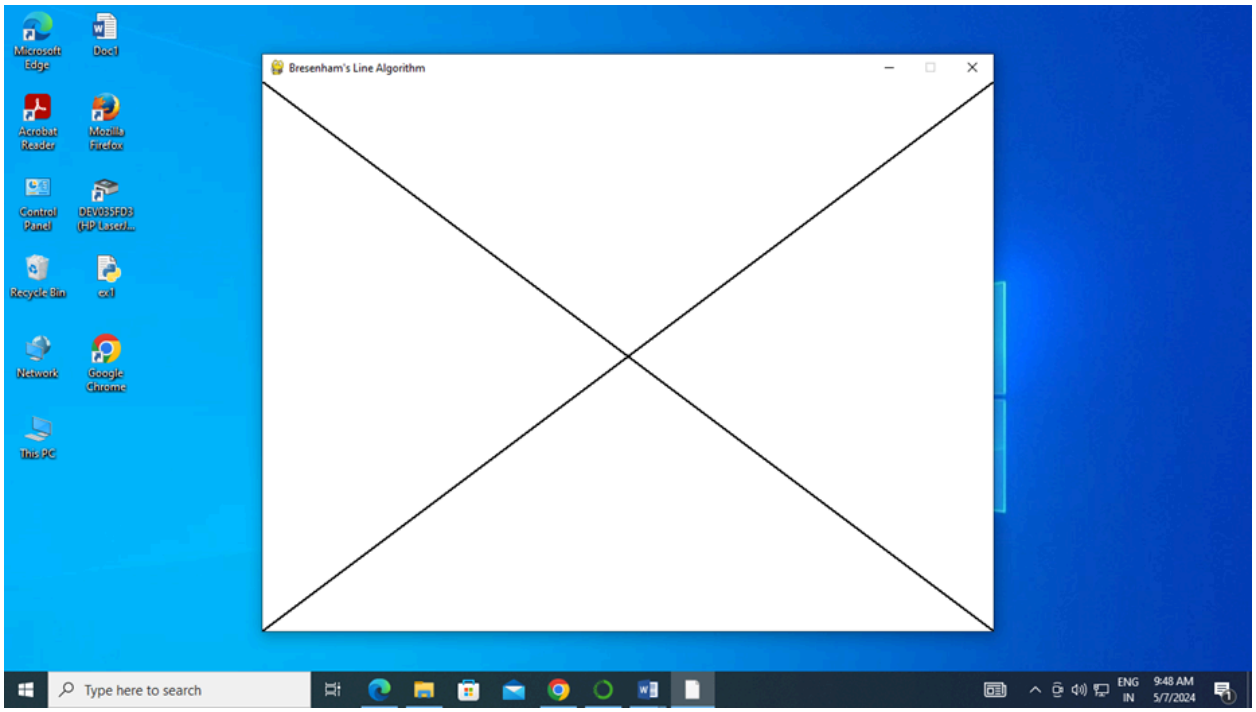
PART A

List of problems for which student should develop program and execute in the Laboratory using openGL/openCV/ Python

1. Develop a program to draw a line using Bresenham's line drawing technique

OUTPUT



2. Develop a program to demonstrate basic geometric operations on the 2D object

```
import matplotlib.pyplot as plt

import numpy as np
```

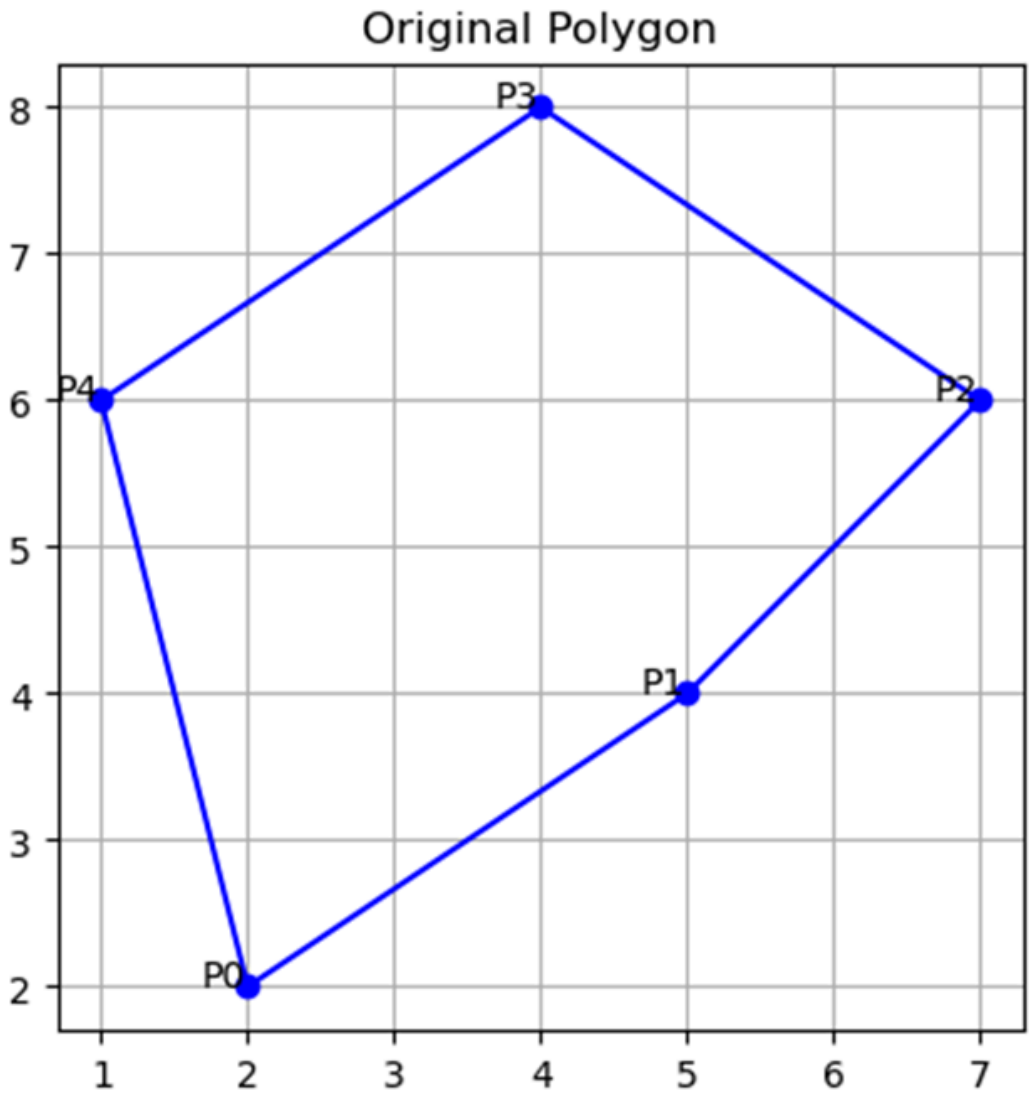
```
# Function to plot a polygon
def plot_polygon(polygon, title):
    plt.figure()
    plt.title(title)
    plt.gca().set_aspect('equal', adjustable='box')
    plt.grid(True)

    # Plot the polygon
    polygon = np.concatenate((polygon, [polygon[0]]))  # Close the polygon
    plt.plot(polygon[:, 0], polygon[:, 1], 'b-')

    # Plot vertices
    plt.plot(polygon[:-1, 0], polygon[:-1, 1], 'bo')
    for i, (x, y) in enumerate(polygon[:-1]):
    plt.text(x, y, f'P{i}', ha='right')

    plt.show()

# Function to translate a polygon
def translate_polygon(polygon):
    tx = float(input("Enter translation along x-axis: "))
    ty = float(input("Enter translation along y-axis: "))
    translated_polygon = polygon + np.array([tx, ty])
    return translated_polygon
```

```
# Function to rotate a polygon
def rotate_polygon(polygon):
    angle_degrees = float(input("Enter rotation angle in degrees: "))
    angle_radians = np.radians(angle_degrees)
    rotation_matrix = np.array([[np.cos(angle_radians), -np.sin(angle_radians)],
                                [np.sin(angle_radians), np.cos(angle_radians)]])
    rotated_polygon = np.dot(polygon, rotation_matrix)
    return rotated_polygon

# Function to scale a polygon
def scale_polygon(polygon):
    sx = float(input("Enter scaling factor along x-axis: "))
    sy = float(input("Enter scaling factor along y-axis: "))
    scaled_polygon = polygon * np.array([sx, sy])
    return scaled_polygon

# Function to reflect a polygon over x-axis
def reflect_polygon_x_axis(polygon):
    reflected_polygon = np.array(polygon)
    reflected_polygon[:, 1] = -reflected_polygon[:, 1]
    return reflected_polygon

# Main function to demonstrate geometric operations
```

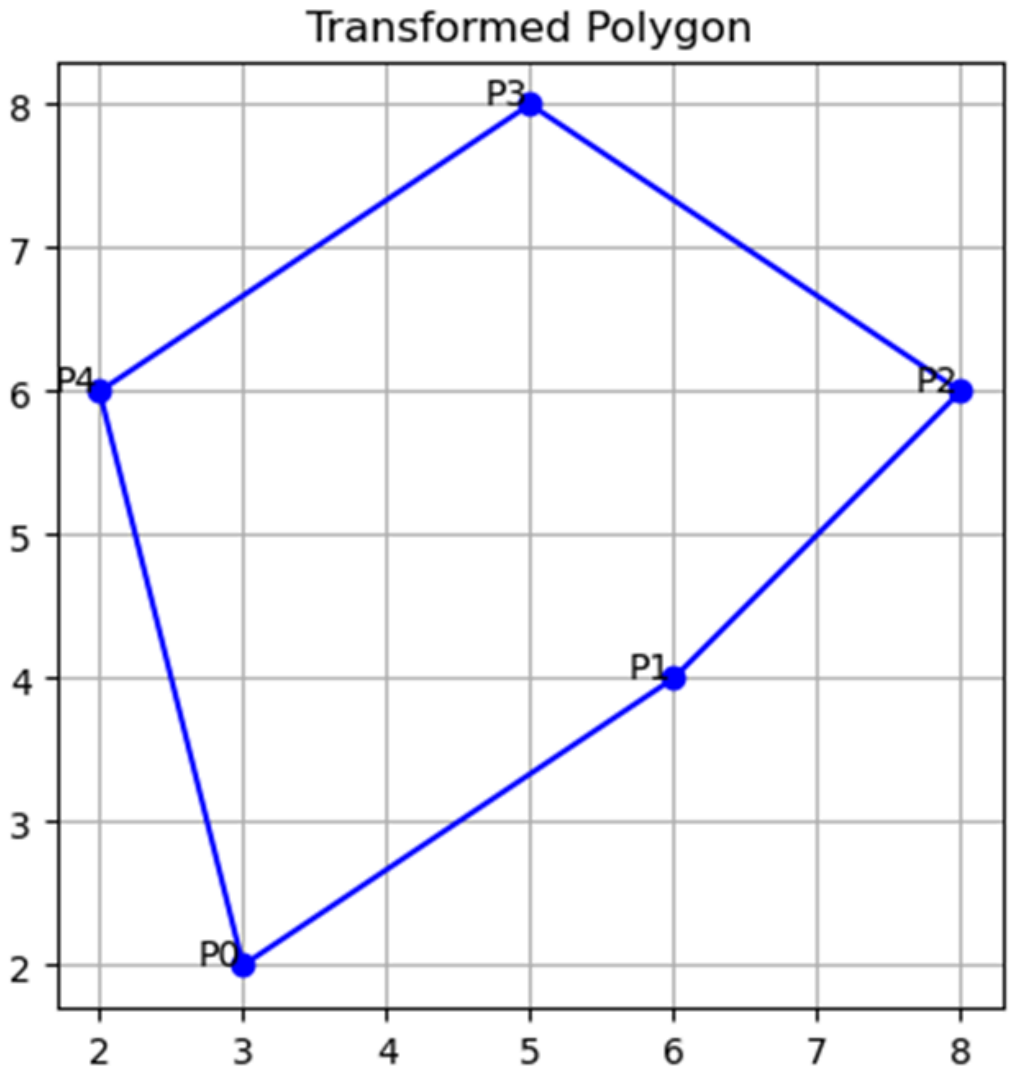
```
def main():
        # Define a simple polygon as a list of vertices (x, y)
        polygon = np.array([[2, 2], [5, 4], [7, 6], [4, 8], [1, 6]])

        # Plot the original polygon
        plot_polygon(polygon, 'Original Polygon')

        # Prompt user for the geometric operation
        while True:
        print("\nChoose a geometric operation:")
        print("1. Translate the polygon")
        print("2. Rotate the polygon")
        print("3. Scale the polygon")
        print("4. Reflect the polygon over x-axis")
        print("5. Exit")

        choice = input("Enter your choice (1-5): ")

        if choice == '1':
        polygon = translate_polygon(polygon)
        elif choice == '2':
        polygon = rotate_polygon(polygon)
        elif choice == '3':
        polygon = scale_polygon(polygon)
```

```
        elif choice == '4':
            polygon = reflect_polygon_x_axis(polygon)
        elif choice == '5':
            break
        else:
            print("Invalid choice. Please enter a valid option.")

    # Plot the transformed polygon
        plot_polygon(polygon, 'Transformed Polygon')

        print("Exiting...")

if __name__ == "__main__":
        main()
```

