Deen Dayal Upadhyaya College

University of Delhi

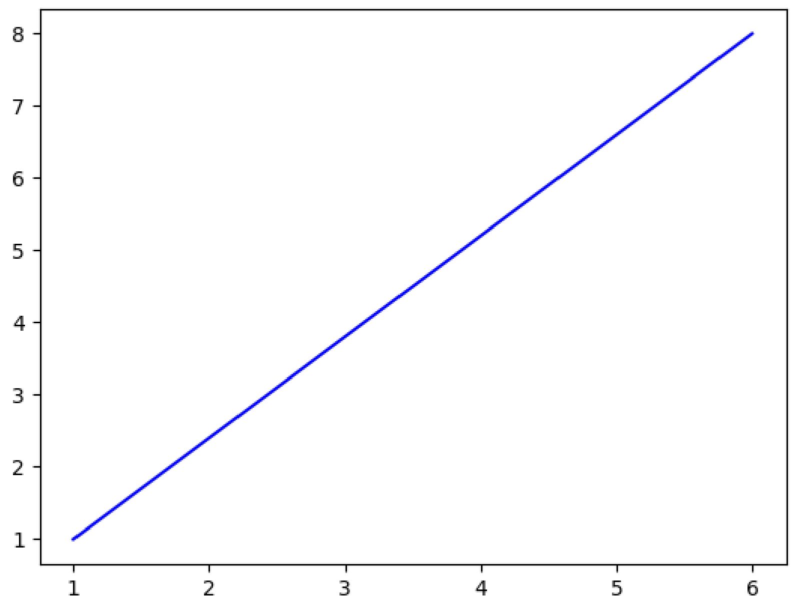
COMPUTER GRAPHICS PRACTICALS

NAME: HARSHITA SYUNARY
ROLL NO.: 21HCS4141
SEMESTER: VI
DATE OF SUBMISSION: 30-04-2024
SUBMITTED TO: PROF. RAJ KUMAR SHARMA

1. Write a program to implement DDA and Bresenham's line drawing algorithm.

```
#DDA line
import matplotlib.pyplot as plt

def dda(x1, y1, x2, y2):
    dx = x2 - x1
    dy = y2 - y1
    steps = max(abs(dx), abs(dy))
    x_inc = dx / steps
    y_inc = dy / steps
    x = x1
    y = y1
    points = []
    for i in range(int(steps) + 1):
        points.append((x, y))
        x += x_inc
        y += y_inc
    return points

# Test the DDA line drawing algorithm
x1, y1, x2, y2 = 1, 1, 6, 8
points = dda(x1, y1, x2, y2)

# Plot the line
x_values = [point[0] for point in points]
y_values = [point[1] for point in points]
plt.plot(x_values, y_values, 'b-')
plt.show()
```

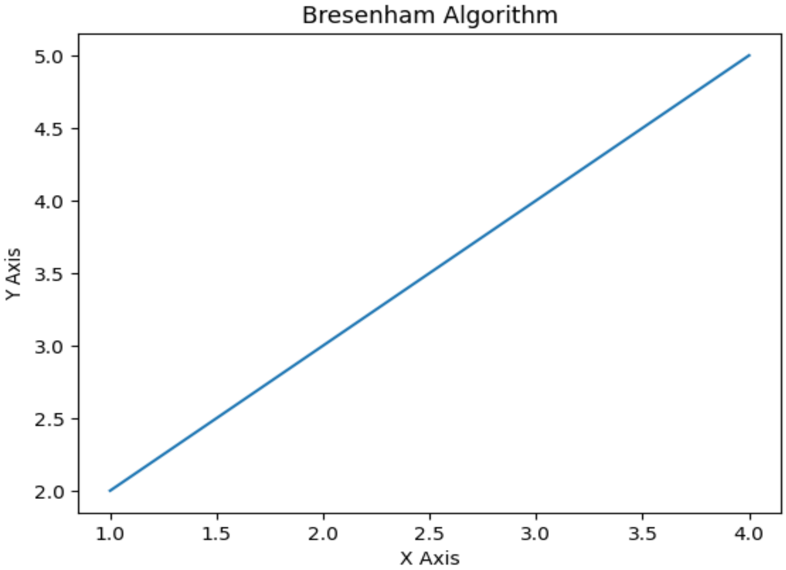
OUTPUT:



```python
#Bresenham's line
import matplotlib.pyplot as plt
plt.title("Bresenham Algorithm")
plt.xlabel("X Axis")
plt.ylabel("Y Axis")

def bres(x1,y1,x2,y2):
    x,y = x1,y1
    dx = abs(x2 - x1)
    dy = abs(y2 -y1)
    gradient = dy/float(dx)

    if gradient > 1:
        dx, dy = dy, dx
        x, y = y, x
        x1, y1 = y1, x1
        x2, y2 = y2, x2

    p = 2*dy - dx
    print(f"x = {x}, y = {y}")
    # Initialize the plotting points
    xcoordinates = [x]
    ycoordinates = [y]

    for k in range(2, dx + 2):
        if p > 0:
            y = y + 1 if y < y2 else y - 1
            p = p + 2 * (dy - dx)
        else:
            p = p + 2 * dy
        x = x + 1 if x < x2 else x - 1

        print(f"x = {x}, y = {y}")
        xcoordinates.append(x)
        ycoordinates.append(y)

    plt.plot(xcoordinates, ycoordinates)
    plt.show()

def main():
    x1 = int(input("Enter the Starting point of x: "))
    y1 = int(input("Enter the Starting point of y: "))
    x2 = int(input("Enter the end point of x: "))
    y2 = int(input("Enter the end point of y: "))

    bres(x1, y1, x2, y2)
if __name__ == "__main__":
 main()
```

