Practical 1

Aim: To implement 2D Geometric Transformations in Python with Practical Exploration of Vectors, Matrices, and Orthogonality.

Part 1: Function to plot 2D vector

Code:

```
import matplotlib.pyplot as plt

def plot_vector(v, color='b', label=None):
    fig, ax = plt.subplots()
    ax.axhline(y=0, color='gray', linestyle='--', linewidth=0.5) ax.axvline(x=0,
    color='gray', linestyle='--', linewidth=0.5)
    ax.quiver(0, 0, v[0], v[1], angles='xy', scale_units='xy', scale=1, color=color,
label=label)

    max_val = max(abs(v[0]), abs(v[1])) + 1
    ax.set_xlim(-max_val, max_val) ax.set_ylim(-
    max_val, max_val)

    ax.set_xlabel('X-axis')
    ax.set_ylabel('Y-axis')
    ax.grid()
    if label:
        ax.legend()

    plt.show()

# Example usage:
plot_vector((3, 4), color='r', label='v')
```

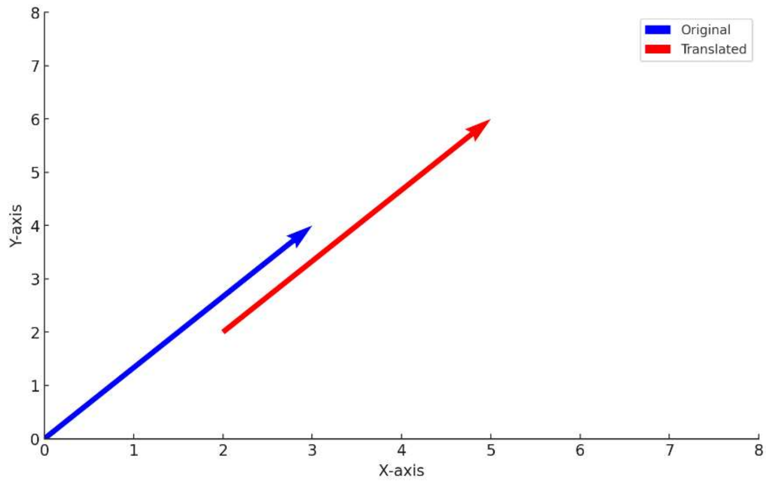
Output:

Part 2: Translation of 2D vector

Code:

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

def plot_translated_vector(vector, tx, ty): x, y = vector
        x_new, y_new = x + tx, y + ty   # Translated vector

        # Plot
        fig, ax = plt.subplots()

        # Axes lines
        ax.axhline(y=0, color='gray', linestyle='--', linewidth=0.5) ax.axvline(x=0,
        color='gray', linestyle='--', linewidth=0.5)

        # Plot original and translated vectors from (0,0) ax.quiver(0, 0, x, y, angles='xy',
        scale_units='xy', scale=1,
color='b', label="Original")
        ax.quiver(0, 0, x_new, y_new, angles='xy', scale_units='xy', scale=1, color='r',
label="Translated")

        # Set plot limits dynamically
        max_val = max(abs(x), abs(y), abs(x_new), abs(y_new)) + 1 ax.set_xlim(-
        max_val, max_val)
        ax.set_ylim(-max_val, max_val)