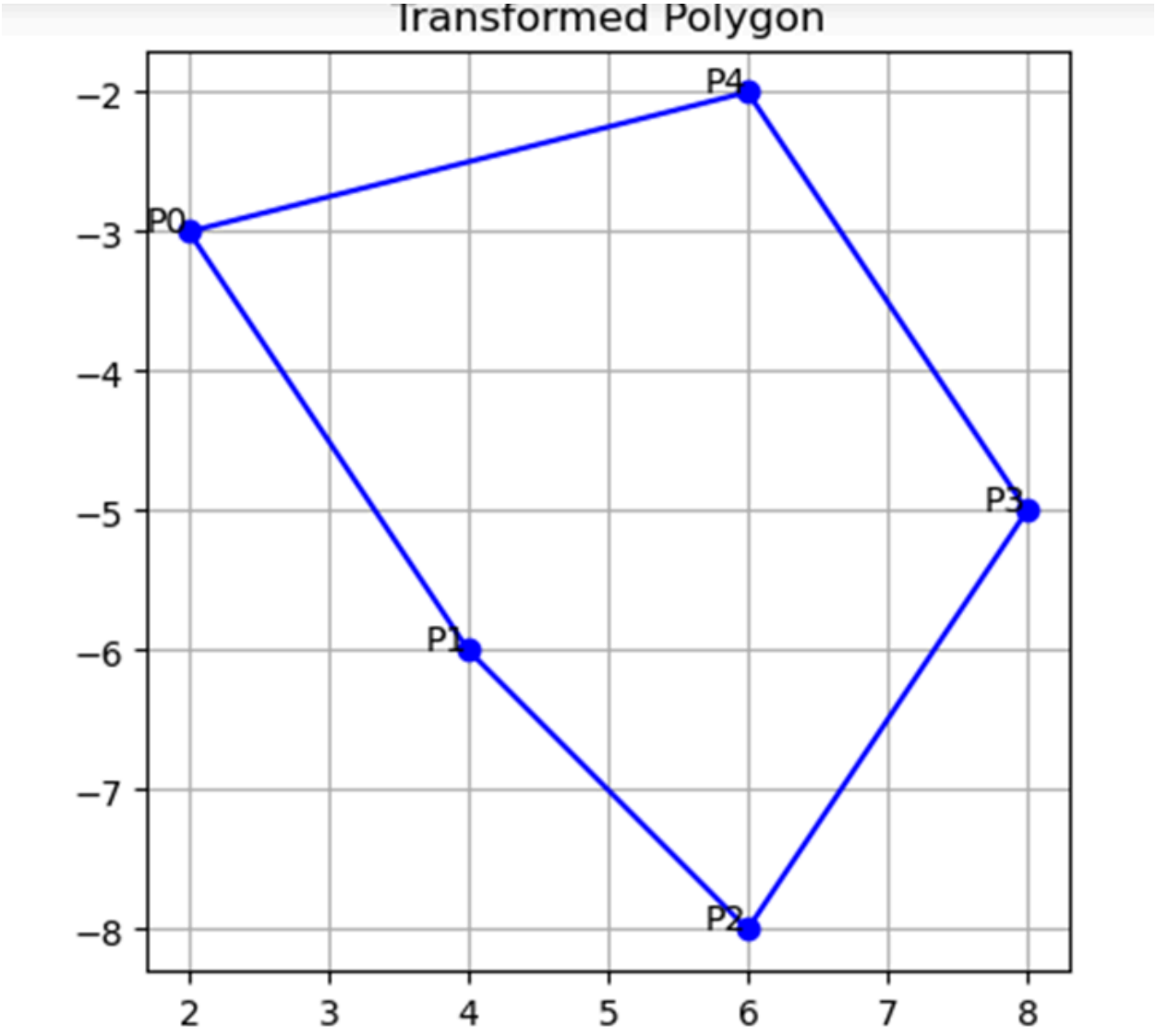
Output :


Original Polygon
P0
P1
P2
P3
P4
1
2
3
4
5
6
7
2
3
4
5
6
7
8

```
Choose a geometric operation:
1. Translate the polygon
2. Rotate the polygon
3. Scale the polygon
4. Reflect the polygon over x-axis
5. Exit
Enter your choice (1-5): 1
Enter translation along x-axis: 1
Enter translation along y-axis: 0
```

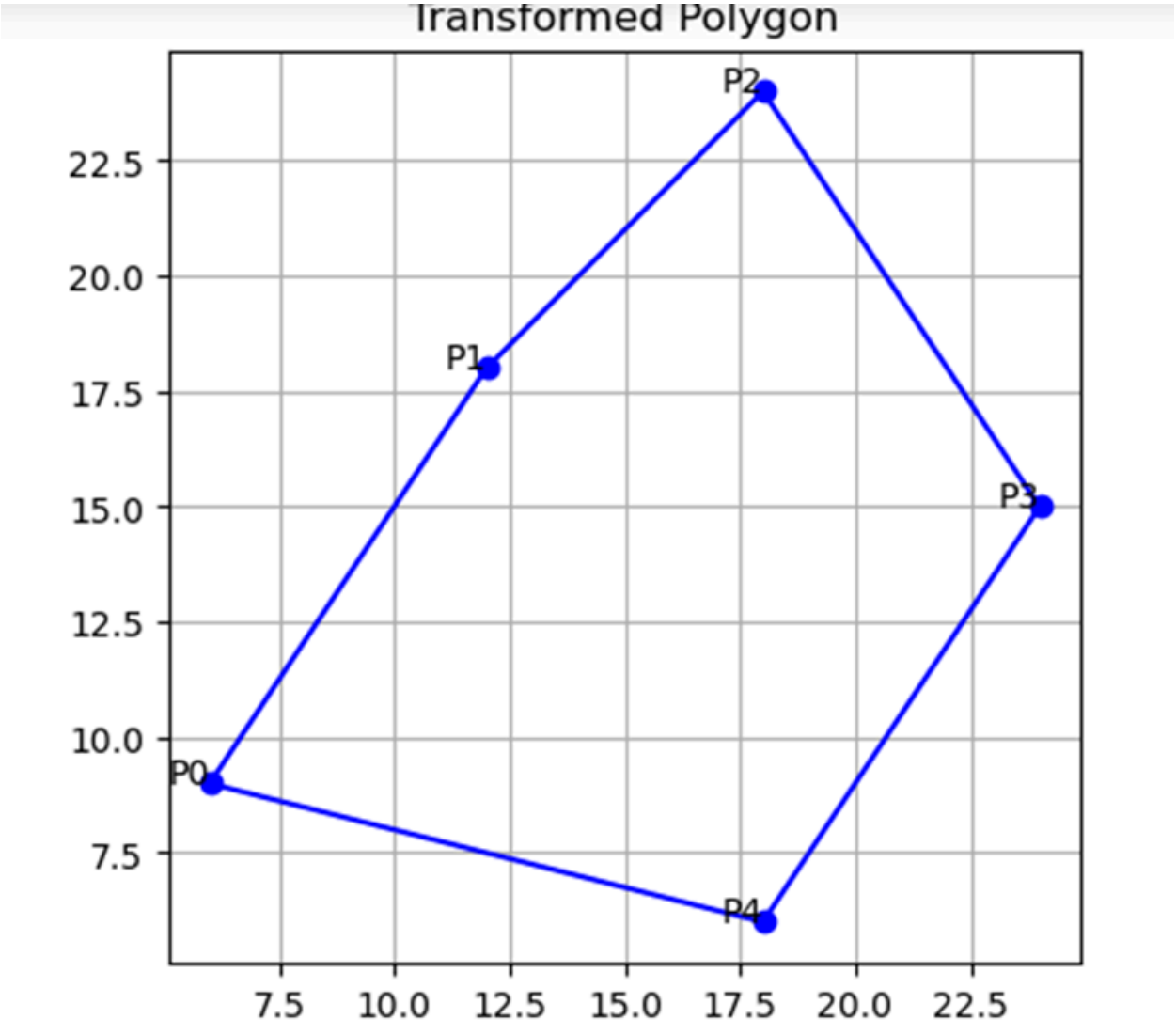



```
Choose a geometric operation:
1. Translate the polygon
2. Rotate the polygon
3. Scale the polygon
4. Reflect the polygon over x-axis
5. Exit
Enter your choice (1-5): 2
Enter rotation angle in degrees: 90
```


Transformed Polygon
P0
P1
P2
P3
P4
−2
−3
−4
−5
−6
−7
−8
2
3
4
5
6
7
8

```
Choose a geometric operation:
1. Translate the polygon
2. Rotate the polygon
3. Scale the polygon
4. Reflect the polygon over x-axis
5. Exit
Enter your choice (1-5): 3
Enter scaling factor along x-axis: 3
Enter scaling factor along y-axis: 3
```

```
Choose a geometric operation:
1. Translate the polygon
2. Rotate the polygon
3. Scale the polygon
4. Reflect the polygon over x-axis
5. Exit
Enter your choice (1-5): 4
```



```
Choose a geometric operation:
1. Translate the polygon
2. Rotate the polygon
3. Scale the polygon
4. Reflect the polygon over x-axis
5. Exit
Enter your choice (1-5): 5
Exiting...
```

3. Develop a program to demonstrate basic geometric operations on the 3D object

```
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D


# Function to plot a 3D object
def plot_3d_object(vertices, title):
    fig = plt.figure()
    ax = fig.add_subplot(111, projection='3d')
    ax.set_title(title)

    # Plot vertices
    ax.scatter(vertices[:, 0], vertices[:, 1], vertices[:, 2], color='b')

    # Connect vertices to form edges
    edges = [[0, 1], [1, 2], [2, 3], [3, 0], [0, 4], [1, 4], [2, 4], [3, 4]]
```

```
        for edge in edges:
        ax.plot(vertices[edge, 0], vertices[edge, 1], vertices[edge, 2], color='r')

        ax.set_xlabel('X')
        ax.set_ylabel('Y')
        ax.set_zlabel('Z')
        ax.set_aspect('auto')

        plt.show()

# Function to translate a 3D object
def translate_3d_object(vertices):
        tx = float(input("Enter translation along x-axis: "))
        ty = float(input("Enter translation along y-axis: "))
        tz = float(input("Enter translation along z-axis: "))
        translated_vertices = vertices + np.array([tx, ty, tz])
        return translated_vertices

# Function to rotate a 3D object around x-axis
def rotate_x_3d_object(vertices):
        angle_degrees = float(input("Enter rotation angle around x-axis in degrees: "))
        angle_radians = np.radians(angle_degrees)
        rotation_matrix = np.array([[1, 0, 0],
                                    [0, np.cos(angle_radians), -np.sin(angle_radians)],
```

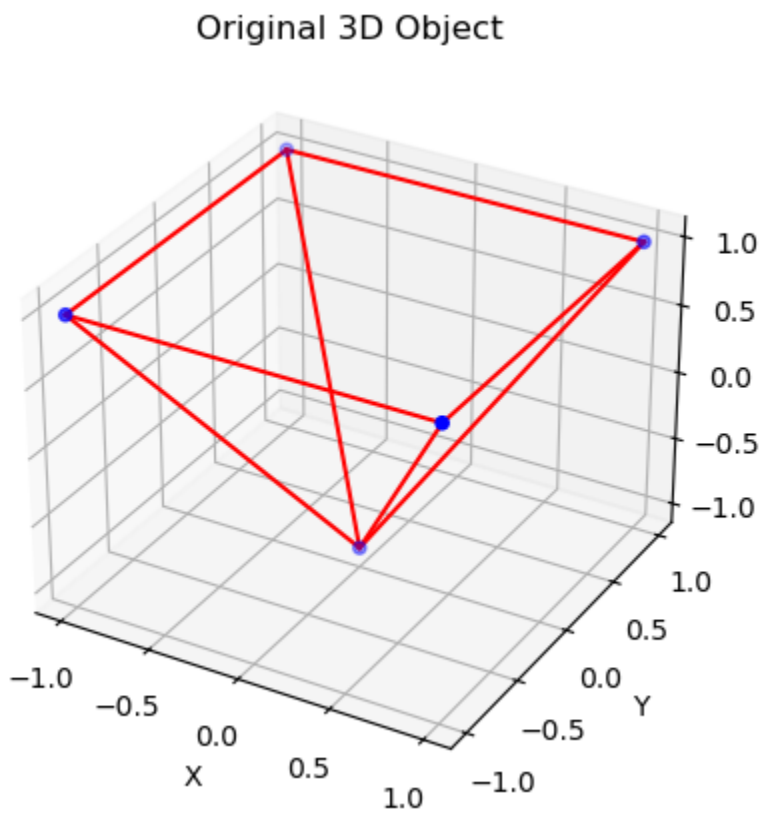
```
                        [0, np.sin(angle_radians), np.cos(angle_radians)]])
    rotated_vertices = np.dot(vertices, rotation_matrix)
    return rotated_vertices

# Function to rotate a 3D object around y-axis
def rotate_y_3d_object(vertices):
    angle_degrees = float(input("Enter rotation angle around y-axis in degrees: "))
    angle_radians = np.radians(angle_degrees)
  rotation_matrix = np.array([[np.cos(angle_radians), 0, np.sin(angle_radians)],
                        [0, 1, 0],
                       [-np.sin(angle_radians), 0, np.cos(angle_radians)]])
    rotated_vertices = np.dot(vertices, rotation_matrix)
    return rotated_vertices

# Function to rotate a 3D object around z-axis
def rotate_z_3d_object(vertices):
    angle_degrees = float(input("Enter rotation angle around z-axis in degrees: "))
    angle_radians = np.radians(angle_degrees)
    rotation_matrix = np.array([[np.cos(angle_radians), -np.sin(angle_radians), 0],
                       [np.sin(angle_radians), np.cos(angle_radians), 0],
                        [0, 0, 1]])
    rotated_vertices = np.dot(vertices, rotation_matrix)
    return rotated_vertices
```

```
# Function to scale a 3D object
def scale_3d_object(vertices):
    sx = float(input("Enter scaling factor along x-axis: "))
    sy = float(input("Enter scaling factor along y-axis: "))
    sz = float(input("Enter scaling factor along z-axis: "))
    scaled_vertices = vertices * np.array([sx, sy, sz])
    return scaled_vertices

# Main function to demonstrate geometric operations on a 3D object
def main():
    # Define a simple 3D object as a numpy array of vertices
    vertices = np.array([[1, 1, 1], [1, -1, 1], [-1, -1, 1], [-1, 1, 1], [0, 0, -1]])

    # Plot the original 3D object
    plot_3d_object(vertices, 'Original 3D Object')

    # Prompt user for the geometric operation
    while True:
    print("\nChoose a geometric operation:")
    print("1. Translate the 3D object")
    print("2. Rotate the 3D object around x-axis")
    print("3. Rotate the 3D object around y-axis")
    print("4. Rotate the 3D object around z-axis")
    print("5. Scale the 3D object")
```