OUTPUT:

```
Enter the Starting point of x: 1
Enter the Starting point of y: 2
Enter the end point of x: 4
Enter the end point of y: 5
x = 1, y = 2
x = 4, y = 5
```

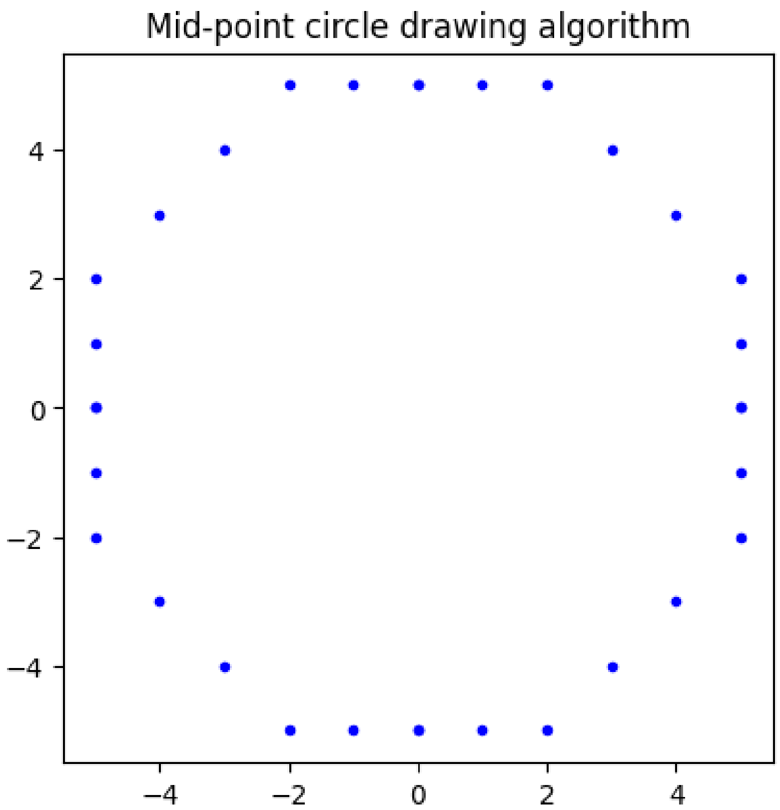


2. Write a program to implement mid-point circle drawing algorithm.

```
import matplotlib.pyplot as plt

def mid_point_circle(radius):
    x = 0
    y = radius
    p = 1 - radius

    x_points = []
    y_points = []

    while x <= y:
        x_points.extend([x, -x, x, -x, y, -y, y, -y])
        y_points.extend([y, y, -y, -y, x, x, -x, -x])
        x += 1
        if p < 0:
            p += 2 * x + 1
        else:
            y -= 1
            p += 2 * (x - y) + 1
```

```
    return x_points, y_points

# Set the radius of the circle
radius = 5

# Generate the points on the circle using the Mid-Point Circle
algorithm
x_points, y_points = mid_point_circle(radius)

# Plot the circle
plt.plot(x_points, y_points, 'b.')
plt.axis('scaled')
plt.title('Mid-point circle drawing algorithm')
plt.show()
```

OUTPUT:



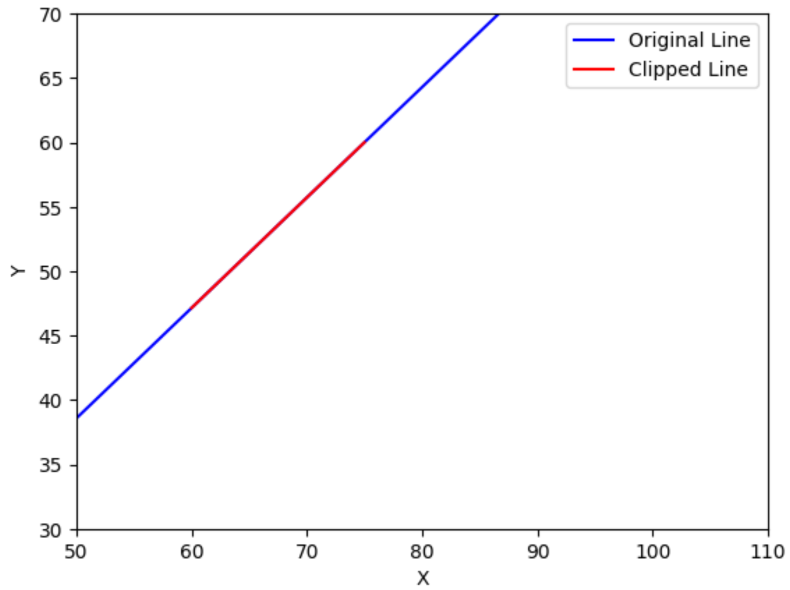
3. Write a program to clip a line using Cohen and Sutherland line clipping algorithm.

```
def cohen_sutherland(x1, y1, x2, y2, xmin, ymin, xmax, ymax):
    # Compute the region codes for the two endpoints
```

```python
    code1 = compute_region_code(x1, y1, xmin, ymin, xmax, ymax)
    code2 = compute_region_code(x2, y2, xmin, ymin, xmax, ymax)

    # The line is completely inside the clipping rectangle
    if code1 == 0 and code2 == 0:
        return x1, y1, x2, y2

    # The line is completely outside the clipping rectangle
    if code1 & code2 != 0:
        return None

    # Calculate the intersection point
    code_out = code1 if code1 != 0 else code2
    if code_out & TOP:
        x = x1 + (x2 - x1) * (ymax - y1) / (y2 - y1)
        y = ymax
    elif code_out & BOTTOM:
        x = x1 + (x2 - x1) * (ymin - y1) / (y2 - y1)
        y = ymin
    elif code_out & RIGHT:
        y = y1 + (y2 - y1) * (xmax - x1) / (x2 - x1)
        x = xmax
    elif code_out & LEFT:
        y = y1 + (y2 - y1) * (xmin - x1) / (x2 - x1)
        x = xmin

        # Update the endpoint of the line
    if code_out == code1:
        x1, y1 = x, y
    else:
        x2, y2 = x, y

    # Recursively clip the line
    return cohen_sutherland(x1, y1, x2, y2, xmin, ymin, xmax, ymax)

def compute_region_code(x, y, xmin, ymin, xmax, ymax):
    code = INSIDE
    if x < xmin:
        code |= LEFT
    elif x > xmax:
        code |= RIGHT
    if y < ymin:
        code |= BOTTOM
    elif y > ymax:
        code |= TOP
    return code

# Region codes
```

```python
INSIDE = 0  # 0000
LEFT = 1    # 0001
RIGHT = 2   # 0010
BOTTOM = 4  # 0100
TOP = 8     # 1000
import matplotlib.pyplot as plt

def plot_clipped_line(x1, y1, x2, y2, xmin, ymin, xmax, ymax):
    # Clip the line segment using Cohen-Sutherland algorithm
    clipped_points = cohen_sutherland(x1, y1, x2, y2, xmin, ymin,
xmax, ymax)

    # Plot the original line
    plt.plot([x1, x2], [y1, y2], 'b', label='Original Line')

    # Plot the clipped line
    if clipped_points is not None:
        clipped_x1, clipped_y1, clipped_x2, clipped_y2 =
clipped_points
        plt.plot([clipped_x1, clipped_x2], [clipped_y1, clipped_y2],
'r', label='Clipped Line')

    # Set the axis limits and labels
    plt.xlim([xmin-10, xmax+10])
    plt.ylim([ymin-10, ymax+10])
    plt.xlabel('X')
    plt.ylabel('Y')

    # Add legend and show the plot
    plt.legend()
    plt.show()

# Example usage
x1, y1, x2, y2 = 40, 30, 110, 90
xmin, ymin, xmax, ymax = 60, 40, 100, 60
plot_clipped_line(x1, y1, x2, y2, xmin, ymin, xmax, ymax)
```