        ax.set_xlabel('X-axis')
        ax.set_ylabel('Y-axis') ax.grid()
        ax.legend() plt.show()
# Example usage: plot_translated_vector((3, 4), 5, -
2)
```

Output:

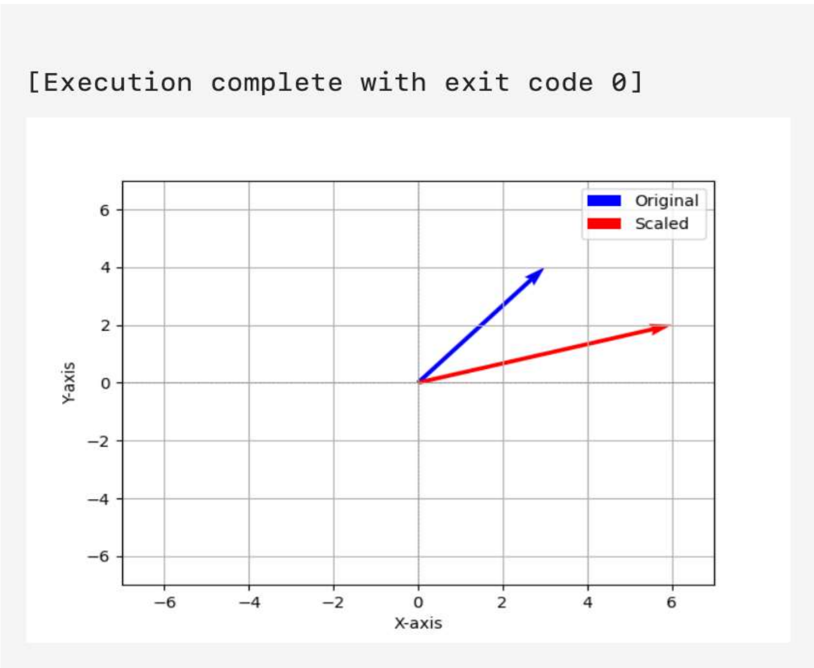


Part 3: Scaling Matrix (by factors sx,sy)

Code:

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

def plot_scaled_vector(vector, sx, sy): x, y = vector
                        S = np.array([[sx, 0, 0],
                                      [0, sy, 0],
                                      [0, 0, 1]])


        P = np.array([x, y, 1]) P_new =
        np.dot(S, P)
        x_new, y_new = P_new[0], P_new[1] fig, ax =
        plt.subplots()
        ax.axhline(y=0, color='gray', linestyle='--', linewidth=0.5) ax.axvline(x=0,
        color='gray', linestyle='--', linewidth=0.5) ax.quiver(0, 0, x, y, angles='xy',
        scale_units='xy', scale=1,
        # Set plot limits dynamically
        max_val = max(abs(x), abs(y), abs(x_new), abs(y_new)) + 1 ax.set_xlim(-
        max_val, max_val)
        ax.set_ylim(-max_val, max_val)
        ax.set_xlabel('X-axis') ax.set_ylabel('Y-
        axis') ax.grid()
        ax.legend() plt.show()

# Example usage: plot_scaled_vector((3, 4),
2, 0.5)
```

Output:

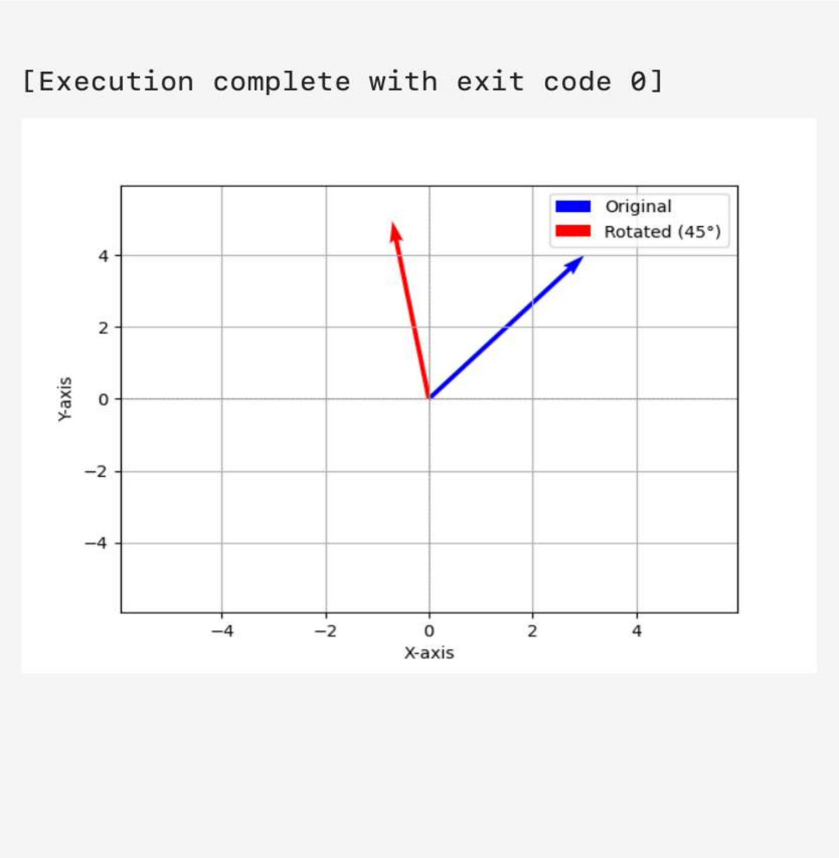


Part 4: Rotation Matrix (by angle theta)
Code:

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

def rotate_vector(original_vector, rotation_angle): """Rotates a 2D
    vector counterclockwise.
    Args:
        original_vector: Tuple (x, y) representing the original
        rotation_angle: Rotation angle in degrees.
    if not isinstance(original_vector, tuple) or len(original_vector) != 2:
        raise ValueError("original_vector must be a tuple of length
2 (x, y)")
    if not isinstance(rotation_angle, (int, float)):
        raise ValueError("rotation_angle must be a number (int or float)")

    theta_rad = np.radians(rotation_angle)
    R = np.array([[np.cos(theta_rad), -np.sin(theta_rad), 0], [np.sin(theta_rad), np.cos(theta_rad), 0],
                  [0, 0, 1]])
    P = np.array([original_vector[0], original_vector[1], 1]) P_new = np.dot(R, P)
    x_new, y_new = P_new[0], P_new[1] return
    (x_new, y_new)



def plot_vectors(original_vector, rotated_vector, rotation_angle): """Plots the original and rotated
    vectors."""
    fig, ax = plt.subplots()
    ax.axhline(y=0, color='gray', linestyle='--', linewidth=0.5) ax.axvline(x=0,
    color='gray', linestyle='--', linewidth=0.5)

    ax.quiver(0, 0, original_vector[0], original_vector[1], angles='xy', scale_units='xy', scale=1,
color='b', label="Original")


  ax.quiver(0, 0, rotated_vector[0], rotated_vector[1], angles='xy', scale_units='xy', scale=1,
color='r', label=f"Rotated ({rotation_angle}°)")

    max_val = max(abs(original_vector[0]), abs(original_vector[1]),
```

```
abs(rotated_vector[0]), abs(rotated_vector[1])) + 1 ax.set_xlim(-
        max_val, max_val)
        ax.set_ylim(-max_val, max_val)

        ax.set_xlabel('X-axis')
        ax.set_ylabel('Y-axis') ax.grid()
        ax.legend() plt.show()