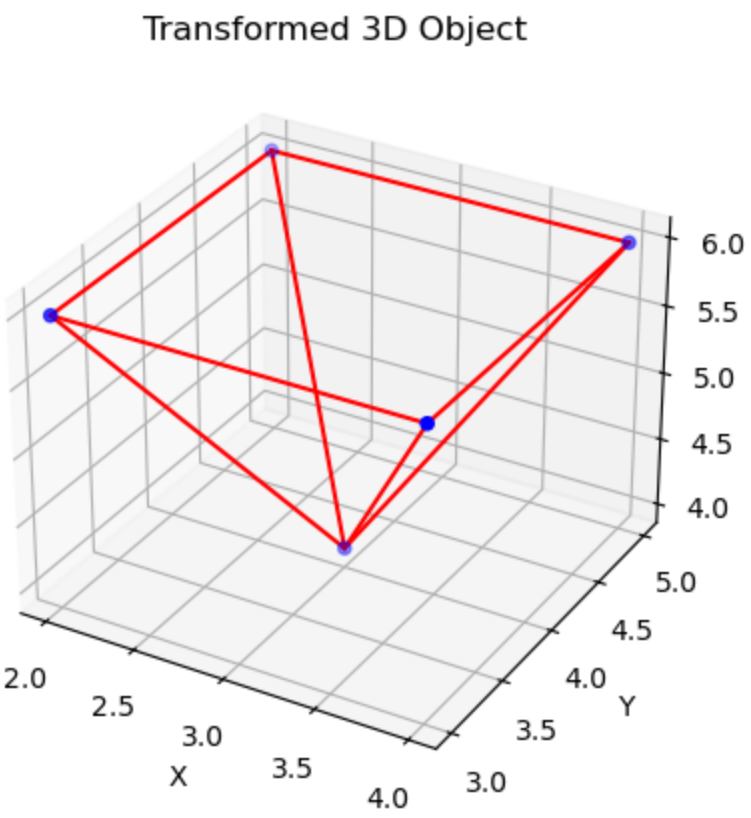
```
print("6. Exit")

choice = input("Enter your choice (1-6): ")

if choice == '1':
vertices = translate_3d_object(vertices)
elif choice == '2':
    vertices = rotate_x_3d_object(vertices)
elif choice == '3':
vertices = rotate_y_3d_object(vertices)
elif choice == '4':
vertices = rotate_z_3d_object(vertices)
elif choice == '5':
vertices = scale_3d_object(vertices)
elif choice == '6':
break
else:
print("Invalid choice. Please enter a valid option.")

# Plot the transformed 3D object
plot_3d_object(vertices, 'Transformed 3D Object')

print("Exiting...")
```

```
if __name__ == "__main__":
        main()
```

OUTPUT

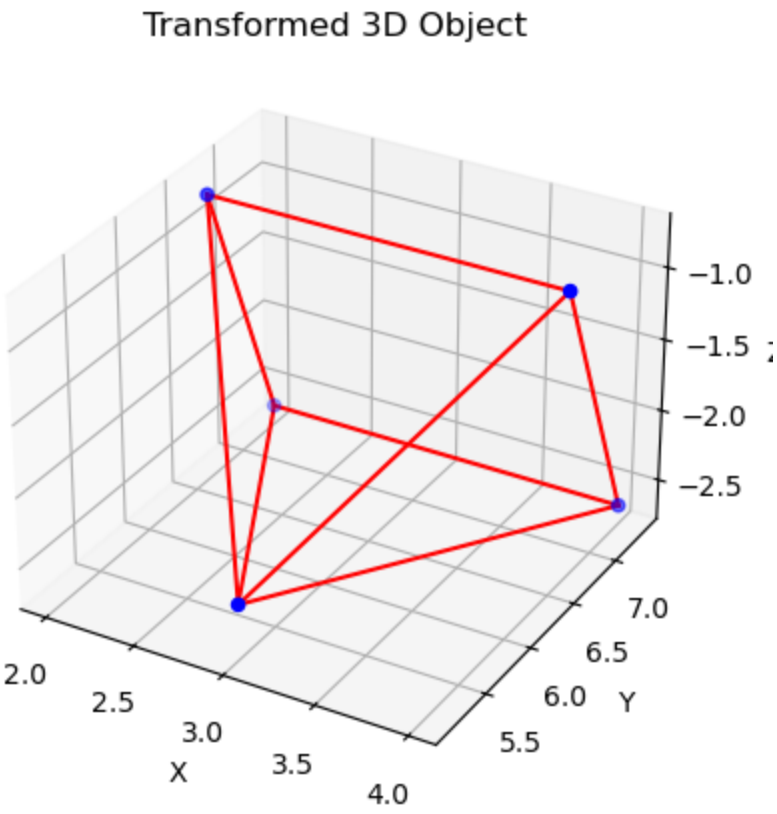



```
Choose a geometric operation:
1. Translate the 3D object
2. Rotate the 3D object around x-axis
3. Rotate the 3D object around y-axis
4. Rotate the 3D object around z-axis
5. Scale the 3D object
6. Exit

Enter your choice (1-6): 1
Enter translation along x-axis: 3
Enter translation along y-axis: 4
Enter translation along z-axis: 5
```



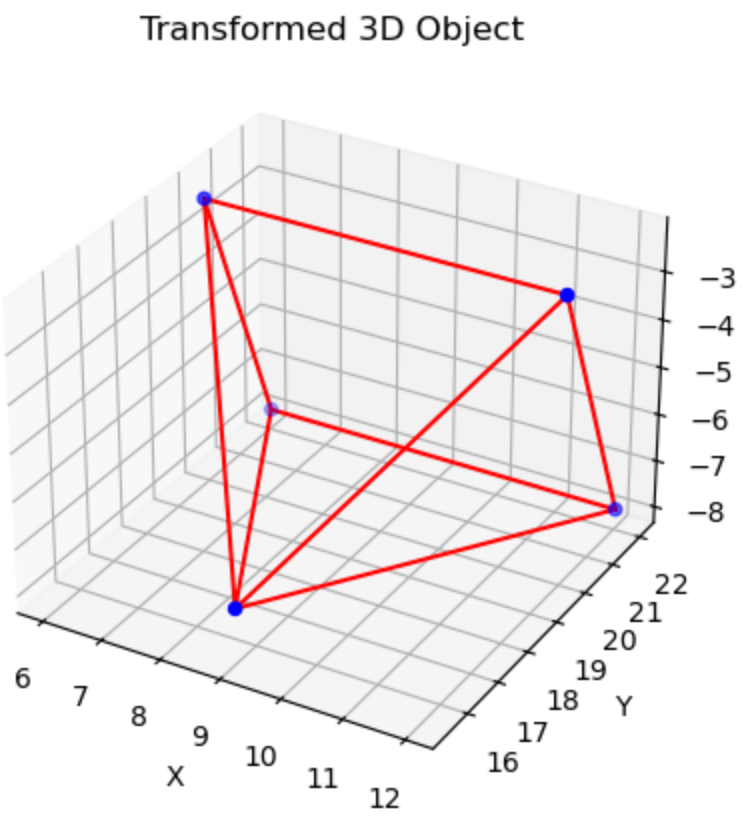
Choose a geometric operation:
1. Translate the 3D object
2. Rotate the 3D object around x-axis
3. Rotate the 3D object around y-axis
4. Rotate the 3D object around z-axis
5. Scale the 3D object
6. Exit

Enter your choice (1-6): 2
Enter rotation angle around x-axis in degrees: 70



Choose a geometric operation:
1. Translate the 3D object
2. Rotate the 3D object around x-axis
3. Rotate the 3D object around y-axis
4. Rotate the 3D object around z-axis
5. Scale the 3D object
6. Exit

Enter your choice (1-6): 5
Enter scaling factor along x-axis: 3
Enter scaling factor along y-axis: 3
Enter scaling factor along z-axis: 3



4. Develop a program to demonstrate 2D transformation on basic objects

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

# Function to plot a rectangle
def plot_rectangle(vertices, title):
    plt.figure()
    plt.title(title)
    plt.gca().set_aspect('equal', adjustable='box')
    plt.grid(True)

    # Plot the rectangle
    rectangle = np.concatenate((vertices, [vertices[0]]))  # Close the rectangle
    plt.plot(rectangle[:, 0], rectangle[:, 1], 'b-')

    plt.show()

# Function to plot a triangle
def plot_triangle(vertices, title):
    plt.figure()
```

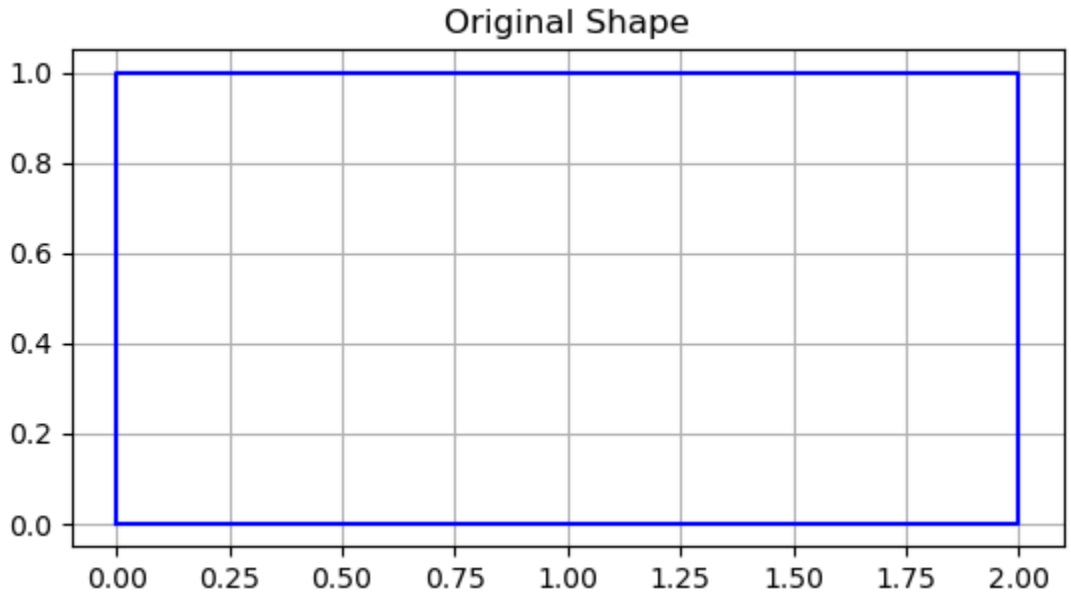
```
    plt.title(title)
    plt.gca().set_aspect('equal', adjustable='box')
    plt.grid(True)

    # Plot the triangle
    triangle = np.concatenate((vertices, [vertices[0]]))  # Close the triangle
    plt.plot(triangle[:, 0], triangle[:, 1], 'b-')

    plt.show()

# Function to translate a shape
def translate_shape(vertices, tx, ty):
    translated_vertices = vertices + np.array([tx, ty])
    return translated_vertices

# Function to rotate a shape (around the origin)
def rotate_shape(vertices, angle_degrees):
    angle_radians = np.radians(angle_degrees)
    rotation_matrix = np.array([[np.cos(angle_radians), -np.sin(angle_radians)],
                                [np.sin(angle_radians), np.cos(angle_radians)]])
    rotated_vertices = np.dot(vertices, rotation_matrix)
    return rotated_vertices

# Function to scale a shape (around the origin)
def scale_shape(vertices, sx, sy):
    scaled_vertices = vertices * np.array([sx, sy])
    return scaled_vertices

# Function to reflect a shape over the x-axis
def reflect_shape_x_axis(vertices):
    reflected_vertices = np.array(vertices)
    reflected_vertices[:, 1] = -reflected_vertices[:, 1]
    return reflected_vertices

# Main function to demonstrate 2D transformations on basic shapes
def main():
    # Prompt user for the type of shape (rectangle or triangle)
    shape_type = input("Enter shape type (rectangle or triangle): ").lower()

    if shape_type == 'rectangle':
        # Define vertices of a rectangle
        vertices = np.array([[0, 0], [2, 0], [2, 1], [0, 1]])
        plot_function = plot_rectangle
    elif shape_type == 'triangle':
```

```
    # Define vertices of a triangle
    vertices = np.array([[0, 0], [2, 0], [1, 2]])
    plot_function = plot_triangle
else:
    print("Invalid shape type. Please choose 'rectangle' or 'triangle'.")
    return

# Plot the original shape
plot_function(vertices, 'Original Shape')

# Prompt user for the type of transformation
while True:
    print("\nChoose a transformation:")
    print("1. Translate")
    print("2. Rotate")
    print("3. Scale")
    print("4. Reflect over x-axis")
    print("5. Exit")

    choice = input("Enter your choice (1-5): ")

    if choice == '1':
        tx = float(input("Enter translation along x-axis: "))
        ty = float(input("Enter translation along y-axis: "))
        vertices = translate_shape(vertices, tx, ty)
    elif choice == '2':
        angle_degrees = float(input("Enter rotation angle in degrees: "))
        vertices = rotate_shape(vertices, angle_degrees)
    elif choice == '3':
        sx = float(input("Enter scaling factor along x-axis: "))
        sy = float(input("Enter scaling factor along y-axis: "))
        vertices = scale_shape(vertices, sx, sy)
    elif choice == '4':
        vertices = reflect_shape_x_axis(vertices)
    elif choice == '5':
        break
    else:
        print("Invalid choice. Please enter a valid option.")