OUTPUT:

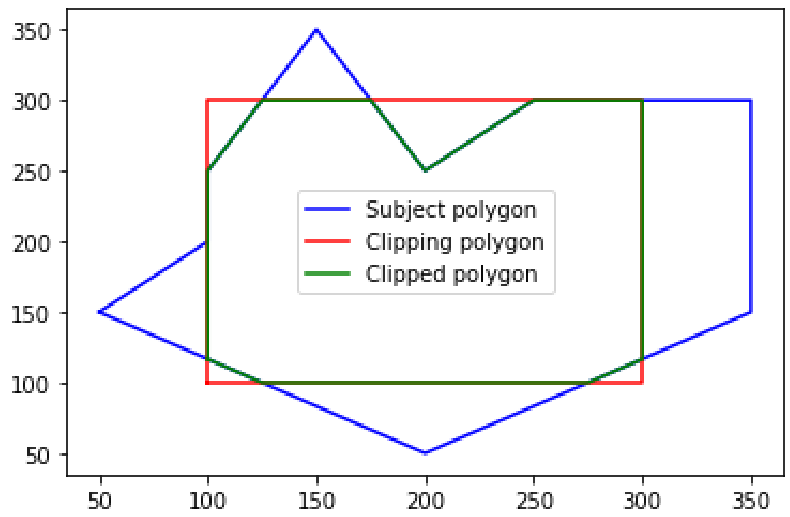


4. Write a program to clip a polygon using Sutherland Hodgeman algorithm.

```
def clip(poly_points, clip_points):
   def inside(p):
      return(cp2[0]-cp1[0])*(p[1]-cp1[1]) > (cp2[1]-cp1[1])*(p[0]-
cp1[0])

   def computeIntersection():
      dc = [ cp1[0] - cp2[0], cp1[1] - cp2[1] ]
      dp = [ s[0] - e[0], s[1] - e[1] ]
      n1 = cp1[0] * cp2[1] - cp1[1] * cp2[0]
      n2 = s[0] * e[1] - s[1] * e[0]
      n3 = 1.0 / (dc[0] * dp[1] - dc[1] * dp[0])
      return [(n1*dp[0] - n2*dc[0]) * n3, (n1*dp[1] - n2*dc[1]) * n3]

   outputList = poly_points
   cp1 = clip_points[-1]

   for clipVertex in clip_points:
      cp2 = clipVertex
      inputList = outputList
      outputList = []
      s = inputList[-1]

      for subjectVertex in inputList:
         e = subjectVertex
         if inside(e):
            if not inside(s):
               outputList.append(computeIntersection())
```

```
            outputList.append(e)
         elif inside(s):
            outputList.append(computeIntersection())
         s = e
      cp1 = cp2
   return(outputList)
subjectPolygon =
[(50,150),(200,50),(350,150),(350,300),(250,300),(200,250),(150,350),(1
00,250),(100,200)]
clipPolygon = [(100,100),(300,100),(300,300),(100,300)]

#subjectPolygon = [(100,150), (200,250), (300,100)]
#clipPolygon = [(100,100), (100,200), (200,200), (100,100)]

clippedPolygon = clip(subjectPolygon, clipPolygon)

# Plot the original polygons and the resulting clipped polygon
import matplotlib.pyplot as plt

subjectPolygon.append(subjectPolygon[0])
clipPolygon.append(clipPolygon[0])
clippedPolygon.append(clippedPolygon[0])

plt.plot([p[0] for p in subjectPolygon], [p[1] for p in
subjectPolygon], 'b-', label='Subject polygon')
plt.plot([p[0] for p in clipPolygon], [p[1] for p in clipPolygon], 'r-
', label='Clipping polygon')
plt.plot([p[0] for p in clippedPolygon], [p[1] for p in
clippedPolygon], 'g-', label='Clipped polygon')
plt.legend()
plt.show()
```

