**Deen Dayal Upadhyaya College**

**University of Delhi**



**COMPUTER GRAPHICS PRACTICALS**

Index

1. DDA and Bresenham’s Line Drawing Algorithm Page 3-5

2. Mid-Point Circle Drawing Algorithm Page 6-7

3. Cohen and Sutherland Line Clipping Algorithm Page 8-10

4. Sutherland Hodgeman Polygon Clipping Algorithm Page 11-12

5. Scan Line Fill Algorithm for Polygon Filling Page 13-14

6. 2D Object Transformations using Homogeneous Coordinates Page 15-17

7. 3D Object Transformations with Parallel and Perspective Projection Page 18-21

8. Hermite / Bezier Curve Drawing Page 22-24Q1. Write a program to implement DDA and Bresenham’s line drawing algorithm.

**Code:**

**import matplotlib.pyplot as plt**

**def draw\_line\_DDA(x1, y1, x2, y2):**

**dx = x2 - x1**

**dy = y2 - y1**

**steps = max(abs(dx), abs(dy))**

**x\_increment = dx / steps**

**y\_increment = dy / steps**

**x = x1**

**y = y1**

**plt.plot(x, y, 'ro')**

**print("Intermediate points using DDA:")**

**for \_ in range(steps):**

**x += x\_increment**

**y += y\_increment**

**plt.plot(round(x), round(y), 'ro')**

**print(f"({round(x)}, {round(y)})")**

**plt.plot([x1, x2], [y1, y2], 'b')**

**def draw\_line\_Bresenham(x1, y1, x2, y2):**

**dx = abs(x2 - x1)**

**dy = abs(y2 - y1)**

**p = 2 \* dy - dx**

**x = x1**

**y = y1**

**plt.plot(x, y, 'ro')**

**print("Intermediate points using Bresenham:")**

**if x1 < x2:**

**x\_increment = 1**

**else:**

**x\_increment = -1**

**if y1 < y2:**

**y\_increment = 1**

**else:**

**y\_increment = -1**

**while x != x2:**

**x += x\_increment**

**if p < 0:**

**p += 2 \* dy**

**else:**

**y += y\_increment**

**p += 2 \* (dy - dx)**

**plt.plot(x, y, 'ro')**

**print(f"({x}, {y})")**

**plt.plot([x1, x2], [y1, y2], 'g')**

**# Test the functions**

**x1, y1 = 9, 18**

**x2, y2 = 14, 22**

**plt.figure(figsize=(8, 6))**

**# Plotting using DDA**

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

**plt.title('DDA Algorithm')**

**draw\_line\_DDA(x1, y1, x2, y2)**

**plt.grid(True)**

**plt.axis('equal')**

**# Plotting using Bresenham's**

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

**plt.title('Bresenham Algorithm')**

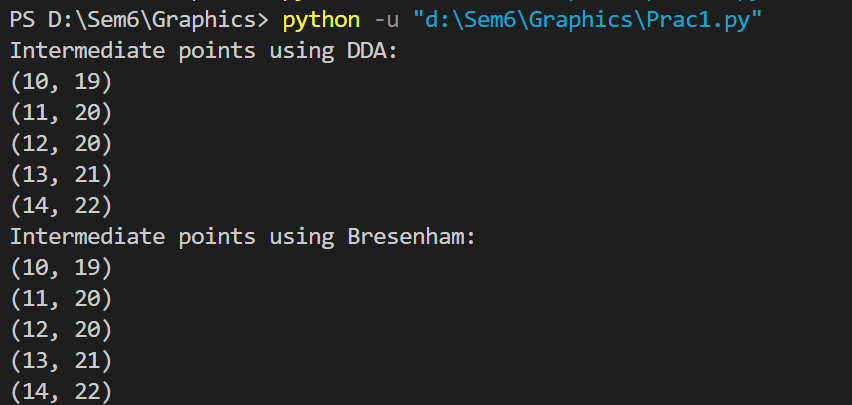
**draw\_line\_Bresenham(x1, y1, x2, y2)**

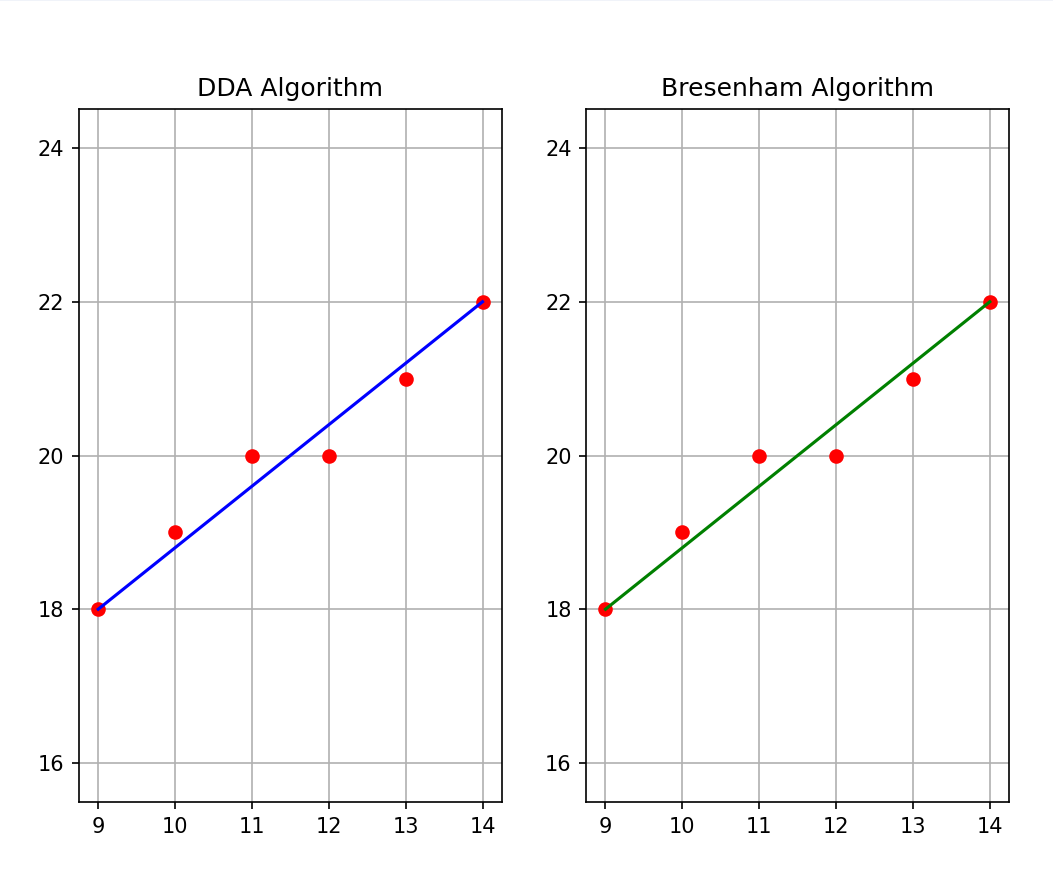
**plt.grid(True)**

**plt.axis('equal')**

**plt.show()**

**Output:**

****

****

Q2. Write a program to implement mid-point circle drawing algorithm.