    # Plot the transformed shape
    plot_function(vertices, 'Transformed Shape')

print("Exiting...")
```

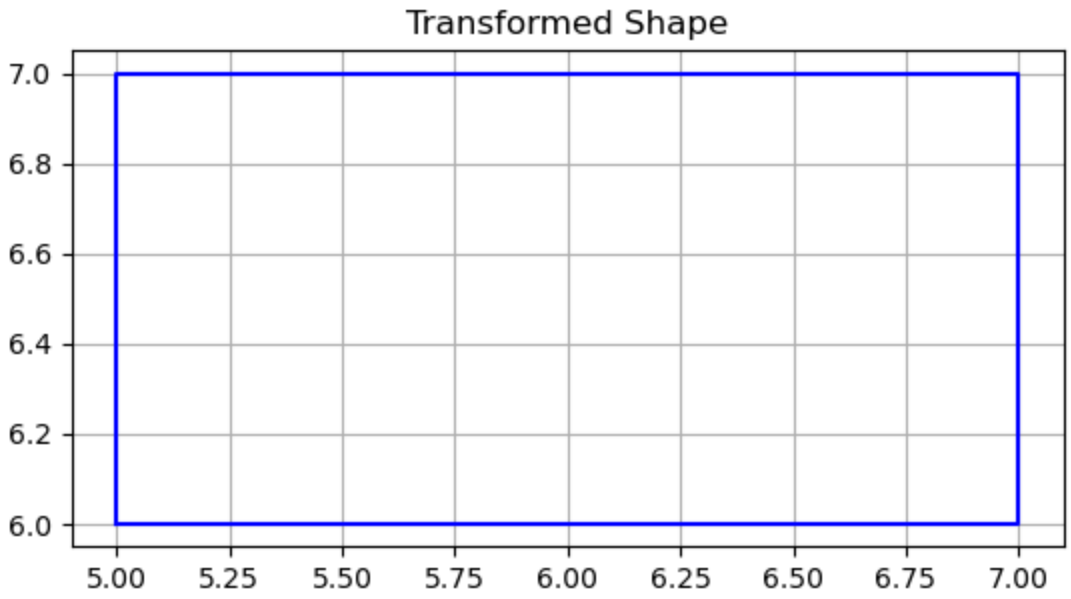
```
if __name__ == "__main__":
    main()
```

OUTPUT

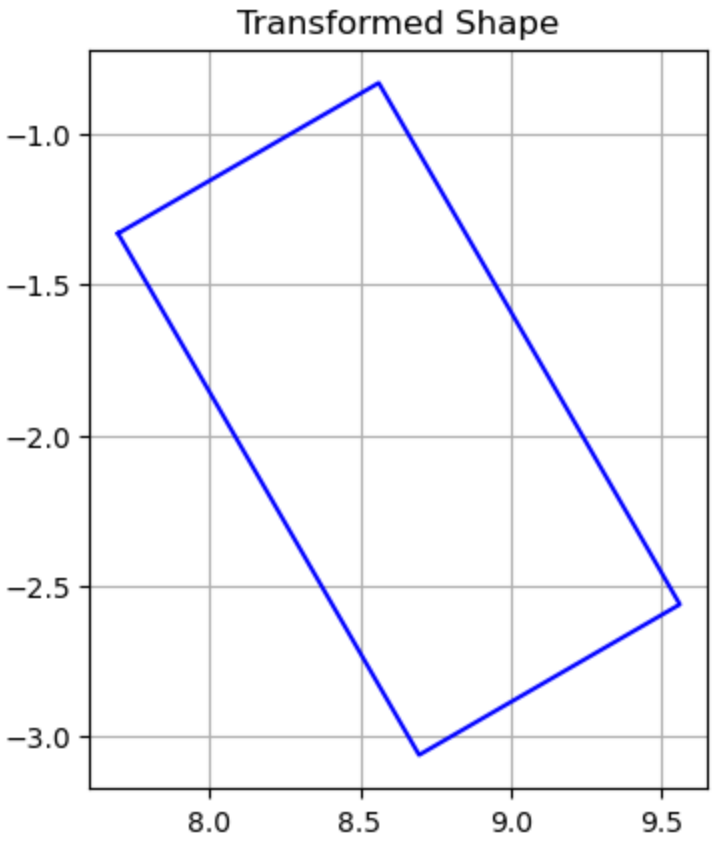
```
Enter shape type (rectangle or triangle): rectangle
```



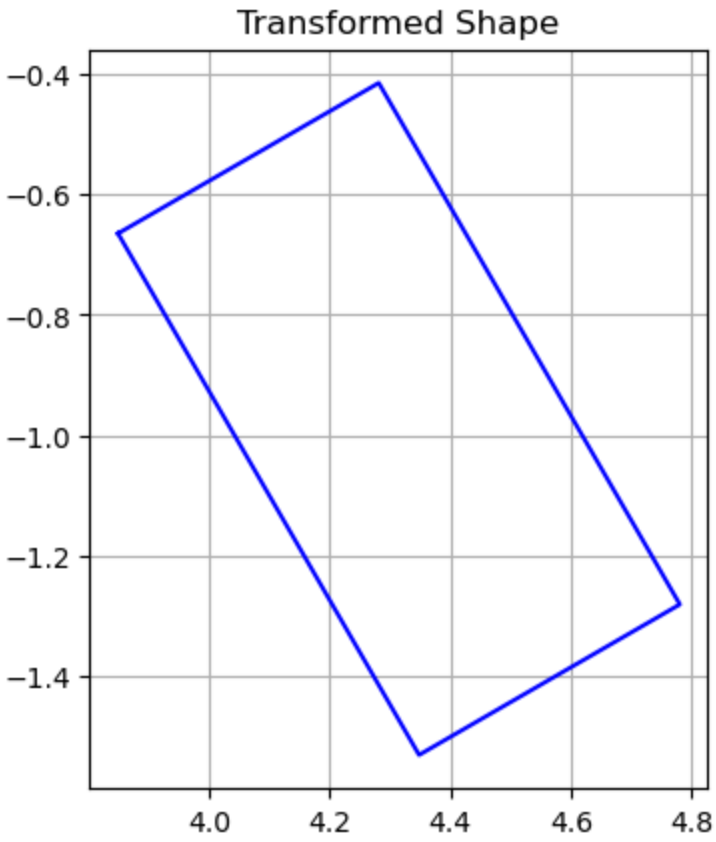

```
Choose a transformation:
1. Translate
2. Rotate
3. Scale
4. Reflect over x-axis
5. Exit
Enter your choice (1-5): 1
Enter translation along x-axis: 5
Enter translation along y-axis: 6
```



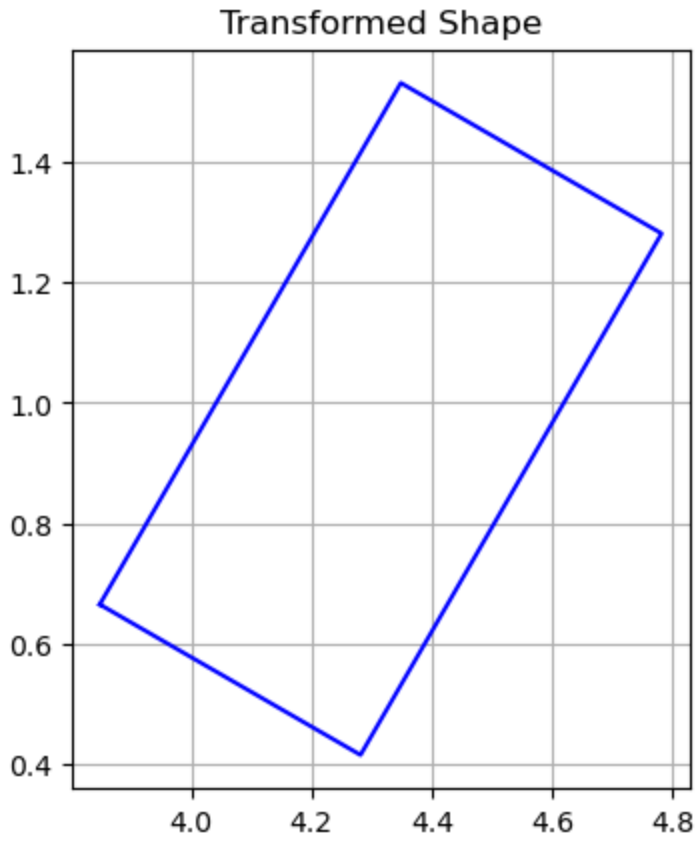
Choose a transformation:
1. Translate
2. Rotate
3. Scale
4. Reflect over x-axis
5. Exit
Enter your choice (1-5): 2
Enter rotation angle in degrees: 60



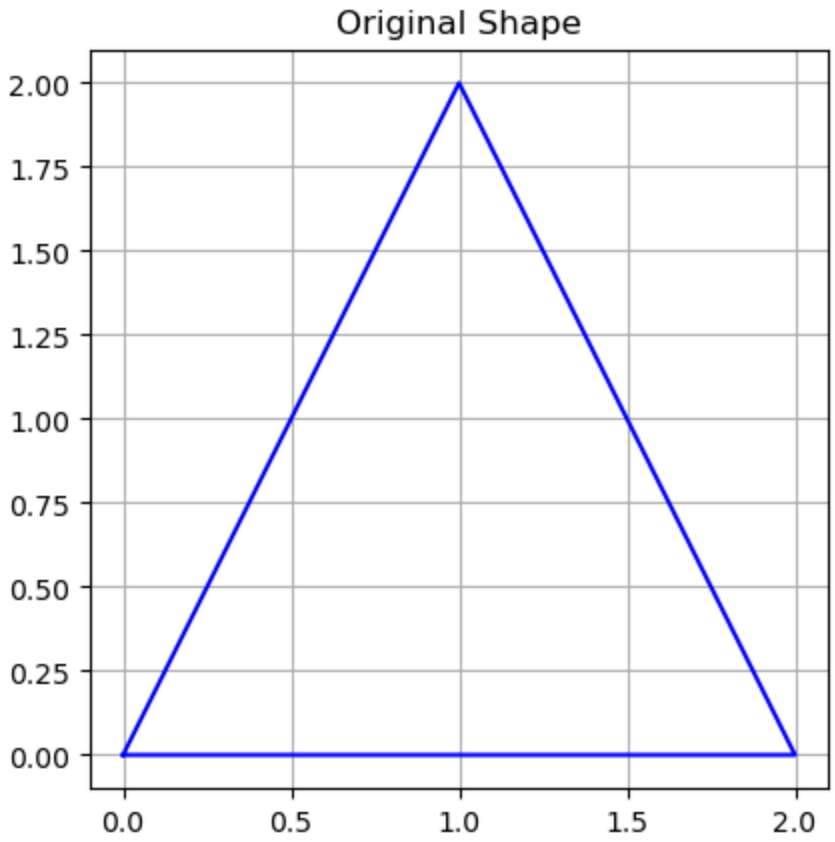
Choose a transformation:
1. Translate
2. Rotate
3. Scale
4. Reflect over x-axis
5. Exit
Enter your choice (1-5): 3
Enter scaling factor along x-axis: 0.5
Enter scaling factor along y-axis: 0.5



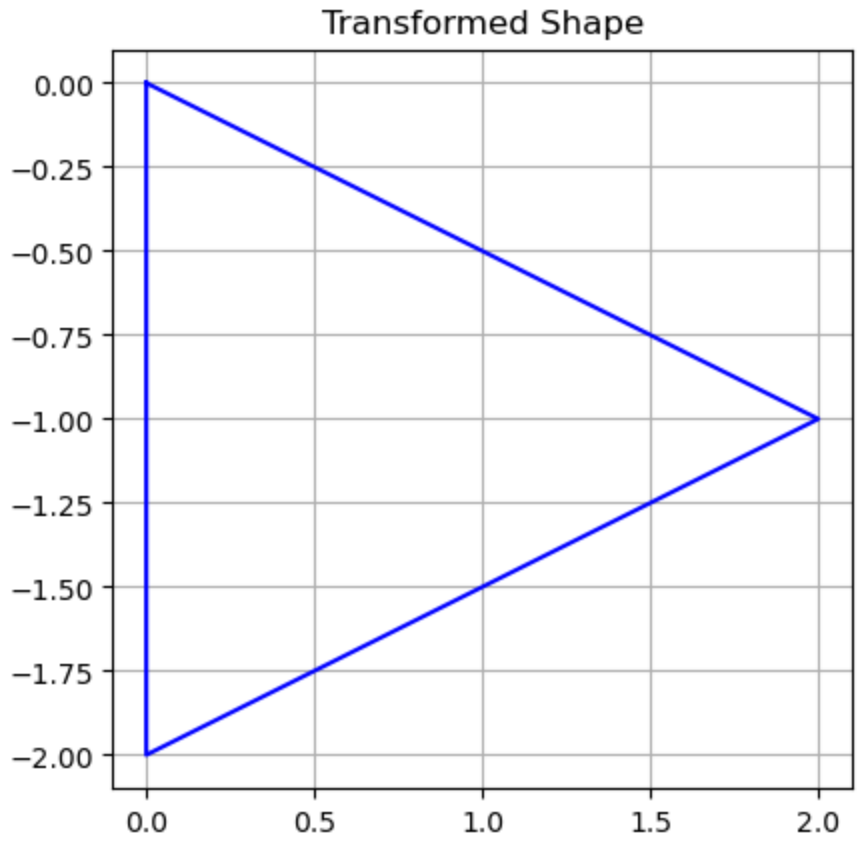
Choose a transformation:
1. Translate
2. Rotate
3. Scale
4. Reflect over x-axis
5. Exit
Enter your choice (1-5): 4



Enter shape type (rectangle or triangle): triangle



Choose a transformation:
1. Translate
2. Rotate
3. Scale
4. Reflect over x-axis
5. Exit
Enter your choice (1-5): 2
Enter rotation angle in degrees: 90



5. Develop a program to demonstrate 3D transformation on 3D objects

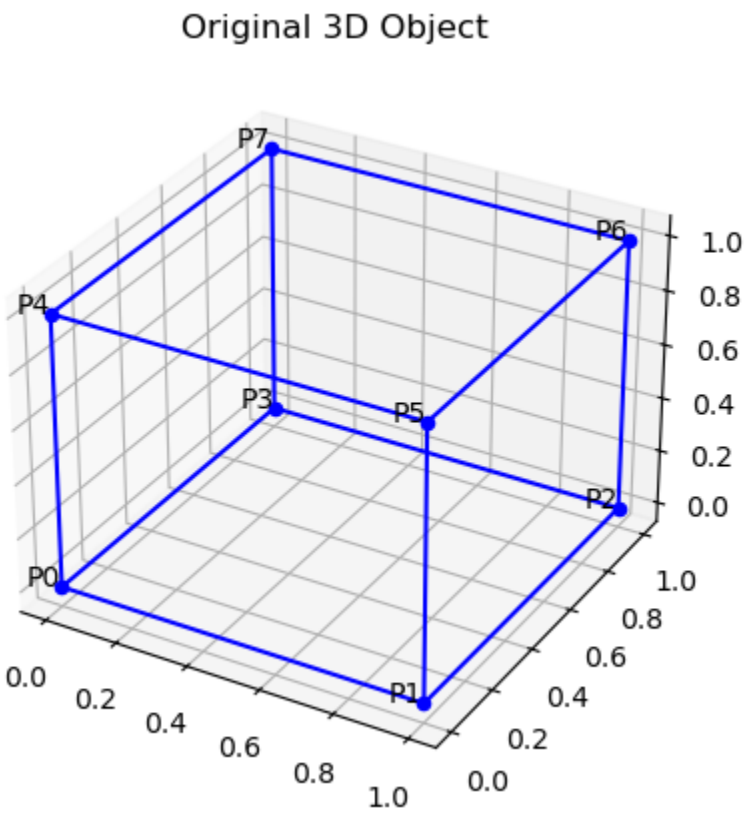
```
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