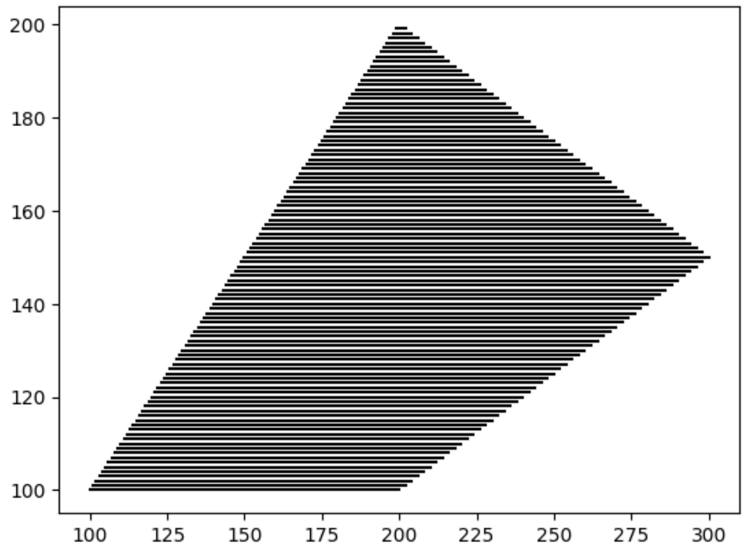
OUTPUT:



5. Write a program to fill a polygon using Scan line fill algorithm.

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

def scanline_fill(points):

    # Find the min and max y-coordinates
    ymin = int(min(points[:,1]))
    ymax = int(max(points[:,1]))

    # Initialize an array to store the x-coordinates of the
intersections
    # between the scanline and the polygon edges
    x_intersections = np.zeros((len(points),))

    # Iterate over each scanline
    for y in range(ymin, ymax+1):
        # Find the edges that intersect the scanline
        j = 0
        for i in range(len(points)):
            if i == len(points) - 1:
                k = 0
            else:
                k = i + 1

            if (points[i][1] <= y and points[k][1] > y) or
(points[k][1] <= y and points[i][1] > y):
                x_intersections[j] = int(points[i][0] + (y -
points[i][1]) / (points[k][1] - points[i][1]) * (points[k][0] -
points[i][0]))
                j += 1

        # Sort the intersections by x-coordinate
        x_intersections = np.sort(x_intersections[:j])

        # Fill the scanline between pairs of intersections
        for i in range(0, len(x_intersections), 2):
            plt.plot([x_intersections[i], x_intersections[i+1]], [y,
y], color='black')

    plt.show()

points = np.array([(100, 100), (200, 200), (300, 150), (200, 100)])
scanline_fill(points)
```

OUTPUT:

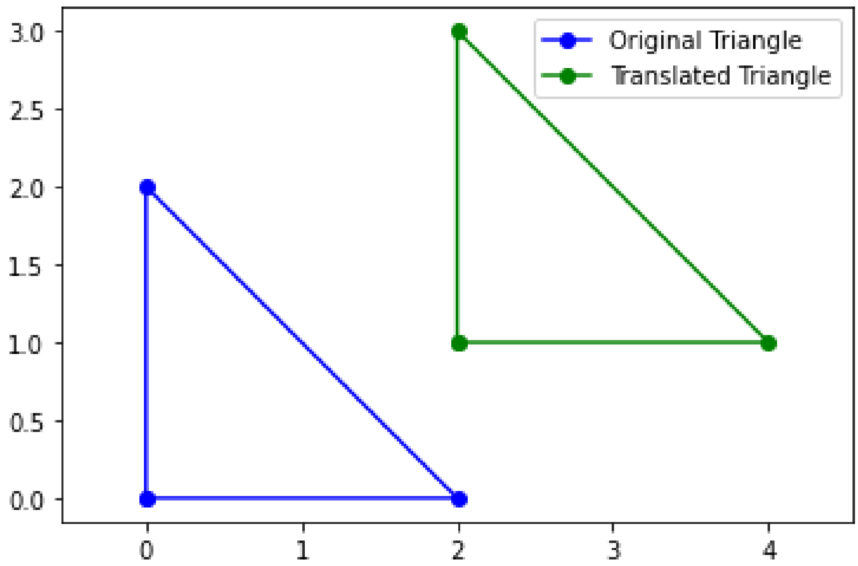


6. Write a program to apply various 2D transformations on a 2D object (use homogenous Coordinates).

CODE:

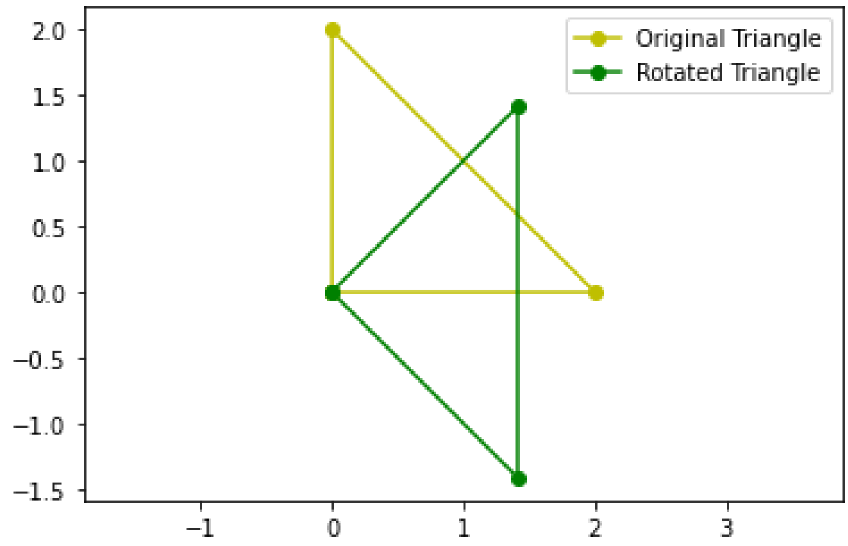
```
import numpy as np
import matplotlib.pyplot as plt