# Example usage:
original = (3, 4)
angle = 45
rotated = rotate_vector(original, angle)
plot_vectors(original, rotated, angle)
```

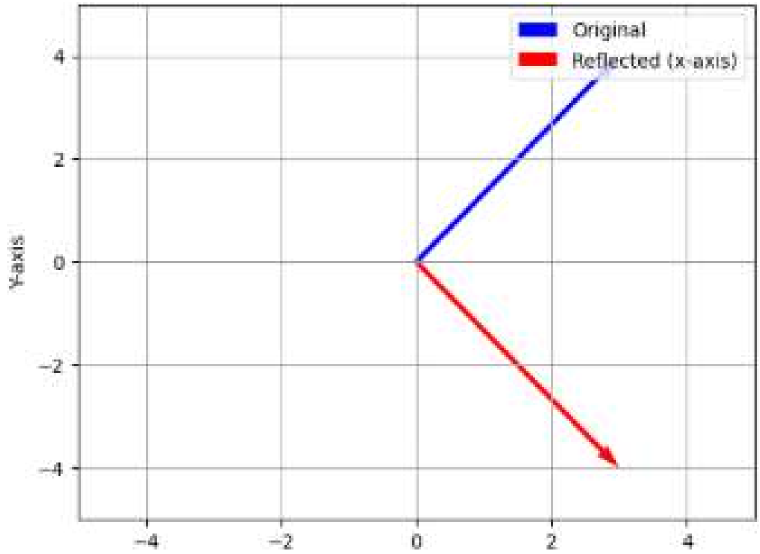
Output:



Part 5: Reflection Matrix (about x-axis)
Code:

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

def reflect_vector_x(original_vector): reflection_matrix =
        np.array([[1, 0],
                                                   [0, -1]])
        reflected_vector = np.dot(reflection_matrix, original_vector) return reflected_vector

def          plot_vectors(original_vector,          transformed_vector,
transformation_name):
        fig, ax = plt.subplots()
        ax.axhline(y=0,  color='gray',  linestyle='--',  linewidth=0.5)  ax.axvline(x=0,
        color='gray',  linestyle='--',  linewidth=0.5)  ax.quiver(0,  0,  original_vector[0],
        original_vector[1],
angles='xy',  scale_units='xy',  scale=1,  color='b',  label="Original")  ax.quiver(0,  0,
        transformed_vector[0], transformed_vector[1],
angles='xy',       scale_units='xy',       scale=1,       color='r',
label=transformation_name)

        max_val = max(abs(original_vector[0]), abs(original_vector[1]), abs(transformed_vector[0]),
abs(transformed_vector[1])) + 1
        ax.set_xlim(-max_val, max_val)
        ax.set_ylim(-max_val, max_val)

        ax.set_xlabel('X-axis')
        ax.set_ylabel('Y-axis') ax.grid()
        ax.legend() plt.show()

# Example usage (Reflection):
original = np.array([3, 4])  # Reset original vector reflected =
reflect_vector_x(original plot_vectors(original, reflected, "Reflected (x-axis)")
```

Output:

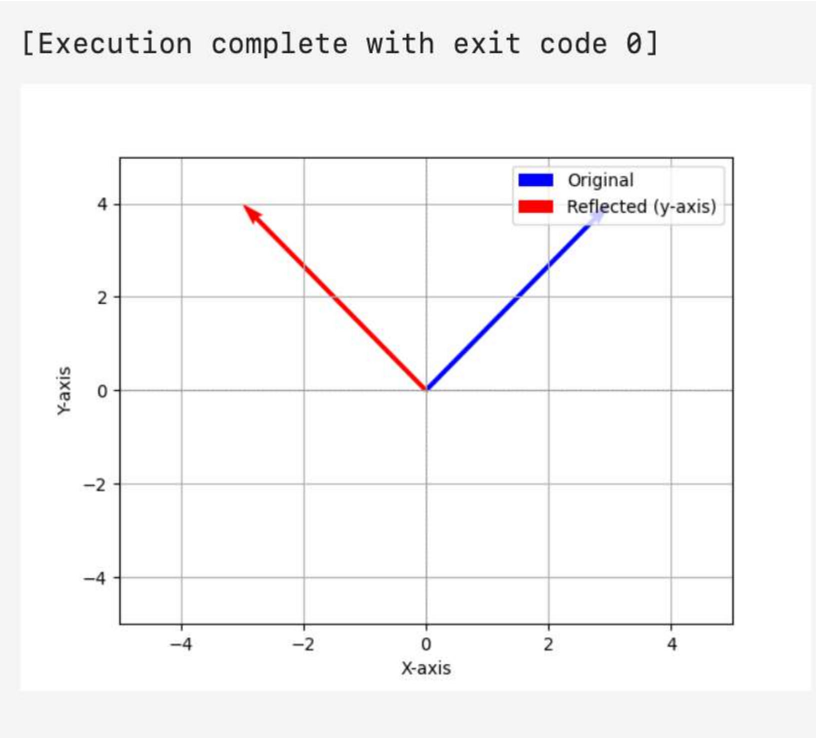


Part 6: Reflection Matrix (about y-axis)

