """
    n = len(y)
    # Create a 2D list initialized with zeros
    table = [[0 for _ in range(n)] for __ in range(n)]

    # First column is y values
    for i in range(n):
        table[i][0] = y[i]
```

```
    # Calculate divided differences
    for j in range(1, n):
        for i in range(n - j):
            table[i][j] = (table[i+1][j-1] - table[i][j-1]) / (x[i+j] - x[i])

    return table

# Function to apply Newton's Interpolation formula
def newton_interpolation(x, y, value):
    """
    Computes the interpolated value at 'value' using Newton's Interpolation.
    x: list of x values
    y: list of y values
    value: the point at which interpolation is required
    returns: interpolated value at x = value
    """
    n = len(x)
    # Get divided difference table
    table = divided_diff(x, y)

    # Start with first y value
    result = table[0][0]

    # Compute interpolation
    product_term = 1.0
    for i in range(1, n):
        product_term *= (value - x[i-1])
        result += table[0][i] * product_term

    return result

# Example usage:
x_points = [1, 2, 3, 4]           # x values
y_points = [1, 4, 9, 16]          # f(x) = x^2

# Interpolate at x = 2.5
value_to_interpolate = 2.5
interpolated_value = newton_interpolation(x_points, y_points,␣
 ↪value_to_interpolate)
print(f"Interpolated value at x = {value_to_interpolate} is␣
 ↪{interpolated_value}")
```

```
Interpolated value at x = 2.5 is 6.25
```