# Define the vertices of the original triangle
triangle = np.array([[0, 0], [0, 2], [2, 0], [0,0]])

# Define the translation vector
translation = np.array([2, 1])

# Translate the triangle by adding the translation vector to each vertex
new_triangle = triangle + translation

# Plot the original and translated triangles
plt.plot(triangle[:,0], triangle[:,1], 'bo-', label='Original Triangle')
plt.plot(new_triangle[:,0], new_triangle[:,1], 'go-', label='Translated Triangle')
plt.axis('equal')
plt.legend()
plt.show()
```

OUTPUT:



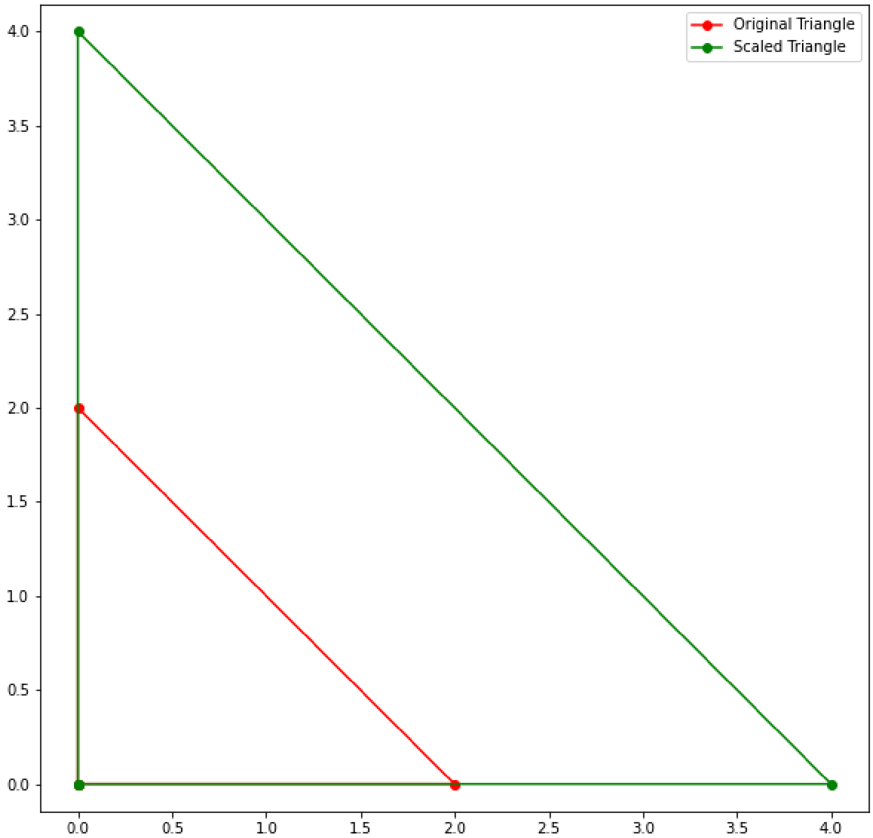

```
#ROTATION

# Define the vertices of the original triangle
triangle = np.array([[0, 0], [0, 2], [2, 0], [0,0]])

# Define the rotation angle in degrees
angle_deg = 45

# Convert the rotation angle to radians
angle_rad = np.deg2rad(angle_deg)

# Define the rotation matrix
rotation = np.array([[np.cos(angle_rad), -np.sin(angle_rad)],
                     [np.sin(angle_rad), np.cos(angle_rad)]])