Code:

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

def reflect_vector_y(original_vector): """Reflects a 2D vector about
        the y-axis.""" reflection_matrix = np.array([[-1, 0],
                                                   [ 0, 1]])
        reflected_vector = np.dot(reflection_matrix, original_vector) return reflected_vector

def plot_vectors(original_vector, transformed_vector, transformation_name):
        """Plots the original and transformed vectors.""" fig, ax = plt.subplots()
        ax.axhline(y=0, color='gray', linestyle='--', linewidth=0.5) ax.axvline(x=0,
        color='gray', linestyle='--', linewidth=0.5)

        ax.quiver(0, 0, original_vector[0], original_vector[1], angles='xy', scale_units='xy', scale=1,
color='b', label="Original")
        ax.quiver(0, 0, transformed_vector[0], transformed_vector[1], angles='xy',
scale_units='xy', scale=1, color='r', label=transformation_name)

        max_val = max(abs(original_vector[0]), abs(original_vector[1]), abs(transformed_vector[0]),
abs(transformed_vector[1])) + 1
        ax.set_xlim(-max_val, max_val)
        ax.set_ylim(-max_val, max_val)

        ax.set_xlabel('X-axis')
        ax.set_ylabel('Y-axis') ax.grid()
        ax.legend() plt.show()

# Example usage (Reflection about y-axis): original =
np.array([3, 4])
reflected_y = reflect_vector_y(original) plot_vectors(original, reflected_y,
"Reflected (y-axis)")
```

Output:



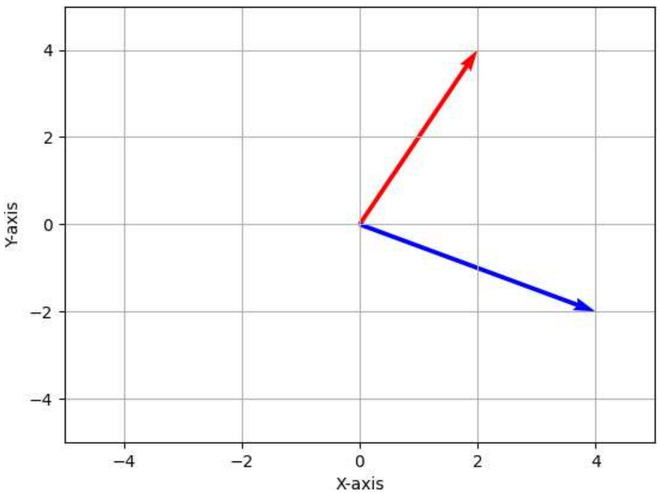
Part 7: Check orthogonality of 2 vector

Code:

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

def plot_vectors(vectors, colors): fig, ax =
        plt.subplots()
        for v, color in zip(vectors, colors):
                ax.quiver(0, 0, v[0], v[1], angles='xy', scale_units='xy', scale=1, color=color)
        ax.set_xlim(-5, 5)
        ax.set_ylim(-5, 5)
        ax.set_xlabel('X-axis')
        ax.set_ylabel('Y-axis') ax.grid()
        plt.show()

def is_orthogonal(v1, v2): return np.dot(v1,
        v2) == 0

vector1 = np.array([2, 4]) vector2 =
np.array([4, -2])
plot_vectors([vector1, vector2], ['r', 'b'])

print(f"Are vectors {vector1} and {vector2} orthogonal?
->{is_orthogonal(vector1, vector2)}")
```

Output:

Are vectors [2 4] and [4 -2] orthogonal? -> True



Practical 2

Aim : Solving linear systems and Matrix Decompositions in Python.

Code:

```
import numpy as np
from scipy.linalg import lu, cholesky


# Define the coefficient matrix A
A = np.array([
    [3, 2, 1],
    [2, 3, 2],
    [1, 1, 3]
])

# Define the constants vector B
B = np.array([1, 2, 3])

# Solve the linear system
X = np.linalg.solve(A, B)

print("Solution:", X)
# Function to perform LU Decomposition def lu_decomposition(A):
P, L, U = lu(A)
return L, U

# Function to perform Cholesky Decomposition def cholesky_decomposition(A):
L = cholesky(A, lower=True) return L

# Function to perform QR Decomposition def qr_decomposition(A):
Q, R = np.linalg.qr(A) return Q, R

# Function to solve linear systems using LU decomposition def solve_lu(A, b):
L, U = lu_decomposition(A) # Solve Ly = b
y = np.linalg.solve(L, b) # Solve Ux = y
x = np.linalg.solve(U, y) return x

# Function to solve linear systems using Cholesky decomposition def
solve_cholesky(A, b):
L = cholesky_decomposition(A) # Solve Ly = b
y = np.linalg.solve(L, b) # Solve L^Tx = y
x = np.linalg.solve(L.T, y) return x

# Function to solve least squares problems using QR decomposition def solve_qr(A,
b):
Q, R = qr_decomposition(A) # Solve Rx = Q^Tb
x = np.linalg.solve(R, Q.T @ b) return x
```

```
# Example matrices and vectors
A_lu = np.array([[4, 3], [6, 3]])  # For LU Decomposition

b_lu = np.array([10, 12])      # Right-hand side for LU

A_chol = np.array([[4, 2], [2, 3]])  # For Cholesky Decomposition b_chol = np.array([8,
6])       # Right-hand side for Cholesky

A_qr = np.array([[1, 1], [1, 2], [1, 3]])  # For QR Decomposition b_qr = np.array([1, 2,
3])       # Right-hand side for QR

# Solve using LU Decomposition x_lu = solve_lu(A_lu, b_lu)
print("Solution using LU Decomposition:") print(x_lu)

# Solve using Cholesky Decomposition x_chol = solve_cholesky(A_chol, b_chol)
print("\nSolution using Cholesky Decomposition:") print(x_chol)

# Solve using QR Decomposition x_qr = solve_qr(A_qr, b_qr)
print("\nLeast Squares Solution using QR Decomposition:")
print(x_qr)
```

output:

```
Solution: [0. 0. 1.]

Solution using LU Decomposition:
[-1.          5.33333333]

Solution using Cholesky Decomposition:
[1.5 1. ]

Least Squares Solution using QR Decomposition:
[2.56395025e-16 1.00000000e+00]

[Execution complete with exit code 0]
```

Practical 3

AIM :- Understanding sensitivity analysis for linear systems by implementing condition numbers, matrix norms and sensitivity analysis for effect of perturbations in python.

Code :-

```
import numpy as np
import matplotlib.pyplot as plt 3.
# Define the linear system Ax = b
A = np.array([[3, 2], [1, 4]])
b = np.array([5, 6])

# Function to compute the condition number
def condition_number(matrix):
return np.linalg.cond(matrix)

# Function to perform sensitivity analysis
def sensitivity_analysis(A, b, perturbation_range):
condition_num = condition_number(A)
perturbations = np.linspace(-perturbation_range,
perturbation_range, 100)
delta_x = []
for delta in perturbations:
# Perturb b
b_perturbed = b + delta
# Solve the perturbed system
x_perturbed = np.linalg.solve(A, b_perturbed)
# Calculate the change in solution
delta_x.append(np.linalg.norm(x_perturbed -
np.linalg.solve(A, b)))
return perturbations, delta_x, condition_num
# Perform sensitivity analysis

perturbation_range = 1.0
perturbations, delta_x, condition_num =
sensitivity_analysis(A, b, perturbation_range)
# Plotting the results
plt.figure(figsize=(10, 6))
plt.plot(perturbations, delta_x, label='Change in
Solution ||Δx||', color='blue')
plt.axhline(y=condition_num, color='red',
linestyle='--', label='Condition Number')
plt.title('Sensitivity Analysis of Linear System')
```

```
plt.xlabel('Perturbation in b')
plt.ylabel('Change in Solution ||Δx||')
plt.legend()
plt.grid()
plt.show()
```

Output :-