# Function to plot a 3D object
def plot_3d_object(vertices, title):
    fig = plt.figure()
    ax = fig.add_subplot(111, projection='3d')
    ax.set_title(title)

    # Plot the 3D object
    for i in range(len(vertices)):
        ax.scatter(vertices[i, 0], vertices[i, 1], vertices[i, 2], color='b')
        ax.text(vertices[i, 0], vertices[i, 1], vertices[i, 2], f'P{i}', ha='right')

    # Connect vertices to form edges of the object
    edges = [[0, 1, 2, 3, 0],
             [4, 5, 6, 7, 4],
             [0, 4], [1, 5], [2, 6], [3, 7]]
    for edge in edges:
```

```
        ax.plot(vertices[edge, 0], vertices[edge, 1], vertices[edge, 2], 'b-')

    plt.show()

# Function to translate a 3D object
def translate_3d_object(vertices, tx, ty, tz):
    translated_vertices = vertices + np.array([tx, ty, tz])
    return translated_vertices

# Function to rotate a 3D object around x, y, or z-axis
def rotate_3d_object(vertices, axis, angle_degrees):
    angle_radians = np.radians(angle_degrees)
    if axis == 'x':
        rotation_matrix = np.array([[1, 0, 0],
                                    [0, np.cos(angle_radians), -np.sin(angle_radians)],
                                    [0, np.sin(angle_radians), np.cos(angle_radians)]])
    elif axis == 'y':
        rotation_matrix = np.array([[np.cos(angle_radians), 0, np.sin(angle_radians)],
                                    [0, 1, 0],
                                    [-np.sin(angle_radians), 0, np.cos(angle_radians)]])
    elif axis == 'z':
        rotation_matrix = np.array([[np.cos(angle_radians), -np.sin(angle_radians), 0],
                                    [np.sin(angle_radians), np.cos(angle_radians), 0],
                                    [0, 0, 1]])
    else:
        print("Invalid axis. Rotation aborted.")
        return vertices

    rotated_vertices = np.dot(vertices, rotation_matrix)
    return rotated_vertices

# Function to scale a 3D object
def scale_3d_object(vertices, sx, sy, sz):
    scaled_vertices = vertices * np.array([sx, sy, sz])
    return scaled_vertices

# Function to reflect a 3D object over the xy-plane, yz-plane, or zx-plane
def reflect_3d_object(vertices, plane):
    if plane == 'xy':
        reflected_vertices = vertices * np.array([1, 1, -1])
    elif plane == 'yz':
        reflected_vertices = vertices * np.array([-1, 1, 1])
    elif plane == 'zx':
        reflected_vertices = vertices * np.array([1, -1, 1])
```

```
    else:
        print("Invalid plane. Reflection aborted.")
        return vertices

    return reflected_vertices

# Main function to demonstrate 3D transformations on a 3D object
def main():
    # Define vertices of a 3D object (e.g., cube)
    vertices = np.array([[0, 0, 0],
                         [1, 0, 0],
                         [1, 1, 0],
                         [0, 1, 0],
                         [0, 0, 1],
                         [1, 0, 1],
                         [1, 1, 1],
                         [0, 1, 1]])

    # Plot the original 3D object
    plot_3d_object(vertices, 'Original 3D Object')

    # Prompt user for the type of transformation
    while True:
        print("\nChoose a transformation:")
        print("1. Translate")
        print("2. Rotate")
        print("3. Scale")
        print("4. Reflect")
        print("5. Exit")

        choice = input("Enter your choice (1-5): ")

        if choice == '1':
            tx = float(input("Enter translation along x-axis: "))
            ty = float(input("Enter translation along y-axis: "))
            tz = float(input("Enter translation along z-axis: "))
            vertices = translate_3d_object(vertices, tx, ty, tz)
        elif choice == '2':
            axis = input("Enter rotation axis (x, y, z): ")
            angle_degrees = float(input("Enter rotation angle in degrees: "))
            vertices = rotate_3d_object(vertices, axis, angle_degrees)
        elif choice == '3':
            sx = float(input("Enter scaling factor along x-axis: "))
            sy = float(input("Enter scaling factor along y-axis: "))
```

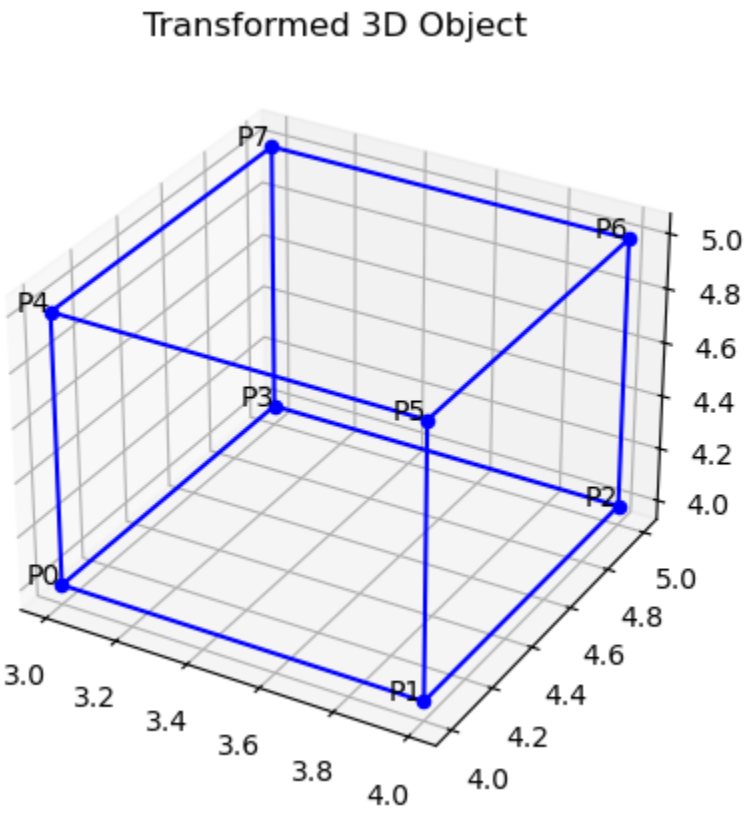
```
            sz = float(input("Enter scaling factor along z-axis: "))
            vertices = scale_3d_object(vertices, sx, sy, sz)
        elif choice == '4':
            plane = input("Enter reflection plane (xy, yz, zx): ")
            vertices = reflect_3d_object(vertices, plane)
        elif choice == '5':
            break
        else:
            print("Invalid choice. Please enter a valid option.")

        # Plot the transformed 3D object
        plot_3d_object(vertices, 'Transformed 3D Object')

    print("Exiting...")

if __name__ == "__main__":
    main()
```

OUTPUT

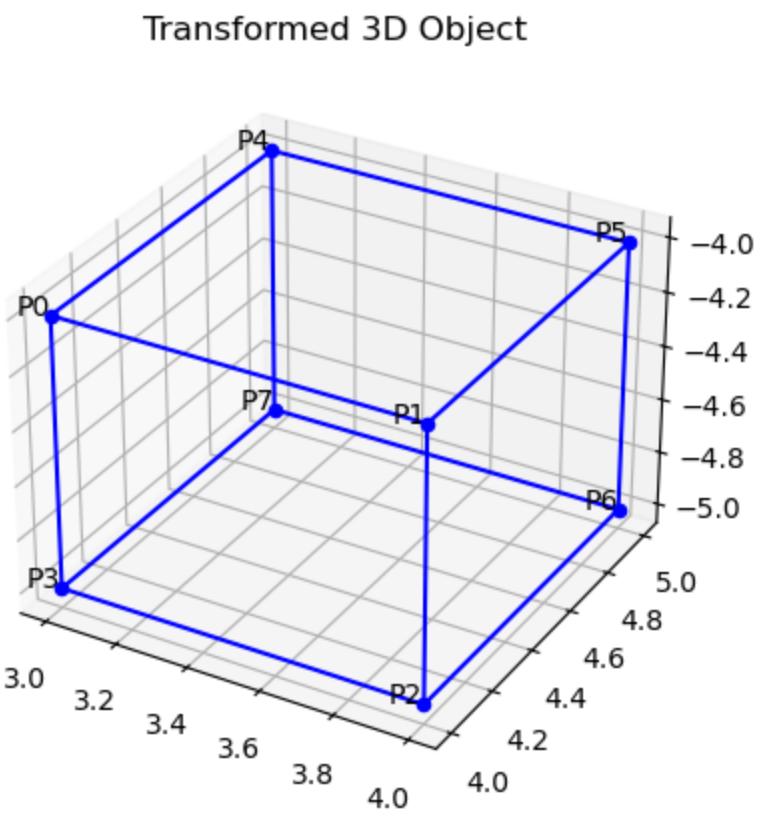


Choose a transformation:
1. Translate

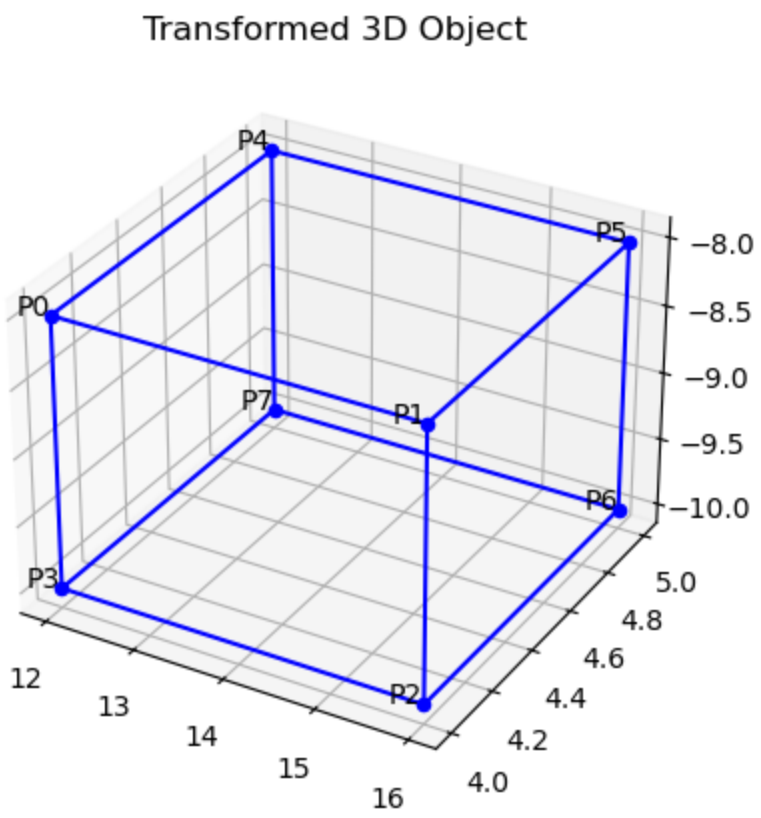
2. Rotate
3. Scale
4. Reflect
5. Exit
Enter your choice (1-5): 1
Enter translation along x-axis: 3
Enter translation along y-axis: 4
Enter translation along z-axis: 4



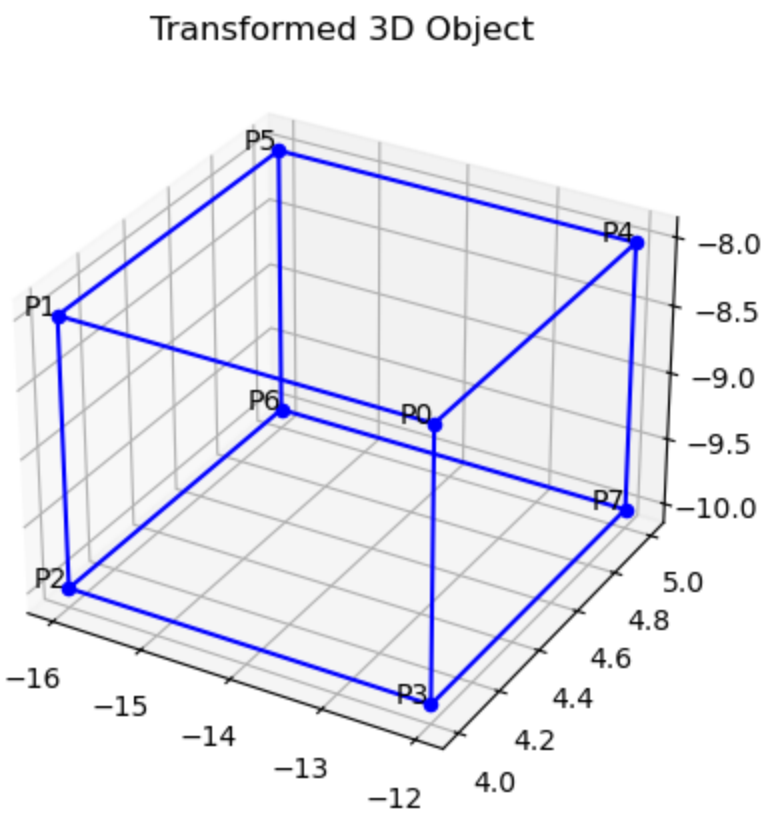
Choose a transformation:
1. Translate
2. Rotate
3. Scale
4. Reflect
5. Exit
Enter your choice (1-5): 2
Enter rotation axis (x, y, z): x
Enter rotation angle in degrees: 90



Choose a transformation:
1. Translate
2. Rotate
3. Scale
4. Reflect
5. Exit
Enter your choice (1-5): 3
Enter scaling factor along x-axis: 4
Enter scaling factor along y-axis: 1
Enter scaling factor along z-axis: 2



Choose a transformation:
1. Translate
2. Rotate
3. Scale
4. Reflect
5. Exit
Enter your choice (1-5): 4
Enter reflection plane (xy, yz, zx): yz



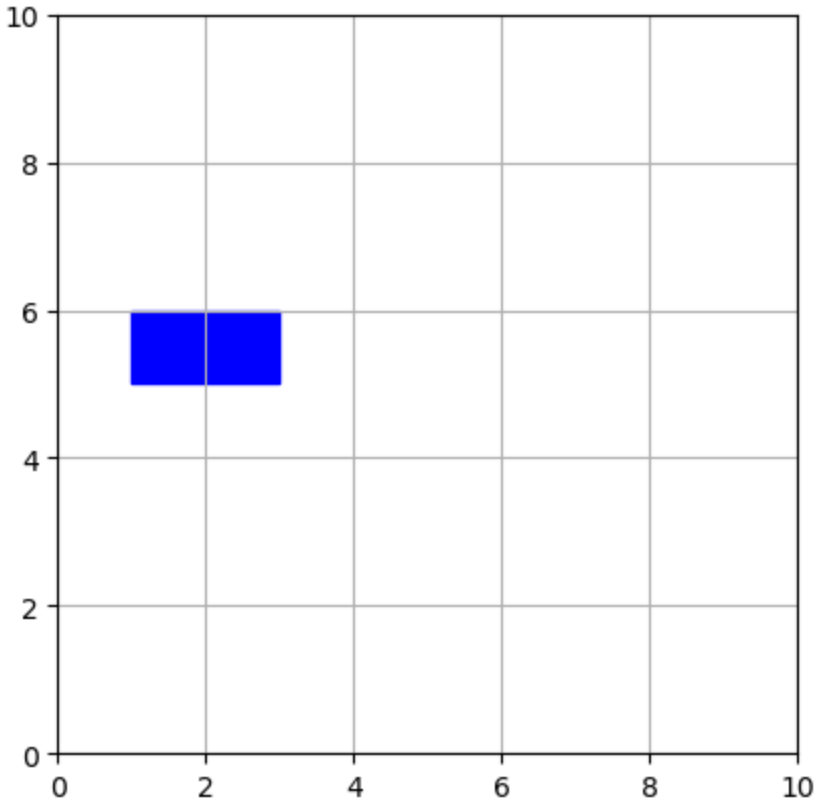
6. Develop a program to demonstrate Animation effects on simple objects.