# Rotate the triangle by multiplying each vertex by the rotation matrix
new_triangle = np.dot(triangle, rotation)

# Plot the original and rotated triangles
plt.plot(triangle[:,0], triangle[:,1], 'yo-', label='Original
Triangle')
plt.plot(new_triangle[:,0], new_triangle[:,1], 'go-', label='Rotated
Triangle')
plt.axis('equal')
plt.legend()
plt.show()
```

OUTPUT:



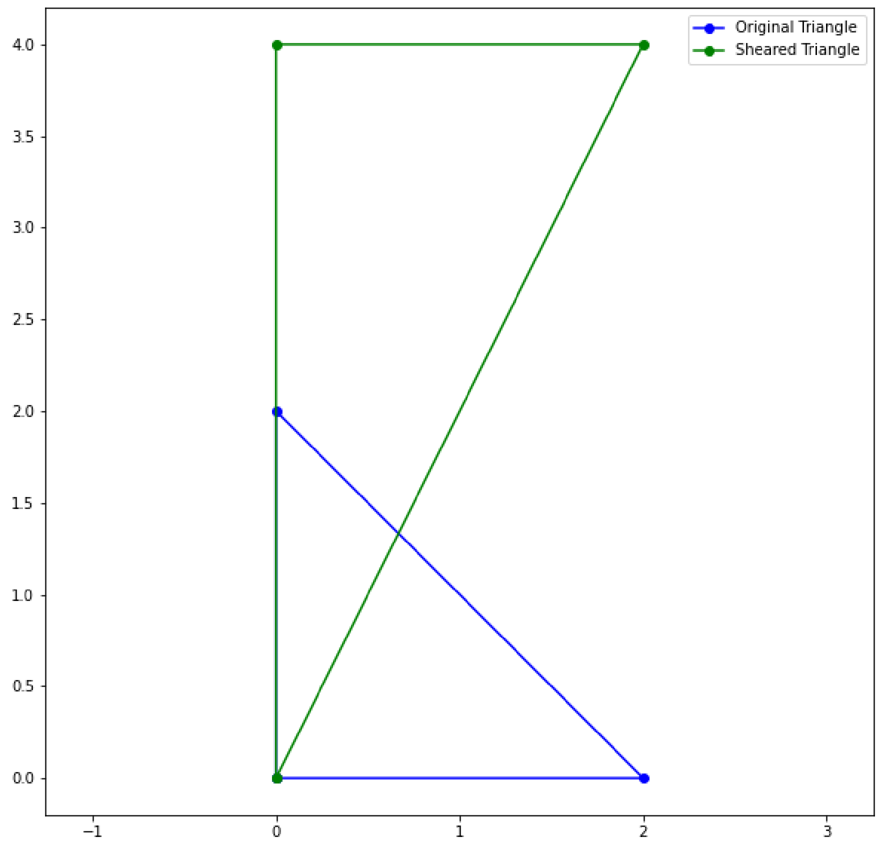

```
#SCALING

# Define the vertices of the original triangle
triangle = np.array([[0, 0], [0, 2], [2, 0], [0,0]])

# Define the scaling factor
scale_factor = 2

# Define the scaling matrix
scaling = np.array([[scale_factor, 0],
                    [0, scale_factor]])

# Scale the triangle by multiplying each vertex by the scaling matrix
new_triangle = np.dot(triangle, scaling)

# Plot the original and scaled triangles
plt.plot(triangle[:,0], triangle[:,1], 'ro-', label='Original
Triangle')
plt.plot(new_triangle[:,0], new_triangle[:,1], 'go-', label='Scaled
Triangle')
plt.axis('equal')
plt.legend()
plt.show()
```

OUTPUT:



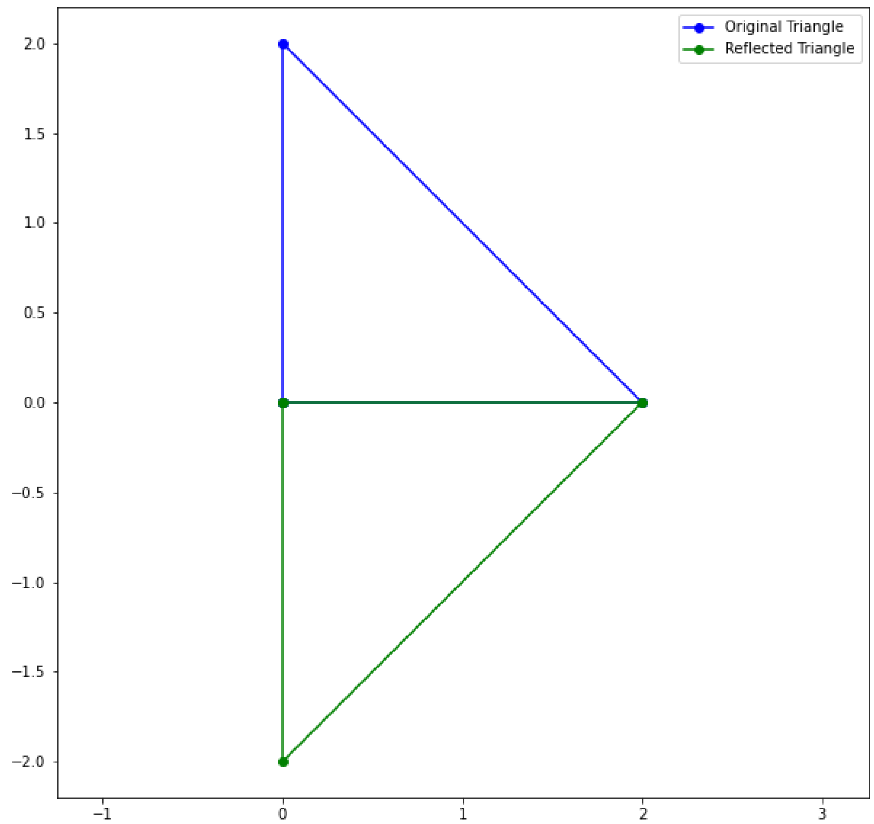

```
#SHEARING

# Define the vertices of the original triangle
triangle = np.array([[0, 0], [0, 2], [2, 0], [0,0]])

# Define the shearing factor
shear_factor = 2

# Define the shearing matrix
shearing = np.array([[1, shear_factor],
                     [0, 1]])

# Shear the triangle by multiplying each vertex by the shearing matrix
new_triangle = np.dot(triangle, shearing)

# Plot the original and sheared triangles
plt.plot(triangle[:,0], triangle[:,1], 'bo-', label='Original
Triangle')
plt.plot(new_triangle[:,0], new_triangle[:,1], 'go-', label='Sheared
Triangle')
plt.axis('equal')
plt.legend()
plt.show()
```

OUTPUT:




```
#REFLECTION

# Define the vertices of the original triangle
triangle = np.array([[0, 0], [0, 2], [2, 0], [0,0]])

# Define the reflection axis
reflection_axis = np.array([[1, 0], [0, -1]])

# Reflect the triangle by multiplying each vertex by the reflection
axis
new_triangle = np.dot(triangle, reflection_axis)