**Code:**

**import matplotlib.pyplot as plt**

**def draw\_circle\_midpoint(radius):**

**x = 0**

**y = radius**

**p = 1 - radius # Initial decision parameter**

**points = set()**

**draw\_circle\_points(x, y, points)**

**print("Intermediate points:")**

**while x < y:**

**x += 1**

**if p < 0:**

**p += 2 \* x + 1**

**else:**

**y -= 1**

**p += 2 \* (x - y) + 1**

**draw\_circle\_points(x, y, points)**

**print(f"({x}, {y}), ({-x}, {y}), ({x}, {-y}), ({-x}, {-y}), ({y}, {x}), ({-y}, {x}), ({y}, {-x}), ({-y}, {-x})")**

**plot\_points(points)**

**def draw\_circle\_points(x, y, points):**

**points.add((x, y))**

**points.add((-x, y))**

**points.add((x, -y))**

**points.add((-x, -y))**

**points.add((y, x))**

**points.add((-y, x))**

**points.add((y, -x))**

**points.add((-y, -x))**

**def plot\_points(points):**

**x\_values = [point[0] for point in points]**

**y\_values = [point[1] for point in points]**

**plt.plot(x\_values, y\_values, 'ro')**

**plt.grid(True)**

**plt.axis('equal')**

**plt.title('Mid-Point Circle Drawing Algorithm')**

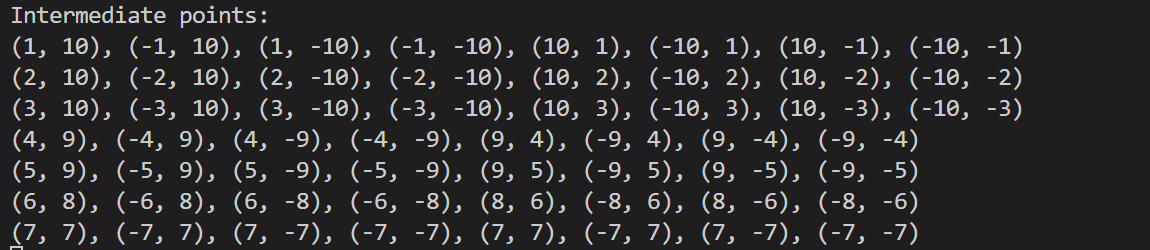
**plt.show()**

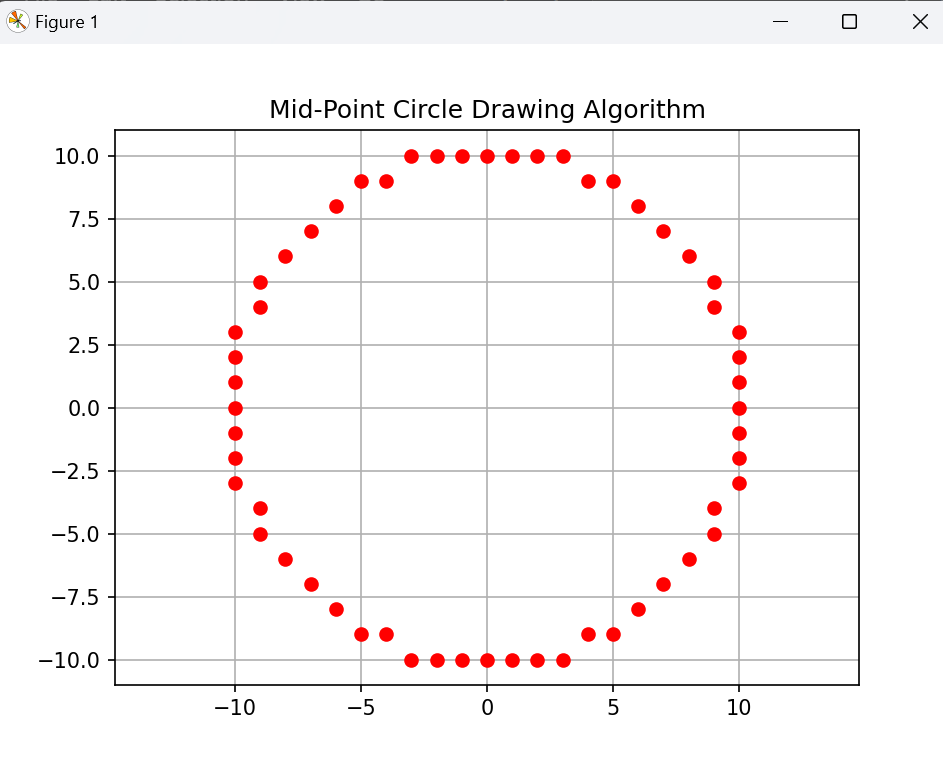
**# function**

**radius = 10**

**draw\_circle\_midpoint(radius)**

**Output:**

****

****

Q3) Write a program to clip a line using Cohen and Sutherland line clipping algorithm.

**Code:**

**import matplotlib.pyplot as plt**

**INSIDE = 0 # 0000**

**LEFT = 1 # 0001**

**RIGHT = 2 # 0010**

**BOTTOM = 4 # 0100**

**TOP = 8 # 1000**

**def computeCode(x, y):**

**code = INSIDE**

**if x < x\_min: # left of rectangle**

**code |= LEFT**

**elif x > x\_max: # right of rectangle**

**code |= RIGHT**

**if y < y\_min: # below**

**code |= BOTTOM**

**elif y > y\_max: # above**

**code |= TOP**

**return code**

**def cohenSutherlandClip(x1, y1, x2, y2):**

**code1 = computeCode(x1, y1)**

**code2 = computeCode(x2, y2)**

**accept = False**

**while True:**

**if code1 == 0 and code2 == 0:**

**accept = True**

**break**

**elif (code1 & code2) != 0:**

**break**

**else:**

**x = 1.0**

**y = 1.0**

**if code1 != 0:**

**code\_out = code1**

**else:**

**code\_out = code2**

**if code\_out & TOP:**

**x = x1 + (x2 - x1) \* (y\_max - y1) / (y2 - y1)**

**y = y\_max**

**elif code\_out & BOTTOM:**

**x = x1 + (x2 - x1) \* (y\_min - y1) / (y2 - y1)**

**y = y\_min**

**elif code\_out & RIGHT:**

**y = y1 + (y2 - y1) \* (x\_max - x1) / (x2 - x1)**

**x = x\_max**

**elif code\_out & LEFT:**

**y = y1 + (y2 - y1) \* (x\_min - x1) / (x2 - x1)**

**x = x\_min**

**if code\_out == code1:**

**x1 = x**

**y1 = y**

**code1 = computeCode(x1, y1)**

**else:**

**x2 = x**

**y2 = y**

**code2 = computeCode(x2, y2)**

**if accept:**

**print(f"Line accepted from {x1:.2f}, {y1:.2f} to {x2:.2f}, {y2:.2f}")**

**plt.plot([x1, x2], [y1, y2], color='black', linewidth=2) # Plot the accepted line**

**else:**

**print("Line rejected")**

**x\_max = 10.0**

**y\_max = 8.0**

**x\_min = 4.0**

**y\_min = 4.0**

**# Plot the clipping window**

**plt.plot([x\_min, x\_max, x\_max, x\_min, x\_min], [y\_min, y\_min, y\_max, y\_max, y\_min], color='blue', linewidth=2)**

**# Test lines**

**cohenSutherlandClip(3, 5, 5, 7)**

**plt.plot([3, 5], [5, 7], color='green', linewidth=1)**

**cohenSutherlandClip(8, 2, 8, 6)**

**plt.plot([8, 8], [2, 6], color='green', linewidth=1)**

**cohenSutherlandClip(1, 5, 4, 1)**

**plt.plot([1, 5], [4, 1], color='green', linewidth=1)**

**# Set plot limits and aspect ratio**

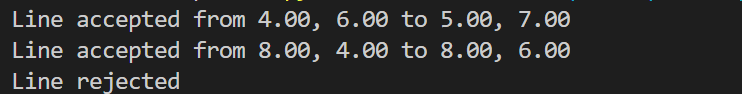
**plt.xlim(0, 12)**

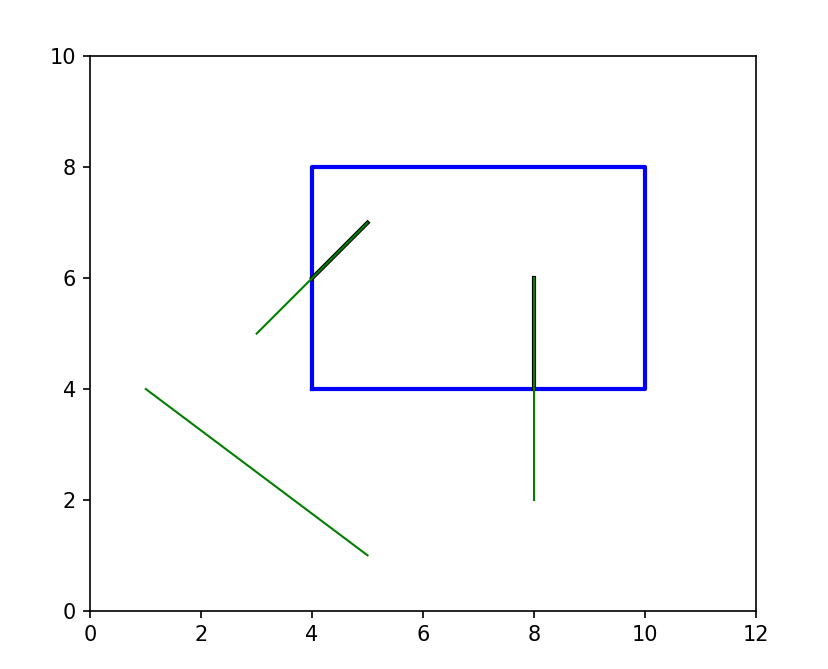
**plt.ylim(0, 10)**

**plt.gca().set\_aspect('equal', adjustable='box')**

**plt.show()**

**Output:**

****



Q4. Write a program to clip a polygon using Sutherland Hodgeman algorithm.

**Code:**

**def clip(subjectPolygon, clipPolygon):**

**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 = subjectPolygon**

**cp1 = clipPolygon[-1]**

**for clipVertex in clipPolygon:**

**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)**

**# Example usage:**

**subjectPolygon = [(50, 50), (200, 50), (200, 150), (50, 150)]**

**clipPolygon = [(100, 100), (300, 100), (300, 250), (100, 250)]**

**clippedPolygon = clip(subjectPolygon, clipPolygon)**

**# Plotting**

**import matplotlib.pyplot as plt**

**subject\_x = [point[0] for point in subjectPolygon]**

**subject\_y = [point[1] for point in subjectPolygon]**

**clip\_x = [point[0] for point in clipPolygon]**

**clip\_y = [point[1] for point in clipPolygon]**

**clipped\_x = [point[0] for point in clippedPolygon]**

**clipped\_y = [point[1] for point in clippedPolygon]**

**plt.plot(subject\_x + [subject\_x[0]], subject\_y + [subject\_y[0]], 'r-', label='Subject Polygon')**

**plt.plot(clip\_x + [clip\_x[0]], clip\_y + [clip\_y[0]], 'b-', label='Clip Polygon')**

**plt.plot(clipped\_x + [clipped\_x[0]], clipped\_y + [clipped\_y[0]], 'g-', label='Clipped Polygon')**

**plt.fill(clipped\_x, clipped\_y, color='green', alpha=0.5) # Fill the clipped polygon**

**plt.legend()**

**plt.xlabel('X')**

**plt.ylabel('Y')**

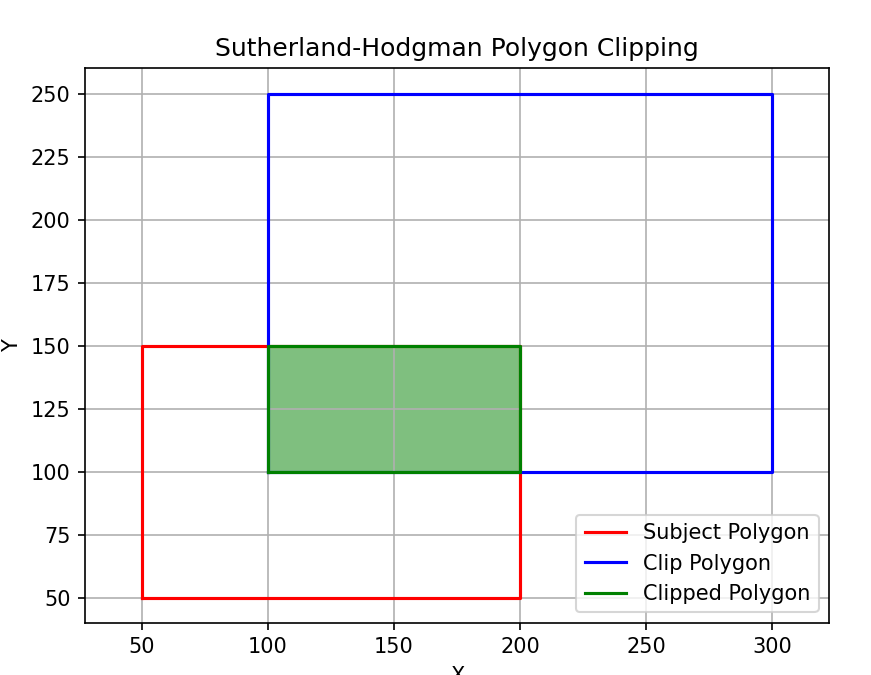
**plt.title('Sutherland-Hodgman Polygon Clipping')**

**plt.axis('equal')**

**plt.grid(True)**

**plt.show()**

**Output:**

****

Q5. Write a program to fill a polygon using Scan line fill algorithm.

**Code:**

**import matplotlib.pyplot as plt**

**import numpy as np**

**def edge\_table(vertices):**

**edges = []**

**ymin = min(vertices, key=lambda x: x[1])[1]**

**ymax = max(vertices, key=lambda x: x[1])[1]**

**for i in range(len(vertices)):**

**x1, y1 = vertices[i]**

**x2, y2 = vertices[(i + 1) % len(vertices)]**

**if y1 != y2:**

**if y1 < y2:**

**edges.append((x1, y1, x2, y2))**

**else:**

**edges.append((x2, y2, x1, y1))**

**return edges, ymin, ymax**

**def intersect\_x(edge, y):**

**x1, y1, x2, y2 = edge**

**if y1 == y2:**

**return x1**

**return x1 + (y - y1) \* (x2 - x1) / (y2 - y1)**

**def fill\_polygon(vertices):**

**edges, ymin, ymax = edge\_table(vertices)**

**active\_edges = []**

**scanline\_points = {}**

**for y in range(ymin, ymax + 1):**

**for edge in edges:**

**x1, y1, x2, y2 = edge**

**if y1 <= y < y2 or y2 <= y < y1:**

**x\_int = intersect\_x(edge, y)**

**if y not in scanline\_points:**

**scanline\_points[y] = []**

**scanline\_points[y].append(x\_int)**

**if y in scanline\_points:**

**active\_edges.extend(scanline\_points[y])**

**active\_edges.sort()**

**for i in range(0, len(active\_edges), 2):**

**x1 = int(active\_edges[i])**

**x2 = int(active\_edges[i + 1])**

**plt.plot(range(x1, x2+1), [y] \* (x2 - x1 + 1), color='r')**

**active\_edges = [x for x in active\_edges if x not in scanline\_points[y]]**

**# Example**

**vertices = [(50, 50), (100, 100), (150, 50), (100, 25),(50,50)]**

**plt.figure()**

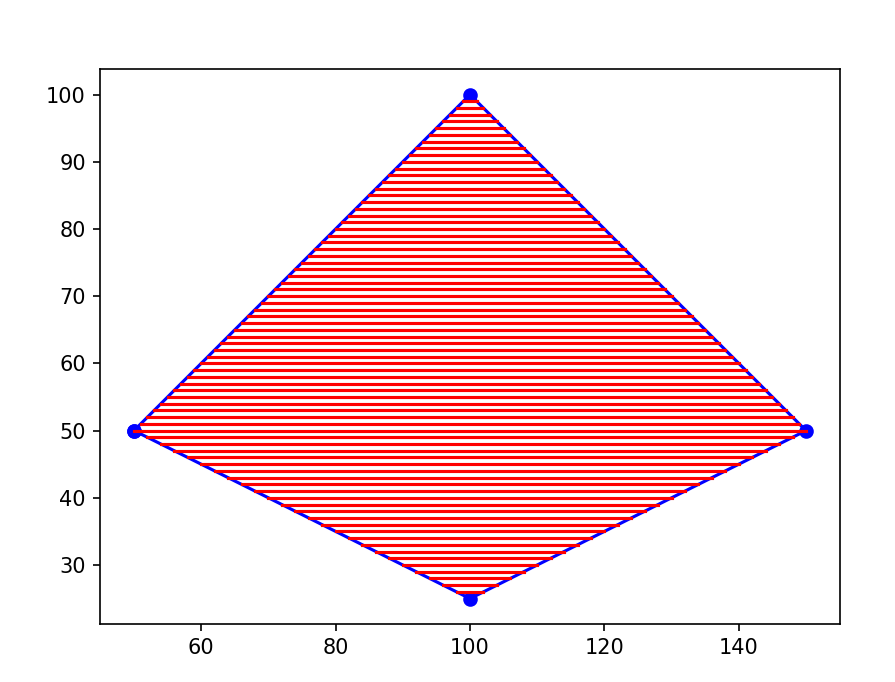
**plt.gca().set\_aspect('equal', adjustable='box')**

**plt.plot(\*zip(\*vertices), marker='o', color='b')**

**fill\_polygon(vertices)**

**plt.show()**

**Output:**

****

Q6. 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**

**def translate(tx, ty):**

**return np.array([**

**[1, 0, tx],**

**[0, 1, ty],**

**[0, 0, 1]**

**])**

**def rotate(theta):**

**cos\_theta = np.cos(theta)**

**sin\_theta = np.sin(theta)**

**return np.array([**

**[cos\_theta, -sin\_theta, 0],**

**[sin\_theta, cos\_theta, 0],**

**[0, 0, 1]**

**])**

**def scale(sx, sy):**

**return np.array([**

**[sx, 0, 0],**

**[0, sy, 0],**

**[0, 0, 1]**

**])**

**def shear(kx, ky):**

**return np.array([**

**[1, kx, 0],**

**[ky, 1, 0],**

**[0, 0, 1]**

**])**

**# Define a square**

**square = np.array([**

**[0, 0, 1],**

**[1, 0, 1],**

**[1, 1, 1],**

**[0, 1, 1],**

**[0, 0, 1]**

**])**

**# Apply transformations**

**translation\_matrix = translate(2, 2)**

**rotation\_matrix = rotate(np.pi/4) # 45 degrees**

**scaling\_matrix = scale(1.5, 1)**

**shearing\_matrix = shear(0.5, 0)**

**translated\_square = np.dot(square, translation\_matrix.T)**

**rotated\_square = np.dot(square, rotation\_matrix.T)**

**scaled\_square = np.dot(square, scaling\_matrix.T)**

**sheared\_square = np.dot(square, shearing\_matrix.T)**

**# Get the maximum and minimum values of x and y coordinates**

**x\_min = min(np.min(square[:, 0]), np.min(translated\_square[:, 0]), np.min(rotated\_square[:, 0]), np.min(scaled\_square[:, 0]), np.min(sheared\_square[:, 0]))**

**x\_max = max(np.max(square[:, 0]), np.max(translated\_square[:, 0]), np.max(rotated\_square[:, 0]), np.max(scaled\_square[:, 0]), np.max(sheared\_square[:, 0]))**

**y\_min = min(np.min(square[:, 1]), np.min(translated\_square[:, 1]), np.min(rotated\_square[:, 1]), np.min(scaled\_square[:, 1]), np.min(sheared\_square[:, 1]))**

**y\_max = max(np.max(square[:, 1]), np.max(translated\_square[:, 1]), np.max(rotated\_square[:, 1]), np.max(scaled\_square[:, 1]), np.max(sheared\_square[:, 1]))**

**# Plot original and transformed squares**

**plt.figure(figsize=(12, 6))**

**plt.subplot(1, 5, 1)**

**plt.plot(square[:, 0], square[:, 1], '-o')**

**plt.xlim(x\_min, x\_max)**

**plt.ylim(y\_min, y\_max)**

**plt.title('Original Square')**

**plt.subplot(1, 5, 2)**

**plt.plot(translated\_square[:, 0], translated\_square[:, 1], '-o')**

**plt.xlim(x\_min, x\_max)**

**plt.ylim(y\_min, y\_max)**

**plt.title('Translated Square')**

**plt.subplot(1, 5, 3)**

**plt.plot(rotated\_square[:, 0], rotated\_square[:, 1], '-o')**

**plt.xlim(x\_min, x\_max)**

**plt.ylim(y\_min, y\_max)**

**plt.title('Rotated Square')**

**plt.subplot(1, 5, 4)**

**plt.plot(scaled\_square[:, 0], scaled\_square[:, 1], '-o')**

**plt.xlim(x\_min, x\_max)**

**plt.ylim(y\_min, y\_max)**

**plt.title('Scaled Square')**

**plt.subplot(1, 5, 5)**

**plt.plot(sheared\_square[:, 0], sheared\_square[:, 1], '-o')**

**plt.xlim(x\_min, x\_max)**

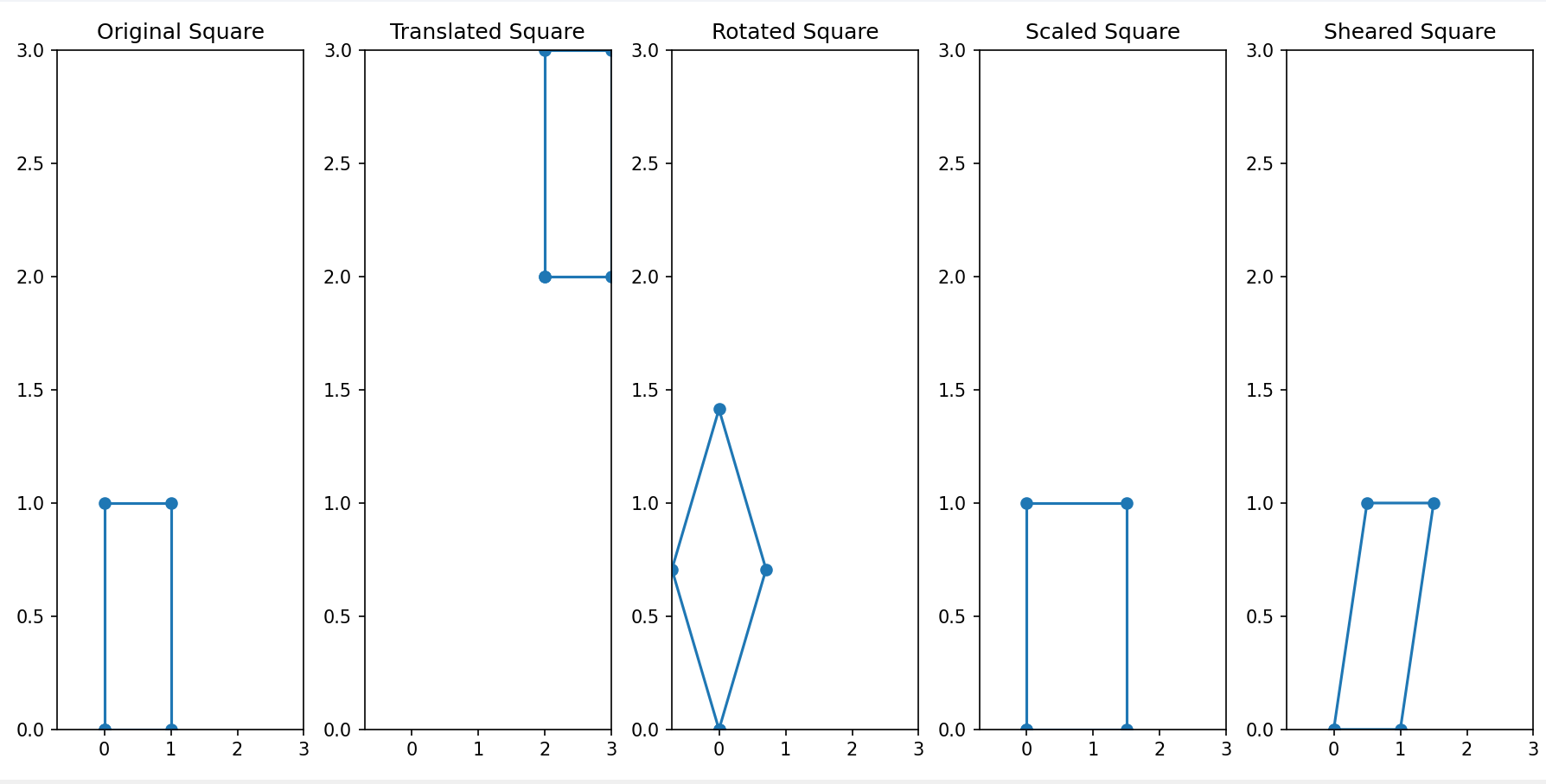
**plt.ylim(y\_min, y\_max)**

**plt.title('Sheared Square')**

**plt.tight\_layout()**

**plt.show()**

**Ouput:**

****

Q7. Write a program to apply various 3D transformations on a 3D object and then apply parallel and perspective projection on it.

**Code:**

**import numpy as np**

**import matplotlib.pyplot as plt**

**from mpl\_toolkits.mplot3d import Axes3D**

**# Define a cube**

**cube\_vertices = np.array([**

**[0, 0, 0, 1],**

**[1, 0, 0, 1],**

**[1, 1, 0, 1],**

**[0, 1, 0, 1],**

**[0, 0, 1, 1],**

**[1, 0, 1, 1],**

**[1, 1, 1, 1],**

**[0, 1, 1, 1]**

**])**

**# Define edges to connect vertices of the cube**

**cube\_edges = [**

**(0, 1), (1, 2), (2, 3), (3, 0), # Bottom face**

**(4, 5), (5, 6), (6, 7), (7, 4), # Top face**

**(0, 4), (1, 5), (2, 6), (3, 7) # Connecting edges**

**]**

**# Function to plot the cube**

**def plot\_cube(vertices, edges, ax):**

**for edge in edges:**

**start = vertices[edge[0]]**

**end = vertices[edge[1]]**

**ax.plot3D([start[0], end[0]], [start[1], end[1]], [start[2], end[2]], color='blue')**

**# Function to apply transformations (translation, rotation, scaling)**

**def transform(vertices, transformation\_matrix):**

**transformed\_vertices = np.dot(vertices, transformation\_matrix.T)**

**return transformed\_vertices**

**# Apply transformations**

**translation\_matrix = np.array([[1, 0, 0, 1],**

**[0, 1, 0, 1],**

**[0, 0, 1, 1],**

**[0, 0, 0, 1]])**

**rotation\_matrix\_x = np.array([[1, 0, 0, 0],**

**[0, np.cos(np.pi/4), -np.sin(np.pi/4), 0],**

**[0, np.sin(np.pi/4), np.cos(np.pi/4), 0],**

**[0, 0, 0, 1]])**

**rotation\_matrix\_y = np.array([[np.cos(np.pi/4), 0, np.sin(np.pi/4), 0],**

**[0, 1, 0, 0],**

**[-np.sin(np.pi/4), 0, np.cos(np.pi/4), 0],**

**[0, 0, 0, 1]])**

**rotation\_matrix\_z = np.array([[np.cos(np.pi/4), -np.sin(np.pi/4), 0, 0],**

**[np.sin(np.pi/4), np.cos(np.pi/4), 0, 0],**

**[0, 0, 1, 0],**

**[0, 0, 0, 1]])**

**scaling\_matrix = np.array([[2, 0, 0, 0],**

**[0, 1.5, 0, 0],**

**[0, 0, 0.5, 0],**

**[0, 0, 0, 1]])**

**parallel\_projection\_matrix = np.array([**

**[1, 0, 0, 0],**

**[0, 1, 0, 0],**

**[0, 0, 0, 0],**

**[0, 0, 0, 1]**

**])**

**perspective\_projection\_matrix = np.array([**

**[1, 0, 0, 0],**

**[0, 1, 0, 0],**

**[0, 0, 0, -0.001],**

**[0, 0, 0, 1]**

**])**

**translated\_cube = transform(cube\_vertices, translation\_matrix)**

**rotated\_x\_cube = transform(cube\_vertices, rotation\_matrix\_x)**

**rotated\_y\_cube = transform(cube\_vertices, rotation\_matrix\_y)**

**rotated\_z\_cube = transform(cube\_vertices, rotation\_matrix\_z)**

**scaled\_cube = transform(cube\_vertices, scaling\_matrix)**

**parallel\_cube = transform(cube\_vertices, parallel\_projection\_matrix)**

**perspective\_cube = transform(cube\_vertices, perspective\_projection\_matrix)**

**# Visualize the cube after transformations**

**fig = plt.figure(figsize=(10, 8))**

**ax1 = fig.add\_subplot(231, projection='3d')**

**plot\_cube(cube\_vertices, cube\_edges, ax1)**

**ax1.set\_title('original cube')**

**ax2 = fig.add\_subplot(232, projection='3d')**

**plot\_cube(translated\_cube, cube\_edges, ax2)**

**ax2.set\_title('Translated Cube')**

**ax3 = fig.add\_subplot(233, projection='3d')**

**plot\_cube(rotated\_x\_cube, cube\_edges, ax3)**

**ax3.set\_title('Rotated 45° around X-axis')**

**ax4 = fig.add\_subplot(234, projection='3d')**

**plot\_cube(rotated\_y\_cube, cube\_edges, ax4)**

**ax4.set\_title('Rotated 45° around Y-axis')**

**ax5 = fig.add\_subplot(235, projection='3d')**

**plot\_cube(rotated\_z\_cube, cube\_edges, ax5)**

**ax5.set\_title('Rotated 45° around Z-axis')**

**ax6 = fig.add\_subplot(236, projection='3d')**

**plot\_cube(scaled\_cube, cube\_edges, ax6)**

**ax6.set\_title('Scaled Cube')**

**plt.show()**

**fig = plt.figure(figsize=(10, 8))**

**ax1 = fig.add\_subplot(131, projection='3d')**

**plot\_cube(cube\_vertices, cube\_edges, ax1)**

**ax1.set\_title('original cube')**

**ax2 = fig.add\_subplot(132, projection='3d')**

**plot\_cube(parallel\_cube, cube\_edges, ax2)**

**ax2.set\_title('parallel projection')**

**ax3 = fig.add\_subplot(133, projection='3d')**

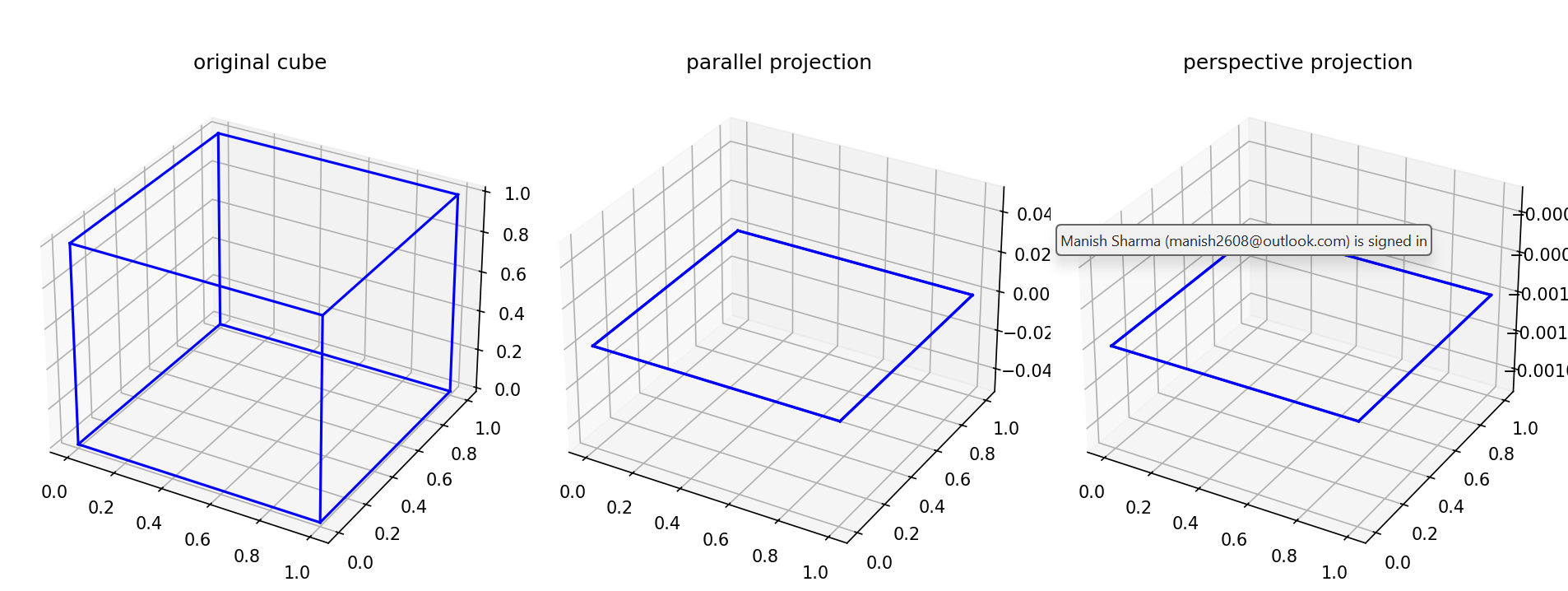
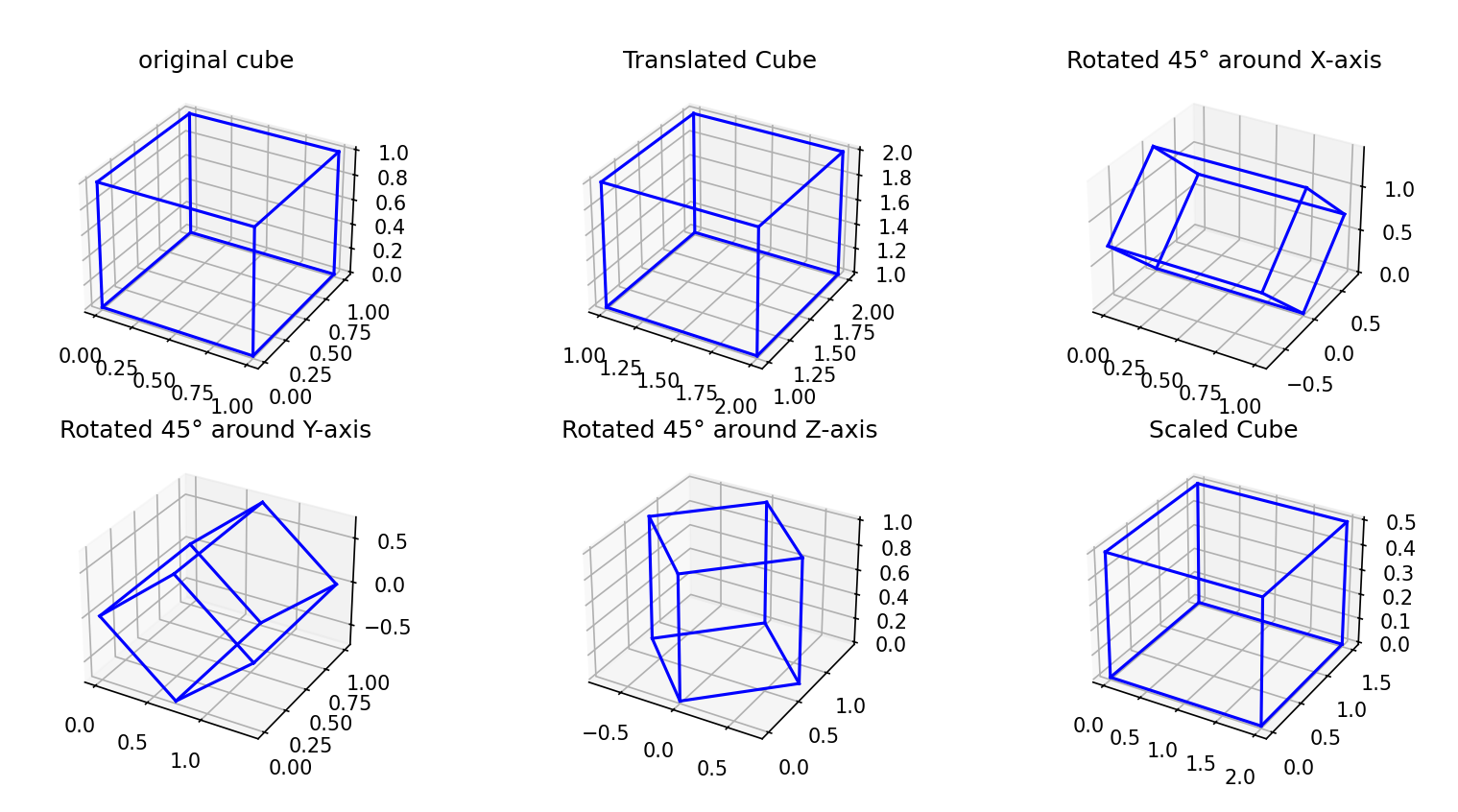
**plot\_cube(perspective\_cube, cube\_edges, ax3)**

**ax3.set\_title('perspective projection')**

**plt.tight\_layout()**

**plt.show()**

**Output:**

****

Q8.Write a program to draw Hermite /Bezier curve.

**Code:**

**import numpy as np**

**import matplotlib.pyplot as plt**

**# Hermite curve function**

**def hermite\_curve(P0, P1, T0, T1, num\_points=100):**

**t = np.linspace(0, 1, num\_points)**

**H = np.zeros((num\_points, 2))**

**for i in range(num\_points):**

**H[i] = (2\*t[i]\*\*3 - 3\*t[i]\*\*2 + 1) \* P0 + \**

**(-2\*t[i]\*\*3 + 3\*t[i]\*\*2) \* P1 + \**

**(t[i]\*\*3 - 2\*t[i]\*\*2 + t[i]) \* T0 + \**

**(t[i]\*\*3 - t[i]\*\*2) \* T1**

**return H**

**# Bezier curve function**

**def bezier\_curve(P0, P1, P2, P3, num\_points=100):**

**t = np.linspace(0, 1, num\_points)**

**B = np.zeros((num\_points, 2))**

**for i in range(num\_points):**

**B[i] = (1-t[i])\*\*3 \* P0 + \**

**3\*t[i]\*(1-t[i])\*\*2 \* P1 + \**

**3\*t[i]\*\*2\*(1-t[i]) \* P2 + \**

**t[i]\*\*3 \* P3**

**return B**

**# Control points for Hermite curve**

**P0\_hermite = np.array([1, 1])**

**P1\_hermite = np.array([3, 3])**

**T0\_hermite = np.array([2, 0])**

**T1\_hermite = np.array([0, 2])**

**# Control points for Bezier curve**

**P0\_bezier = np.array([1, 1])**

**P1\_bezier = np.array([2, 3])**

**P2\_bezier = np.array([4, 2])**

**P3\_bezier = np.array([5, 5])**

**# Calculate Hermite curve points**

**H = hermite\_curve(P0\_hermite, P1\_hermite, T0\_hermite, T1\_hermite)**

**# Calculate Bezier curve points**

**B = bezier\_curve(P0\_bezier, P1\_bezier, P2\_bezier, P3\_bezier)**

**# Plot Hermite curve**

**plt.plot(H[:,0], H[:,1], label='Hermite Curve', color='blue')**

**# Plot Bezier curve**

**plt.plot(B[:,0], B[:,1], label='Bezier Curve', color='red')**

**# Plot control points**

**plt.scatter([P0\_hermite[0], P1\_hermite[0], T0\_hermite[0], T1\_hermite[0]],**

**[P0\_hermite[1], P1\_hermite[1], T0\_hermite[1], T1\_hermite[1]],**

**color='blue')**

**plt.scatter([P0\_bezier[0], P1\_bezier[0], P2\_bezier[0], P3\_bezier[0]],**

**[P0\_bezier[1], P1\_bezier[1], P2\_bezier[1], P3\_bezier[1]],**

**color='red')**

**# Plot lines connecting control points for Hermite curve**

**plt.plot([P0\_hermite[0], T0\_hermite[0]], [P0\_hermite[1], T0\_hermite[1]], 'b--')**

**plt.plot([P1\_hermite[0], T1\_hermite[0]], [P1\_hermite[1], T1\_hermite[1]], 'b--')**

**# Plot lines connecting control points for Bezier curve**

**plt.plot([P0\_bezier[0], P1\_bezier[0], P2\_bezier[0], P3\_bezier[0]],**

**[P0\_bezier[1], P1\_bezier[1], P2\_bezier[1], P3\_bezier[1]], 'r--')**

**plt.xlabel('X')**

**plt.ylabel('Y')**

**plt.title('Hermite and Bezier Curves')**

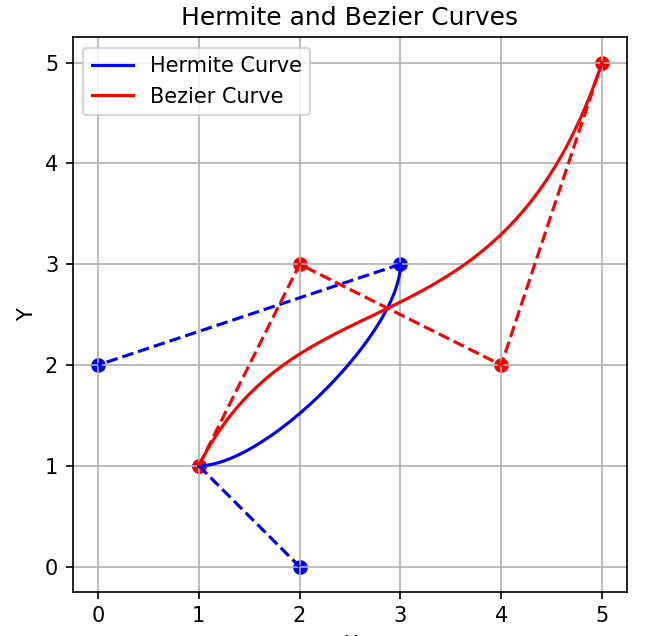
**plt.legend()**

**plt.grid(True)**

**plt.gca().set\_aspect('equal', adjustable='box')**

**plt.show()**

**Output:**

****