```
import pygame
import sys

# Initialize Pygame
pygame.init()

# Set up display
width, height = 800, 600
window = pygame.display.set_mode((width, height))
pygame.display.set_caption('Moving Circle Animation')

# Define colors
black = (0, 0, 0)
white = (255, 255, 255)

# Initial position of the circle
x, y = width // 2, height // 2
radius = 20
dx, dy = 5, 5  # Movement step
```

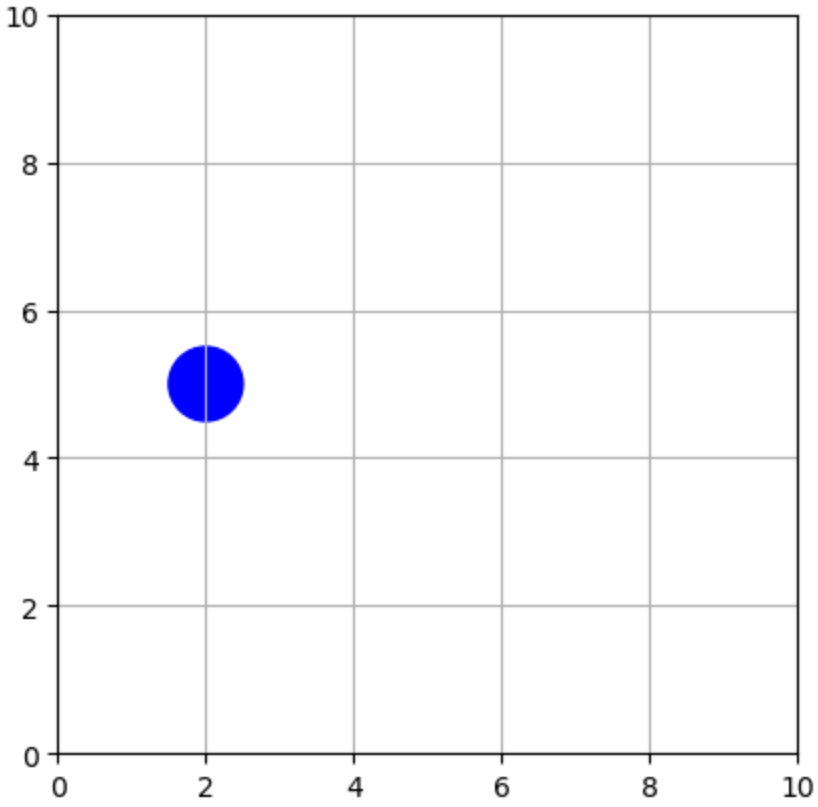
```
# Run the game loop
running = True
while running:
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            running = False

    # Move the circle
    x += dx
    y += dy

    # Bounce the circle off the edges
    if x - radius < 0 or x + radius > width:
        dx = -dx
    if y - radius < 0 or y + radius > height:
        dy = -dy

    # Fill the screen with black
    window.fill(black)

    # Draw the circle
    pygame.draw.circle(window, white, (x, y), radius)

    # Update the display
    pygame.display.flip()

    # Cap the frame rate
    pygame.time.Clock().tick(60)

# Quit Pygame
pygame.quit()
sys.exit()
```

OUTPUT

```
Enter shape type (rectangle or circle): rectangle
Enter motion style (linear or bounce): linear
```



Enter shape type (rectangle or circle): circle
Enter motion style (linear or bounce): bounce



7. Write a Program to read a digital image. Split and display image into 4 quadrants, up, down, right and left.

7. Write a Program to read a digital image. Split and display image into 4 quadrants, up, down, right and left.

```
import cv2

import numpy as np

# Read the image

img = cv2.imread(image_pat)



# Get the height and width of the image

height, width = img.shape[:2]
```

```
# Split the image into four quadrants
quad1 = img[:height//2, :width//2]
quad2 = img[:height//2, width//2:]
quad3 = img[height//2:, :width//2]
quad4 = img[height//2:, width//2:]
plt.figure(figsize=(10, 5))
plt.subplot(1, 2, 1)
plt.imshow(quad1)
plt.title("1")
plt.axis("off")

plt.subplot(1, 2, 2)
plt.imshow(quad2)
plt.title("2")
plt.axis("off")

plt.figure(figsize=(10, 5))
plt.subplot(1, 2, 1)
plt.imshow(quad3)
plt.title("3")
plt.axis("off")

plt.subplot(1, 2, 2)
```

```
plt.imshow(quad4)

plt.title("4")

plt.axis("off")


plt.show()
```

output

Additional pgm

```
# Up- down

import cv2

import numpy as np

# Read the image

img = cv2.imread(image_path)


# Get the height and width of the image

height, width = img.shape[:2]


up = img[:height//2,:]

down = img[height//2:,:]

plt.figure(figsize=(10, 5))

plt.subplot(1, 2, 1)

plt.imshow(up)

plt.title("Up")

plt.axis("off")
```

```
plt.subplot(1, 2, 2)
plt.imshow(down)
plt.title("down")
plt.axis("off")

plt.show()
```

```
# left- right
import cv2
import numpy as np
# Read the image
img = cv2.imread('/content/3.PNG')

# Get the height and width of the image
height, width = img.shape[:2]

left = img[:, :width//2]

right = img[:, width//2:]

up = img[:height//2,:]
down = img[height//2:,:]
```

```
quad1 = img[:height//2, :width//2]
quad2 = img[:height//2, width//2:]
quad3 = img[height//2:, :width//2]
quad4 = img[height//2:, width//2:]

plt.figure(figsize=(10, 5))
plt.subplot(1, 2, 1)
plt.imshow(left)
plt.title("left")
plt.axis("off")

plt.subplot(1, 2, 2)
plt.imshow(right)
plt.title("right")
plt.axis("off")

plt.show()
```

8. Write a program to show rotation, scaling, and translation on an image.

```
#Rotation and scaling of image
import cv2
```

```
def translate_image(image, dx, dy):
    rows, cols = image.shape[:2]
    translation_matrix = np.float32([[1, 0, dx], [0, 1, dy]])
    translated_image = cv2.warpAffine(image, translation_matrix, (cols, rows))
    return translated_image

# Read the image
image = cv2.imread('/content/sample_data/3.png')

# Get image dimensions
height, width = image.shape[:2]

# Calculate the center coordinates of the image
center = (width // 2, height // 2)
rotation_value = int(input("Enter the degree of Rotation:"))
scaling_value = int(input("Enter the zooming factor:"))
# Create the 2D rotation matrix
rotated = cv2.getRotationMatrix2D(center=center, angle=rotation_value, scale=1)
rotated_image = cv2.warpAffine(src=image, M=rotated, dsize=(width, height))
scaled = cv2.getRotationMatrix2D(center=center, angle=0, scale=scaling_value)
scaled_image = cv2.warpAffine(src=rotated_image, M=scaled, dsize=(width, height))
h = int(input("How many pixels you want the image to be translated horizontally? "))
```

```
v = int(input("How many pixels you want the image to be translated vertically? "))

translated_image = translate_image(scaled_image, dx=h, dy=v)

cv2.imwrite('Final_image.png', translated_image)
```

9. Read an image and extract and display low-level features such as edges, textures using filtering techniques.

```
import cv2

import numpy as np


# Load the image

image_path = "image/atc.jpg"  # Replace with the path to your image

img = cv2.imread(image_path)


# Convert the image to grayscale

gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)


# Edge detection

edges = cv2.Canny(gray, 100, 200)  # Use Canny edge detector


# Texture extraction
```

```
kernel = np.ones((5, 5), np.float32) / 25  # Define a 5x5 averaging kernel

texture = cv2.filter2D(gray, -1, kernel)  # Apply the averaging filter for texture extraction


# Display the original image, edges, and texture

cv2.imshow("Original Image", img)

cv2.imshow("Edges", edges)

cv2.imshow("Texture", texture)


# Wait for a key press and then close all windows

cv2.waitKey(0)

cv2.destroyAllWindows()
```

Output

10. Write a program to blur and smoothing an image.

```
img = cv2.imread("/content/sample_data/smaple.jpg",cv2.IMREAD_GRAYSCALE)

image_array = np.array(img)

print(image_array)

def sharpen():

  return np.array([[1,1,1],[1,1,1],[1,1,1]])

def filtering(image, kernel):

    m, n = kernel.shape

    if (m == n):
```

```
        y, x = image.shape
        y = y - m + 1 # shape of image - shape of kernel + 1
        x = x - m + 1
        new_image = np.zeros((y,x))
        for i in range(y):
            for j in range(x):
                    new_image[i][j] = np.sum(image[i:i+m, j:j+m]*kernel)
    return new_image
# Display the original and sharpened images
plt.figure(figsize=(10, 5))
plt.subplot(1, 2, 1)
plt.imshow(image_array,cmap='gray')
plt.title("Original Grayscale Image")
plt.axis("off")

plt.subplot(1, 2, 2)
plt.imshow(filtering(image_array, sharpen()),cmap='gray')
plt.title("Blurred Image")
plt.axis("off")
plt.show()
```

Extra programs:

1. #blur

```
import cv2


# Read the input image (replace 'your_image.jpg' with the actual image path)
image_path = '1.png'
image = cv2.imread(image_path)


# Apply average blur (simple box filter)
average_blur = cv2.blur(image, (5, 5))  # Adjust the kernel size as needed


# Apply Gaussian blur
gaussian_blur = cv2.GaussianBlur(image, (5, 5), 0)  # Adjust the kernel size and sigma as needed


# Display the results
cv2.imshow('Original Image', image)
cv2.waitKey(0)
cv2.imshow('Average Blurred Image', average_blur)
cv2.waitKey(0)
cv2.imshow('Gaussian Blurred Image', gaussian_blur)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

11. Write a program to contour an image.

```
import cv2

import numpy as np



image_path = '1.png'

image = cv2.imread(image_path)


# Convert the image to grayscale (contours work best on binary images)

gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)


# Apply thresholding (you can use other techniques like Sobel edges)

_, binary_image = cv2.threshold(gray, 127, 255, cv2.THRESH_BINARY)


# Find contours

contours, _ = cv2.findContours(binary_image, cv2.RETR_EXTERNAL,
cv2.CHAIN_APPROX_SIMPLE)


# Draw all contours on the original image

cv2.drawContours(image, contours, -1, (0, 255, 0), 3)
```

```
# Display the result

cv2.imshow('Contours', image)

cv2.waitKey(0)

cv2.destroyAllWindows()
```

output

12. Write a program to detect a face/s in an image.

```
import cv2


# Load the pre-trained Haar Cascade classifier for face detection

face_cascade = cv2.CascadeClassifier(cv2.data.haarcascades +
'haarcascade_frontalface_default.xml')

eye_cascade = cv2.CascadeClassifier(cv2.data.haarcascades + 'haarcascade_eye.xml')


# Read the input image (replace 'your_image.jpg' with the actual image path)

image_path = 'face.jpeg'

image = cv2.imread(image_path)


# Convert the image to grayscale

gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)


# Detect faces in the image
```

```
faces = face_cascade.detectMultiScale(gray, scaleFactor=1.3, minNeighbors=5)

# Draw rectangles around detected faces
for (x, y, w, h) in faces:
    cv2.rectangle(image, (x, y), (x + w, y + h), (255, 0, 0), 2)

# Save or display the result
cv2.imwrite('detected_faces.jpg', image)  # Save the result
cv2.imshow('Detected Faces', image)  # Display the result
cv2.waitKey(0)
cv2.destroyAllWindows()
```

2. another one

```
import cv2
import matplotlib.pyplot as plt
from IPython.display import display, clear_output

# Initialize the webcam
video_capture = cv2.VideoCapture(0)

while True:
    # Capture frame-by-frame
    ret, frame = video_capture.read()
```

```
        # Perform face detection (you can use any pre-trained face detection model)
        # For example, using Haar Cascade classifier:
        face_cascade = cv2.CascadeClassifier(cv2.data.haarcascades + 'haarcascade_frontalface_default.xml')
        faces = face_cascade.detectMultiScale(frame, scaleFactor=1.1, minNeighbors=5, minSize=(30, 30))

        # Draw rectangles around detected faces
        for (x, y, w, h) in faces:
        cv2.rectangle(frame, (x, y), (x + w, y + h), (0, 255, 0), 2)

        # Display the frame in the notebook
        plt.imshow(cv2.cvtColor(frame, cv2.COLOR_BGR2RGB))
        plt.axis('off')
        plt.show()
        clear_output(wait=True)

        # Press 'q' to exit the loop
        if cv2.waitKey(0) & 0xFF == ord('q'):
        break

# Release the webcam
video_capture.release()
```

```
cv2.destroyAllWindows()
```

```
#face detection with emotions

import cv2

from deepface import DeepFace


# Read an image (replace 'your_image.jpg' with the actual image path)

image_path = 'Angry.jpg'

image = cv2.imread(image_path)


# Detect faces in the image

faces = cv2.CascadeClassifier(cv2.data.haarcascades +
'haarcascade_frontalface_default.xml').detectMultiScale(image, scaleFactor=1.1,
minNeighbors=5)


# Predict emotions for each detected face

for (x, y, w, h) in faces:

        face_roi = image[y:y + h, x:x + w]

        result = DeepFace.analyze(face_roi)

        emotion = result[0]['emotion']
```

```
        emotion = dict(sorted(emotion.items(), key=lambda item: item[1]))
        emotion = (list(emotion.keys())[-1])
        cv2.putText(image, emotion, (x, y - 10), cv2.FONT_HERSHEY_SIMPLEX, 0.5, (0, 255, 0), 2)
        cv2.rectangle(image, (x, y), (x + w, y + h), (0, 255, 0), 2)

# Save or display the result
cv2.imwrite('emotion_detected.jpg', image)  # Save the result
cv2.imshow('Emotion Detection', image)  # Display the result
cv2.waitKey(0)
cv2.destroyAllWindows()
```

output

Extra programs in IP

```
1. import cv2

# Read the input image
image = cv2.imread(image_pat)

# Flip the image horizontally
flipped_image = cv2.flip(image,-1 )
```

```
# Save the flipped image

cv2.imwrite('flipped_image.png', flipped_image)
```

output

```
flipped_image = cv2.flip(image,1 )

flipped_image = cv2.flip(image,0 )
```

```
2. # Thresholding

import cv2

import numpy as np

import matplotlib.pyplot as plt

%matplotlib inline


#here 0 means that the image is loaded in gray scale format

gray_image = cv2.imread('/content/3.PNG',0)


ret,thresh_binary = cv2.threshold(gray_image,127,255,cv2.THRESH_BINARY)

ret,thresh_binary_inv = cv2.threshold(gray_image,127,255,cv2.THRESH_BINARY_INV)

ret,thresh_trunc = cv2.threshold(gray_image,127,255,cv2.THRESH_TRUNC)

ret,thresh_tozero = cv2.threshold(gray_image,127,255,cv2.THRESH_TOZERO)

ret,thresh_tozero_inv = cv2.threshold(gray_image,127,255,cv2.THRESH_TOZERO_INV)
```

```
#DISPLAYING THE DIFFERENT THRESHOLDING STYLES

names = ['Oiriginal Image','BINARY','THRESH_BINARY_INV','THRESH_TRUNC','THRESH_TOZERO','THRESH_TOZERO_INV']

images = gray_image,thresh_binary,thresh_binary_inv,thresh_trunc,thresh_tozero,thresh_tozero_inv


for i in range(6):

    plt.subplot(2,3,i+1),plt.imshow(images[i],'gray')

    plt.title(names[i])

    plt.xticks([]),plt.yticks([])


plt.show()
```

output

Extra Programs to run in google colab

Pgm1:

```
img = cv2.imread("/content/sample_data/1.png",cv2.IMREAD_GRAYSCALE)

image_array = np.array(img)

print(image_array)

def sharpen():

  return np.array([

[0,-1,0],[-1,5,-1],[0,-1,0]

  ])
```

```
def filtering(image, kernel):
    m, n = kernel.shape
    if (m == n):
        y, x = image.shape
        y = y - m + 1 # shape of image - shape of kernel + 1
        x = x - m + 1
        new_image = np.zeros((y,x))
        for i in range(y):
            for j in range(x):
                new_image[i][j] = np.sum(image[i:i+m, j:j+m]*kernel)
    return new_image
# Display the original and sharpened images
plt.figure(figsize=(10, 5))
plt.subplot(1, 2, 1)
plt.imshow(image_array,cmap='gray')
plt.title("Original Grayscale Image")
plt.axis("off")


plt.subplot(1, 2, 2)
plt.imshow(filtering(image_array, sharpen()),cmap='gray')
plt.title("Blurred Image")
plt.axis("off")
plt.show()
```

```
"""# New Section"""
```

Pgm2:

```
import numpy as np
import cv2
import matplotlib.pyplot as plt
from google.colab.patches import cv2_imshow
img = cv2.imread("C:\Users\HP-PC\Pictures\smaple.jpg")
image_array = np.array(img)
def rgb2gray(image):
    return np.dot(image[..., :3], [0.2989, 0.5870, 0.1140])
grayscale_image = rgb2gray(image_array)
print(image_array.shape)
plt.figure(figsize=(10, 5))
plt.subplot(1, 2, 1)
plt.imshow(image_array)
plt.title("Original  Image")
plt.axis("off")
plt.subplot(1, 2, 2)
plt.imshow(grayscale_image,cmap='gray')
plt.title("Grayscale Image")
```

```
plt.axis("off")
plt.show()
grayscale_image[24,8]
def sharpen():
  return np.array([
[0,-1,0],[-1,5,-1],[0,-1,0]
 ])
def filtering(image, kernel):
   m, n = kernel.shape
   if (m == n):
      y, x = image.shape
      y = y - m + 1 # shape of image - shape of kernel + 1
      x = x - m + 1
      new_image = np.zeros((y,x))
      for i in range(y):
         for j in range(x):
            new_image[i][j] = np.sum(image[i:i+m, j:j+m]*kernel)
   return
plt.figure(figsize=(10, 5))
plt.subplot(1, 2, 1)
plt.imshow(image_array, cmap='gray')
plt.title("Original  Image")
plt.axis("off")
```

```
plt.subplot(1, 2, 2)
plt.imshow(filtering(grayscale_image,sharpen()),cmap='gray')
plt.title("Sharpen Image")
plt.axis("off")
plt.show()
```

pgm3:

```
#Color image to Gray image
import numpy as np
import cv2
import matplotlib.pyplot as plt
def rgb2gray(image):
    return np.dot(image[..., :3], [0.2989, 0.5870, 0.1140])
    filename = '1.png'
image = cv2.imread("/content/sample_data/JS pp photo.jpg")
image_array = np.array(image)
grayscale_image = rgb2gray(image_array)
print(image_array.shape)
plt.figure(figsize=(10, 5))
plt.subplot(1, 2, 1)
plt.imshow(image_array)
plt.title("Original  Image")
```

```
plt.axis("off")
plt.subplot(1, 2, 2)
plt.imshow(grayscale_image,cmap='gray')
plt.title("Grayscale Image")
plt.axis("off")
plt.show()
```

pgm4:

```
#Rotating an image
filename = '/content/sample_data/JS pp photo.jpg'
image = cv2.imread(filename,cv2.IMREAD_UNCHANGED)
image_array = np.array(image)

def get_rotation(angle):
    angle = np.radians(angle)
    return np.array([
[np.cos(angle), -np.sin(angle), 0],
[np.sin(angle), np.cos(angle), 0],
[0, 0, 1]
])
img_transformed = np.zeros((400,400,3), dtype=np.uint8)
R1 = get_rotation(45)
```

```
for i, row in enumerate(image_array):
    for j, col in enumerate(row):
        pixel_data = image_array[i, j, :]


        input_coords = np.array([i, j,1])
        i_out, j_out, _ = (R1 @ input_coords).astype(int)
        img_transformed[i_out+150,j_out, :] = pixel_data
plt.figure(figsize=(10, 5))
plt.subplot(1, 2, 1)
plt.imshow(image_array)
plt.title("Original Grayscale Image")
plt.axis("off")


plt.subplot(1, 2, 2)
plt.imshow(img_transformed)
plt.title("Rotated Image")
plt.axis("off")
plt.show()


cv2.imwrite('/content/sample_data/rotate.jpg',img_transformed)
```

pgm5:

```
import cv2
```

```
from google.colab.patches import cv2_imshow

img  = cv2.imread('/content/sample_data/Colors.jpg',-1)

cv2_imshow(img)

cv2.waitKey(0)

cv2.destroyAllWindows()


#prgm-8

image = cv2.imread('/content/sample_data/Rainbow.jpg',1)

B, G, R = cv2.split(image)

# Corresponding channels are separated


cv2_imshow(image)

cv2.waitKey(0)


cv2_imshow(B)

cv2.waitKey(0)


cv2_imshow(G)

cv2.waitKey(0)


cv2_imshow(R)

cv2.waitKey(0)
```

```
cv2.destroyAllWindows()
```

Pgm6:

```
import cv2
from google.colab.patches import cv2_imshow
img1 = cv2.imread("/content/sample_data/do_not_copy.png")
#img1=cv2.imread("")
print(img1.shape)
img2 = cv2.imread("/content/sample_data/3.png")

img2 = cv2.resize(img2,(224,225))
print(img2.shape)
final_img = cv2.addWeighted(img2,1,img1,0.7,0)
cv2_imshow(final_img)

cv2.imwrite('/content/sample_data/rgbchannels.jpg',image)

filename = '/content/sample_data/smaple.jpg'
image = cv2.imread(filename,cv2.IMREAD_GRAYSCALE)
image_array = np.array(image)

def sharpen():
```

```
    return np.array([
[0,-1, 0],
[-1,10, -1],
[0,-1, 0]
])
def filtering(image, kernel):
    m, n = kernel.shape
    if (m == n):
        y, x = image.shape
        y = y - m + 1 # shape of image - shape of kernel + 1
        x = x - m + 1
        new_image = np.zeros((y,x))
        for i in range(y):
            for j in range(x):
                new_image[i][j] = np.sum(image[i:i+m, j:j+m]*kernel)
    return new_image
# Display the original and sharpened images
plt.figure(figsize=(10, 5))
plt.subplot(1, 2, 1)
plt.imshow(image_array, cmap='gray')
plt.title("Original Grayscale Image")
plt.axis("off")
```

```
plt.subplot(1, 2, 2)
plt.imshow(filtering(image_array, sharpen()),cmap='gray')
plt.title("Sharpened Image")
plt.axis("off")
plt.show()
```

pgm7:

```
img = cv2.imread("/content/sample_data/smaple.jpg",cv2.IMREAD_GRAYSCALE)
image_array = np.array(img)
print(image_array)
def sharpen():
  return np.array([
[1/9,1/9,1/9],[1/9,1/9,1/9],[1/9,1/9,1/9]
  ])
def filtering(image, kernel):
    m, n = kernel.shape
    if (m == n):
        y, x = image.shape
        y = y - m + 1 # shape of image - shape of kernel + 1
        x = x - m + 1
        new_image = np.zeros((y,x))
        for i in range(y):
```

```
            for j in range(x):
                new_image[i][j] = np.sum(image[i:i+m, j:j+m]*kernel)
    return new_image
# Display the original and sharpened images
plt.figure(figsize=(10, 5))
plt.subplot(1, 2, 1)
plt.imshow(image_array,cmap='gray')
plt.title("Original Grayscale Image")
plt.axis("off")

plt.subplot(1, 2, 2)
plt.imshow(filtering(image_array, sharpen()),cmap='gray')
plt.title("Blurred Image")
plt.axis("off")
plt.show()
```

pgm8:

```
#Guassian Blur
img = cv2.imread("/content/sample_data/smaple.jpg",cv2.IMREAD_GRAYSCALE)
image_array = np.array(img)
print(image_array)
def sharpen():
```

```
 return np.array([
[1/16,2/16,1/16],[2/16,4/16,2/16],[1/16,2/16,1/16]
 ])
def filtering(image, kernel):
   m, n = kernel.shape
   if (m == n):
      y, x = image.shape
      y = y - m + 1 # shape of image - shape of kernel + 1
      x = x - m + 1
      new_image = np.zeros((y,x))
      for i in range(y):
         for j in range(x):
            new_image[i][j] = np.sum(image[i:i+m, j:j+m]*kernel)
   return new_image
# Display the original and sharpened images
plt.figure(figsize=(10, 5))
plt.subplot(1, 2, 1)
plt.imshow(image_array,cmap='gray')
plt.title("Original Grayscale Image")
plt.axis("off")

plt.subplot(1, 2, 2)
plt.imshow(filtering(image_array, sharpen()),cmap='gray')
```

```
plt.title("Guassian Blurred Image")
plt.axis("off")
plt.show()
```

pgm9:

```
#Ridge Detection
img = cv2.imread("/content/sample_data/JS pp photo.jpg",cv2.IMREAD_GRAYSCALE)
image_array = np.array(img)
print(image_array)
def sharpen():
  return np.array([
[0,-1,0],[-1,0,-1],[0,-1,0]
  ])
def filtering(image, kernel):
    m, n = kernel.shape
    if (m == n):
        y, x = image.shape
        y = y - m + 1 # shape of image - shape of kernel + 1
        x = x - m + 1
        new_image = np.zeros((y,x))
        for i in range(y):
```

```
        for j in range(x):
            new_image[i][j] = np.sum(image[i:i+m, j:j+m]*kernel)
    return new_image
# Display the original and sharpened images
plt.figure(figsize=(10, 5))
plt.subplot(1, 2, 1)
plt.imshow(image_array,cmap='gray')
plt.title("Original Grayscale Image")
plt.axis("off")

plt.subplot(1, 2, 2)
plt.imshow(filtering(image_array, sharpen()),cmap='gray')
plt.title("Ridge detection Image")
plt.axis("off")
plt.show()
```

pgm10:

```
#Edge Detection
img = cv2.imread("/content/sample_data/JS pp photo.jpg",cv2.IMREAD_GRAYSCALE)
image_array = np.array(img)
print(image_array)
def sharpen():
```

```
 return np.array([
[-1,-1,-1],[-1,8,-1],[-1,-1,-1]
 ])
def filtering(image, kernel):
   m, n = kernel.shape
   if (m == n):
      y, x = image.shape
      y = y - m + 1 # shape of image - shape of kernel + 1
      x = x - m + 1
      new_image = np.zeros((y,x))
      for i in range(y):
         for j in range(x):
            new_image[i][j] = np.sum(image[i:i+m, j:j+m]*kernel)
   return new_image
# Display the original and sharpened images
plt.figure(figsize=(10, 5))
plt.subplot(1, 2, 1)
plt.imshow(image_array,cmap='gray')
plt.title("Original Grayscale Image")
plt.axis("off")


plt.subplot(1, 2, 2)
plt.imshow(filtering(image_array, sharpen()),cmap='gray')
```

```
plt.title("edge detection Image")

plt.axis("off")

plt.show()
```

pgm11:

```
# comment


import cv2

from google.colab.patches import cv2_imshow

image = cv2.imread('/content/3.png',1)

B, G, R = cv2.split(image)

# Corresponding channels are separated


cv2_imshow(image)

cv2.waitKey(0)


cv2_imshow(B)

cv2.waitKey(0)


cv2_imshow(G)

cv2.waitKey(0)
```

```
cv2_imshow(R)

cv2.waitKey(0)


cv2.destroyAllWindows()


#Image resizing 1

import cv2

from google.colab.patches import cv2_imshow

img1 = cv2.imread("/content/sample_data/do_not_copy.png")

print(img1.shape)

img2 = cv2.imread("/content/sample_data/3.png")


img2 = cv2.resize(img2,(224,225))

print(img2.shape)

final_img = cv2.addWeighted(img2,1,img1,0.7,0)

cv2_imshow(final_img)

#cv2.waitKey(0)

#cv2.destroyAllWindows()


#Image resizing 2

import cv2

from google.colab.patches import cv2_imshow

img1 = cv2.imread("/content/sample_data/circle.png")
```

```
#img1=cv2.imread("")

print(img1.shape)

img2 = cv2.imread("/content/sample_data/square.png")


img2 = cv2.resize(img2,(img1.shape[1],img1.shape[0]))

print(img2.shape)

final_img = cv2.addWeighted(img1,0.7,img2,0.6,0)

cv2_imshow(final_img)


#Image subtraction

import cv2

from google.colab.patches import cv2_imshow

img_1 = cv2.imread('/content/sample_data/square.png')

print(img_1.shape)

img_2 = cv2.imread('/content/sample_data/circle.png')

print(img_2.shape)

final_img = cv2.subtract(img_2,img_1)

cv2_imshow(final_img)

cv2.waitKey(0)

cv2.destroyAllWindows()


#Image subtraction for gray images

img_1 = cv2.imread('/content/sample_data/square.png',0)
```

```
img_2 = cv2.imread('/content/sample_data/circle.png',0)
img_2  = cv2.resize(img_2,(img1.shape[1],img_1.shape[0]))
final_img = cv2.subtract(img_2,img_1)
cv2_imshow(final_img)
```

pgm12:

```
#Image translation
import cv2
import numpy as np

def translate_image(image, dx, dy):
    rows, cols = image.shape[:2]
    translation_matrix = np.float32([[1, 0, dx], [0, 1, dy]])
    translated_image = cv2.warpAffine(image, translation_matrix, (cols, rows))
    return translated_image

# Read the image
image = cv2.imread('/content/sample_data/circle.png')

# Translate the image by dx=50 pixels and dy=30 pixels
translated_image = translate_image(image, dx=20, dy=30)
```

```
# Save the translated image to disk
cv2.imwrite('translated_image.png', translated_image)
```

pgm13:

```
import cv2
import numpy as np

def translate_image(image, dx, dy):
    rows, cols = image.shape[:2]
    translation_matrix = np.float32([[1, 0, dx], [0, 1, dy]])
    translated_image = cv2.warpAffine(image, translation_matrix, (cols, rows))
    return translated_image

# Read the image
image = cv2.imread('/content/sample_data/circle.png')

# Translate the image by dx=20 pixels and dy=0 pixels, translate horizontally by 20px
translated_image = translate_image(image, dx=20, dy=0)
# Save the translated image to disk
cv2.imwrite('translated_image.png', translated_image)
```

pgm14:

```
#Image Zoom in

import cv2

import numpy as np

# Read the image

image = cv2.imread('//content/sample_data/circle.png')


# Get image dimensions

height, width = image.shape[:2]


# Calculate the center coordinates of the image

center = (width / 2, height / 2)


# Create the 2D rotation matrix

rotate_matrix = cv2.getRotationMatrix2D(center=center, angle=30, scale=2)


# Rotate the image

rotated_image = cv2.warpAffine(src=image, M=rotate_matrix, dsize=(width, height))


# Display the original and rotated images


def translate_image(image, dx, dy):

    rows, cols = image.shape[:2]

    translation_matrix = np.float32([[1, 0, dx], [0, 1, dy]])
```

```
    translated_image = cv2.warpAffine(image, translation_matrix, (cols, rows))
    return translated_image

# Read the image
#image = cv2.imread('1.jpg')

# Translate the image by dx=50 pixels and dy=30 pixels
translated_image = translate_image(rotated_image, dx=00, dy=20)
# Save the translated image to disk
cv2.imwrite('translated_image.png', translated_image)
```

pgm15:

```
#Rotation and scaling of image
import cv2

def translate_image(image, dx, dy):
    rows, cols = image.shape[:2]
    translation_matrix = np.float32([[1, 0, dx], [0, 1, dy]])
    translated_image = cv2.warpAffine(image, translation_matrix, (cols, rows))
    return translated_image

# Read the image
```

```
image = cv2.imread('/content/sample_data/3.png')


# Get image dimensions

height, width = image.shape[:2]


# Calculate the center coordinates of the image

center = (width // 2, height // 2)

rotation_value = int(input("Enter the degree of Rotation:"))

scaling_value = int(input("Enter the zooming factor:"))

# Create the 2D rotation matrix

rotated = cv2.getRotationMatrix2D(center=center, angle=rotation_value, scale=1)

rotated_image = cv2.warpAffine(src=image, M=rotated, dsize=(width, height))

scaled = cv2.getRotationMatrix2D(center=center, angle=0, scale=scaling_value)

scaled_image = cv2.warpAffine(src=rotated_image, M=scaled, dsize=(width, height))

h = int(input("How many pixels you want the image to be translated horizontally? "))

v = int(input("How many pixels you want the image to be translated vertically? "))

translated_image = translate_image(scaled_image, dx=h, dy=v)

cv2.imwrite('Final_image.png', translated_image)
```

pgm16:

```
#Splitting an image into 4 equal quadrants
```

```
import cv2

import numpy as np

# Read the image

from google.colab.patches import cv2_imshow

img = cv2.imread('/content/sample_data/3.png')


# Get the height and width of the image

height, width = img.shape[:2]


# Split the image into four quadrants

quad1 = img[:height//2, :width//2]

quad2 = img[:height//2, width//2:]

quad3 = img[height//2:, :width//2]

quad4 = img[height//2:, width//2:]


# Display the four quadrants

cv2_imshow(quad1)

cv2_imshow(quad2)

cv2_imshow(quad3)

cv2_imshow(quad4)
```

pgm17:

```
# Commented out IPython magic to ensure Python compatibility.
#displaying an Image with different levels of thresholds
import cv2
import numpy as np
import matplotlib.pyplot as plt
# %matplotlib inline

#here 0 means that the image is loaded in gray scale format
gray_image = cv2.imread('/content/sample_data/3.png',0)

ret,thresh_binary = cv2.threshold(gray_image,127,255,cv2.THRESH_BINARY)
ret,thresh_binary_inv = cv2.threshold(gray_image,127,255,cv2.THRESH_BINARY_INV)
ret,thresh_trunc = cv2.threshold(gray_image,127,255,cv2.THRESH_TRUNC)
ret,thresh_tozero = cv2.threshold(gray_image,127,255,cv2.THRESH_TOZERO)
ret,thresh_tozero_inv = cv2.threshold(gray_image,127,255,cv2.THRESH_TOZERO_INV)

#DISPLAYING THE DIFFERENT THRESHOLDING STYLES
names = ['Oiriginal Image','BINARY','THRESH_BINARY_INV','THRESH_TRUNC','THRESH_TOZERO','THRESH_TOZERO_INV']
images = gray_image,thresh_binary,thresh_binary_inv,thresh_trunc,thresh_tozero,thresh_tozero_inv
```

```
for i in range(6):
    plt.subplot(2,3,i+1),plt.imshow(images[i],'gray')
    plt.title(names[i])
    plt.xticks([]),plt.yticks([])

plt.show()
```

pgm17:

```
#Zooming out of an image
import cv2
from google.colab.patches import cv2_imshow
# Read the input image
original_image = cv2.imread('/content/sample_data/3.png')

# Zoom out (reduce size by half)
zoomed_out_image = cv2.pyrDown(original_image)

# Display the original and zoomed-out images
cv2_imshow(original_image)
cv2_imshow(zoomed_out_image)
cv2.waitKey(0)
```

```
cv2.destroyAllWindows()
```

pgm18:

```
import cv2
import numpy as np
# Read the image
from google.colab.patches import cv2_imshow
img = cv2.imread('/content/3.png')

# Get the height and width of the image
height, width = img.shape[:2]

# Split the image into four quadrants
quad1 = img[:height//2, :width//2]
quad2 = img[:height//2, width//2:]
quad3 = img[height//2:, :width//2]
quad4 = img[height//2:, width//2:]

# Display the four quadrants
cv2.imshow('quadrant1',quad1)
cv2.imshow('quadrant2',quad2)
```

```
cv2.imshow('quadrant3',quad3)

cv2.imshow('quadrant4',quad4)


cv2.imwrite('quad1.png', quad1)

cv2.imwrite('quad2.png', quad2)

cv2.imwrite('quad3.png', quad3)

cv2.imwrite('quad4.png', quad4)
```

pgm19:

```
import cv2

import numpy as np

# Read the image

img = cv2.imread('/content/sample_data/3.png')


# Get the height and width of the image

height, width = img.shape[:2]


# Split the image into four quadrants

quad1 = img[:height//2, :width//2]

quad2 = img[:height//2, width//2:]

quad3 = img[height//2:, :width//2]

quad4 = img[height//2:, width//2:]
```

```
plt.figure(figsize=(10, 5))
plt.subplot(1, 2, 1)
plt.imshow(quad1)
plt.title("1")
plt.axis("off")

plt.subplot(1, 2, 2)
plt.imshow(quad2)
plt.title("2")
plt.axis("off")

plt.figure(figsize=(10, 5))
plt.subplot(1, 2, 1)
plt.imshow(quad3)
plt.title("3")
plt.axis("off")

plt.subplot(1, 2, 2)
plt.imshow(quad4)
plt.title("4")
plt.axis("off")

plt.show()
```

pgm20:

```
# Up- down
import cv2
import numpy as np
# Read the image

img = cv2.imread('/content/3.png')

# Get the height and width of the image
height, width = img.shape[:2]

up = img[:height//2,:]
down = img[height//2:,:]
plt.figure(figsize=(10, 5))
plt.subplot(1, 2, 1)
plt.imshow(up)
plt.title("Up")
plt.axis("off")

plt.subplot(1, 2, 2)
```

```
plt.imshow(down)

plt.title("down")

plt.axis("off")


plt.show()
```

pgm21:

```
# Up- down

import cv2

import numpy as np

# Read the image

img = cv2.imread('/content/3.png')


# Get the height and width of the image

height, width = img.shape[:2]


up = img[:height//2,:]

down = img[height//2:,:]

plt.figure(figsize=(10, 5))

plt.subplot(2, 1, 1)

plt.imshow(up)

plt.title("Up")
```

```
plt.axis("off")


plt.subplot(2, 1, 2)
plt.imshow(down)
plt.title("down")
plt.axis("off")


plt.show()
```

pgm21:

```
# left right
import cv2
import numpy as np
# Read the image
img = cv2.imread('/content/sample_data/JS pp photo.jpg')
#height means all rows and
#width means all the columns
# Get the height and width of the image
height, width = img.shape[:2]
left = img[:, :width//2]
right = img[:, width//2:]
up = img[:height//2,:]
down = img[height//2:,:]
```

```
plt.figure(figsize=(10, 5))
plt.subplot(2, 1, 1)
plt.imshow(left)
plt.title("Left")
plt.axis("off")


plt.subplot(2, 1, 2)
plt.imshow(right)
plt.title("right")
plt.axis("off")


plt.show()


# Split the image into four quadrants
quad1 = img[:height//2]
quad2 = img[height//2:]
quad3 = img[:, :width//2:]
quad4 = img[:, width//2:]
plt.figure(figsize=(10, 5))
plt.subplot(1, 2, 1)
plt.imshow(quad1)
plt.title("1")
plt.axis("off")
```

```
plt.subplot(1, 2, 2)
plt.imshow(quad2)
plt.title("2")
plt.axis("off")

plt.figure(figsize=(5, 10))
plt.subplot(1, 2, 1)
plt.imshow(quad3)
plt.title("3")
plt.axis("off")

plt.subplot(1, 2, 2)
plt.imshow(quad4)
plt.title("4")
plt.axis("off")

plt.show()
```

pgm22:

```
import cv2
import numpy as np
image = cv2.imread('/content/sample_data/JS pp photo.jpg')
```

```
# Increase the brightness by adding 20 to each pixel value
brightness = 20

# Increase the contrast by scaling the pixel values by 5
contrast = 2

# Apply the brightness and contrast adjustments
image = cv2.addWeighted(image, contrast, np.zeros(image.shape, image.dtype), 0, brightness)

# Save the image
cv2.imwrite('image_brightened_and_contrasted.png', image)
```

8. Write a program to show rotation, scaling, and translation on an image.
9.
Read an image and extract and display low-level features such as edges, textures using filtering techniques.
10. Write a program to blur and smoothing an image.
11. Write a program to contour an image.
12. Write a program to detect a face/s in an image.