# Plot the original and reflected triangles
plt.plot(triangle[:,0], triangle[:,1], 'bo-', label='Original
Triangle')
plt.plot(new_triangle[:,0], new_triangle[:,1], 'go-', label='Reflected
Triangle')
plt.axis('equal')
plt.legend()
plt.show()
```

OUTPUT:



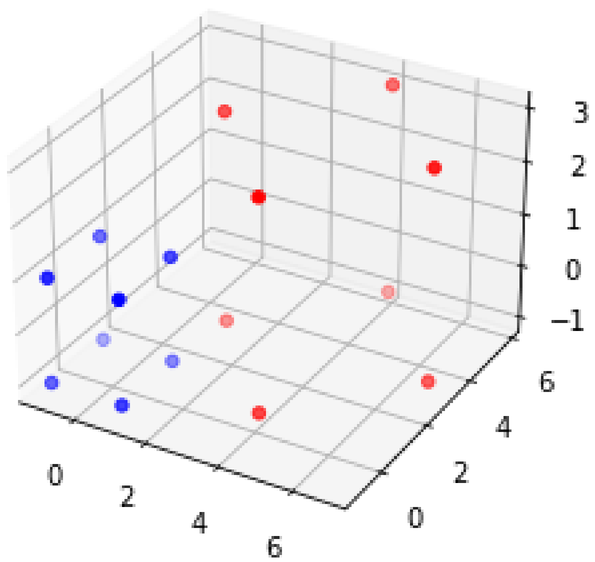
7. Write a program to apply various 3D transformations on a 3D object and then apply parallel and perspective projection on it.

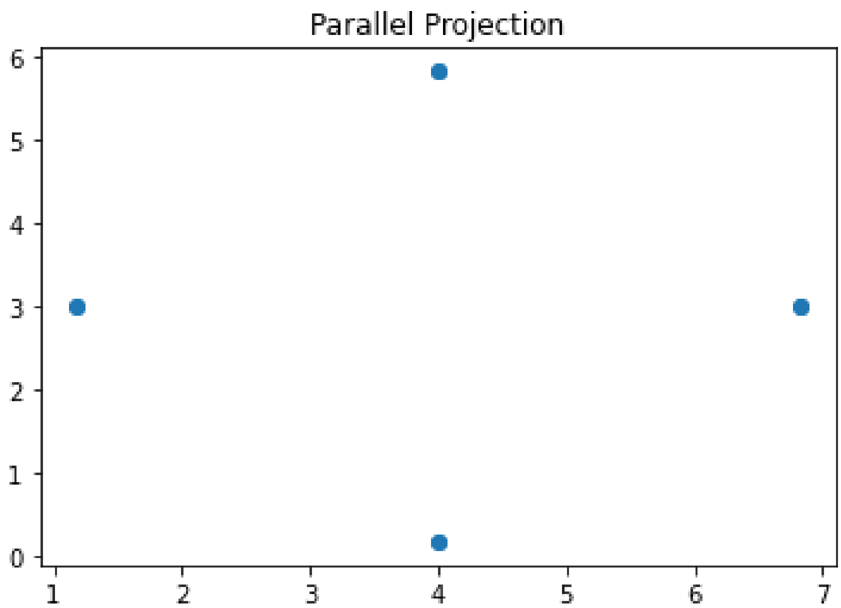
```
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

# Define the 3D cube vertices
cube_vertices = np.array([
    [-1, -1, -1],
    [1, -1, -1],
    [1, 1, -1],
    [-1, 1, -1],
    [-1, -1, 1],
    [1, -1, 1],
    [1, 1, 1],
    [-1, 1, 1]
])

# Define the transformation matrices
scaling_matrix = np.array([
    [2, 0, 0],
    [0, 2, 0],
    [0, 0, 2]
])
```

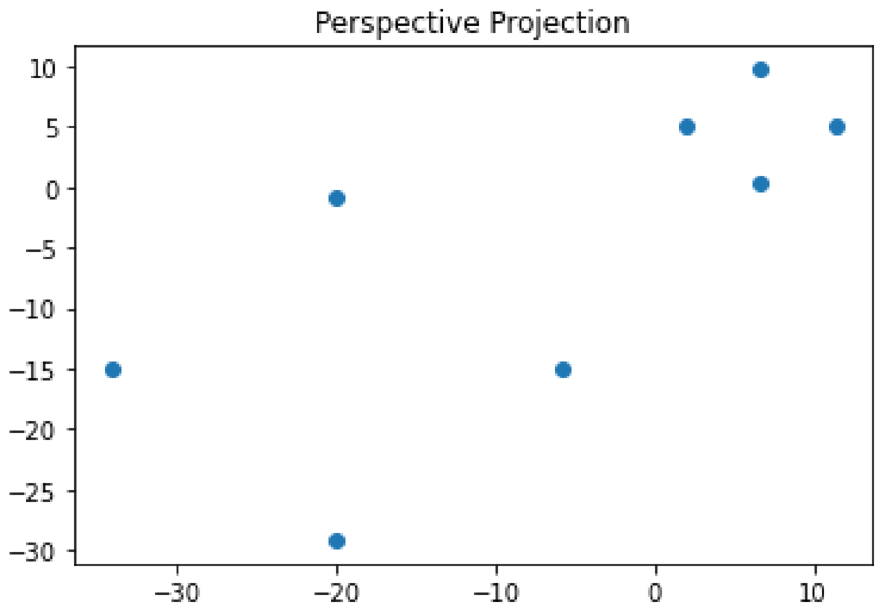
```
rotation_matrix = np.array([
    [np.cos(np.pi/4), -np.sin(np.pi/4), 0],
    [np.sin(np.pi/4), np.cos(np.pi/4), 0],
    [0, 0, 1]
])

translation_matrix = np.array([
    [1, 0, 0, 4],
    [0, 1, 0, 3],
    [0, 0, 1, 1],
    [0, 0, 0, 1]
])

# Apply the transformations to the cube vertices
transformed_cube_vertices =
cube_vertices.dot(scaling_matrix).dot(rotation_matrix) +
translation_matrix[:3,3]

# Perform parallel projection
parallel_projection_matrix = np.array([
    [1, 0, 0],
    [0, 1, 0]
])
projected_cube_vertices = transformed_cube_vertices[:,
:2].dot(parallel_projection_matrix)

# Perform perspective projection
focal_length = 5
perspective_projection_matrix = np.array([
    [focal_length, 0, 0],
    [0, focal_length, 0]
])
projected_cube_vertices_perspective = transformed_cube_vertices[:,
:2].dot(perspective_projection_matrix) /
transformed_cube_vertices[:, 2:]

# Plot the original and transformed 3D cube
fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')

ax.scatter(cube_vertices[:, 0], cube_vertices[:, 1],
cube_vertices[:, 2], color='blue')
ax.scatter(transformed_cube_vertices[:, 0],
transformed_cube_vertices[:, 1], transformed_cube_vertices[:, 2],
color='red')
plt.show()
```

```
# Plot the parallel projection
plt.scatter(projected_cube_vertices[:, 0],
projected_cube_vertices[:, 1])
plt.title('Parallel Projection')
plt.show()

# Plot the perspective projection
plt.scatter(projected_cube_vertices_perspective[:, 0],
projected_cube_vertices_perspective[:, 1])
plt.title('Perspective Projection')
plt.show()
```

OUTPUT:





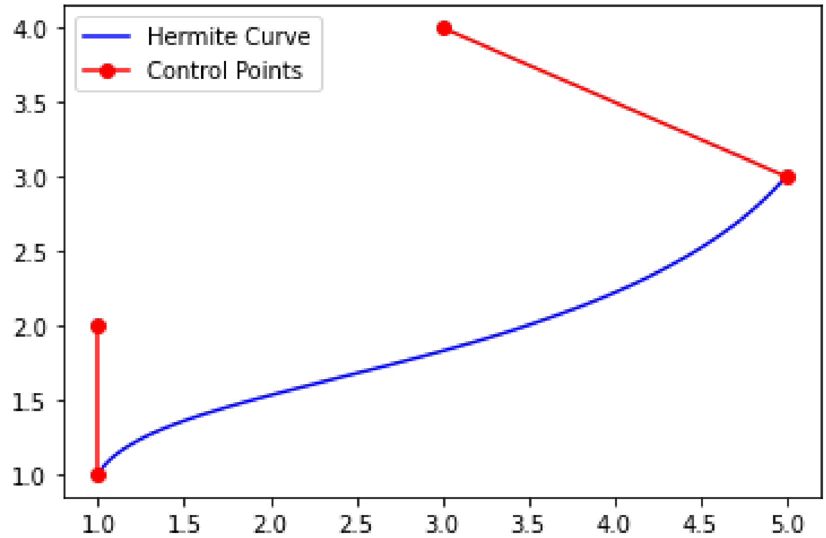


8. Write a program to draw Hermite /Bezier curve.

```
#Hermite curve
import numpy as np
import matplotlib.pyplot as plt

def hermite_curve(P0, P1, P2, P3):
    #
    def H1(t):
        return 2*t**3 - 3*t**2 + 1

    def H2(t):
        return -2*t**3 + 3*t**2

    def H3(t):
        return t**3 - 2*t**2 + t

    def H4(t):
        return t**3 - t**2

    # Create a range of values for t
    t_values = np.linspace(0.0, 1.0, 100)

    # Evaluate the Hermite curve function for each value of t
    curve_points = np.array([P0 * H1(t) + P3 * H2(t) + P1 * H3(t) + 
P2 * H4(t)
                             for t in t_values])

    # Plot the Hermite curve
    plt.plot(curve_points[:,0], curve_points[:,1], 'b-', 
label='Hermite Curve')
```

```
    plt.plot([P0[0], P1[0]], [P0[1], P1[1]], 'ro-', label='Control
Points')
    plt.plot([P2[0], P3[0]], [P2[1], P3[1]], 'ro-')
    plt.legend()
    plt.show()

# Example usage:
P0 = np.array([1, 1])
P1 = np.array([1, 2])
P2 = np.array([3, 4])
P3 = np.array([5, 3])
hermite_curve(P0, P1, P2, P3)
```

OUTPUT:




```
#Bezier curve
import matplotlib.pyplot as plt
import numpy as np
import math
def bezier_curve(control_points):

    control_points=np.array(control_points)

    def B(t):
        n = len(control_points)-1
        return np.sum([control_points[i] * math.comb(n,i) * (t**i) *(1-
t)**(n-i)
                          for i in range(n+1)], axis=0)


    # Create a range of values for t
    t_values = np.linspace(0.0, 1.0, 100)
```

```
    # Evaluate the Bezier curve function for each value of t
    curve_points = np.array([B(t) for t in t_values])

    # Plot the Bezier curve
    plt.plot(curve_points[:,0], curve_points[:,1], 'b-', label='Bezier
Curve')

    plt.plot(control_points[:,0], control_points[:,1], 'ro-',
label='Control Points')
    plt.legend()
    plt.show()

# Example usage:
control_points = [(1,1), (2,3), (4,4), (6,1)]
bezier_curve(control_points)
```

OUTPUT: