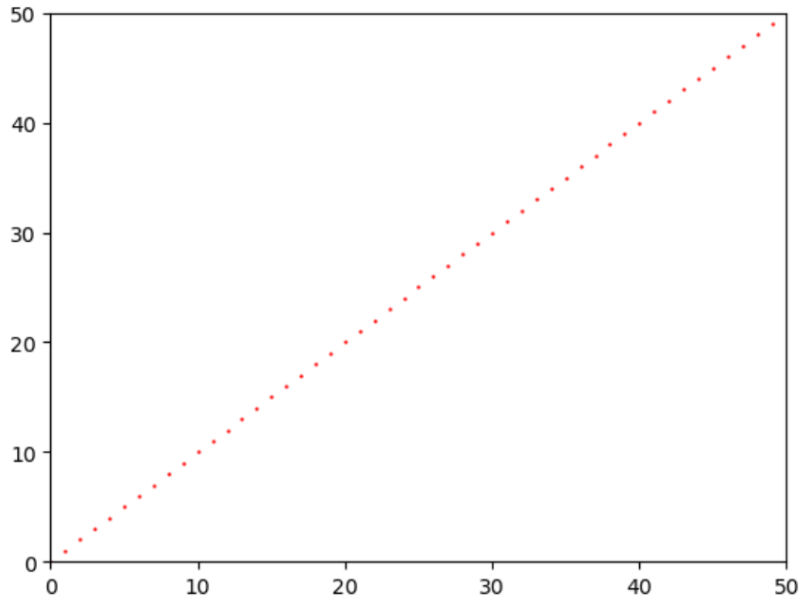
1. Write a program to implement DDA and Bresenham's line drawing algorithm.

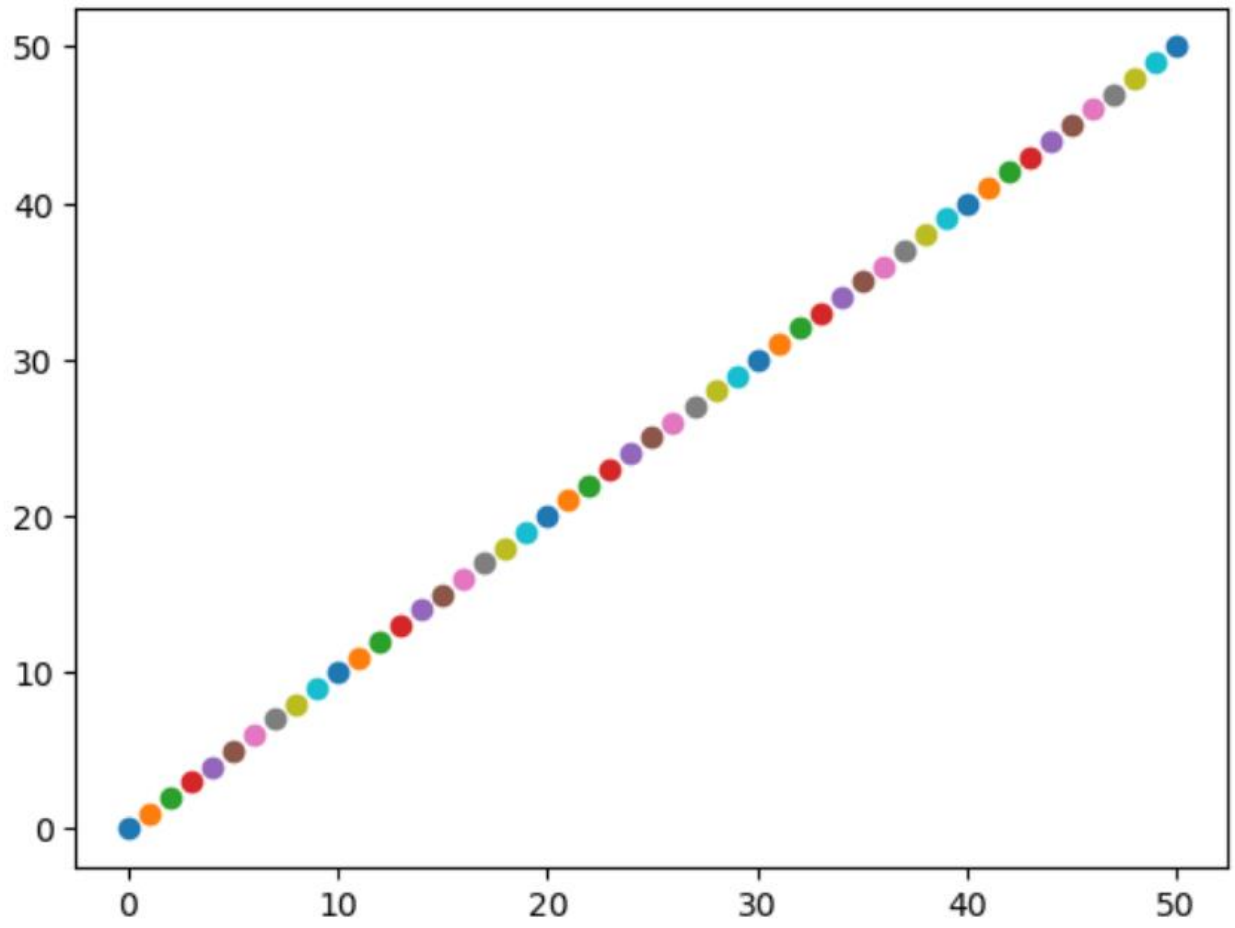
```
#CODE for DDA ALGORITHM
import matplotlib.pyplot as plt

#Initialising points and slope for the line
x0 = 0
y0 = 0
x1 = 50
y1 = 50
m = (y1-y0)/(x1-x0)

#Calculating x and y coordinates using DDA Algorithm
x_points = []
y_points = []
while x0<x1:
    x_points.append(x0)
    y_points.append(round(y0))
    x0+=1
    y0+=m

#Plotting the line
plt.scatter(x_points, y_points, c = 'red', s=0.5)
plt.xlim(0,50)
plt.ylim(0,50)
plt.show()
```



#CODE for BRESENHAM'S ALGORITHM

```
import matplotlib.pyplot as plt

def draw_line(x1, y1, x2, y2):
    dx = abs(x2 - x1)
    dy = abs(y2 - y1)
    slope = dy > dx

    if slope:
        x1, y1 = y1, x1
        x2, y2 = y2, x2

    if x1 > x2:
        x1, x2 = x2, x1
        y1, y2 = y2, y1

    dx = x2 - x1
    dy = abs(y2 - y1)
    p = 2 * dy - dx

    points = []
    y = y1
    for x in range(x1, x2 + 1):
        points.append((y, x) if slope else (x, y))
        if p >= 0:
            y += 1 if y1 < y2 else -1
            p -= 2 * dx
        p += 2 * dy

    return points

# Input endpoints of the line
x1, y1 = map(int, input("Enter endpoint 1 (x1 y1): ").split())
x2, y2 = map(int, input("Enter endpoint 2 (x2 y2): ").split())

# Get points of the line using Bresenham's Line Drawing Algorithm
line_points = draw_line(x1, y1, x2, y2)

# Print the line points
#print("Line points:")
#for point in line_points:
#    print(point)
```

```
for i in line_points:
    plt.scatter(i[0], i[1])
plt.show()
```

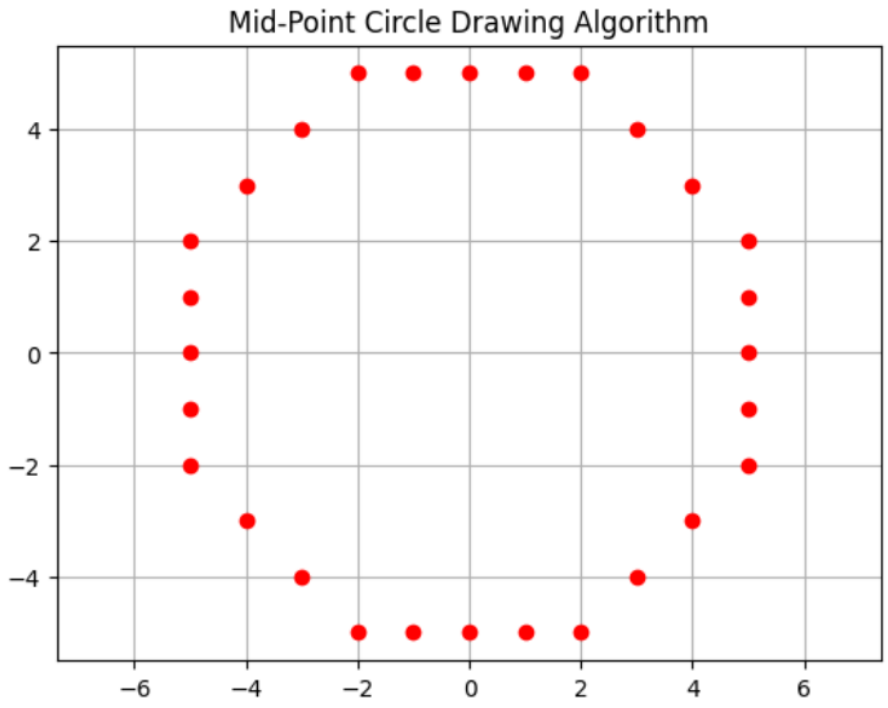


2. Write a program to implement mid-point circle drawing algorithm.

```
#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()

# Test the function
radius = 5
draw_circle_midpoint(radius)
```

```
Intermediate points:
(1, 5), (-1, 5), (1, -5), (-1, -5), (5, 1), (-5, 1), (5, -1), (-5, -1)
(2, 5), (-2, 5), (2, -5), (-2, -5), (5, 2), (-5, 2), (5, -2), (-5, -2)
(3, 4), (-3, 4), (3, -4), (-3, -4), (4, 3), (-4, 3), (4, -3), (-4, -3)
(4, 3), (-4, 3), (4, -3), (-4, -3), (3, 4), (-3, 4), (3, -4), (-3, -4)
```



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

```
#CODE
def compute_outcode(x, y, xmin, ymin, xmax, ymax):
    code = 0
    if x < xmin:
        code |= 1
    elif x > xmax:
        code |= 2
    if y < ymin:
        code |= 4
    elif y > ymax:
        code |= 8
    return code

def cohen_sutherland(x1, y1, x2, y2, xmin, ymin, xmax, ymax):
    code1 = compute_outcode(x1, y1, xmin, ymin, xmax, ymax)
    code2 = compute_outcode(x2, y2, xmin, ymin, xmax, ymax)
    accept = False

    while True:
        if code1 == 0 and code2 == 0:
            accept = True
            break
        elif code1 & code2 != 0:
            break
        else:
            x, y = 0, 0
            if code1 != 0:
                code_out = code1
            else:
                code_out = code2

            if code_out & 1:
                x = xmin
                y = y1 + (y2 - y1) * (xmin - x1) / (x2 - x1)
            elif code_out & 2:
                x = xmax
                y = y1 + (y2 - y1) * (xmax - x1) / (x2 - x1)
            elif code_out & 4:
```

```
                y = ymin
                x = x1 + (x2 - x1) * (ymin - y1) / (y2 - y1)
            elif code_out & 8:
                y = ymax
                x = x1 + (x2 - x1) * (ymax - y1) / (y2 - y1)

            if code_out == code1:
                x1 = x
                y1 = y
                code1 = compute_outcode(x1, y1, xmin, ymin, xmax, ymax)
            else:
                x2 = x
                y2 = y
                code2 = compute_outcode(x2, y2, xmin, ymin, xmax, ymax)

    if accept:
        print("Line after clipping:", (x1, y1), "-", (x2, y2))
    else:
        print("Line lies completely outside the clipping window")

# Input endpoint of the line and screen dimensions
x1, y1 = map(int, input("Enter endpoint 1 (x1 y1): ").split())
x2, y2 = map(int, input("Enter endpoint 2 (x2 y2): ").split())
width = int(input("Enter screen width: "))
height = int(input("Enter screen height: "))

# Clipping window coordinates
xmin, ymin = 0, 0
xmax, ymax = width - 1, height - 1

# Perform Cohen-Sutherland line clipping
cohen_sutherland(x1, y1, x2, y2, xmin, ymin, xmax, ymax)
```

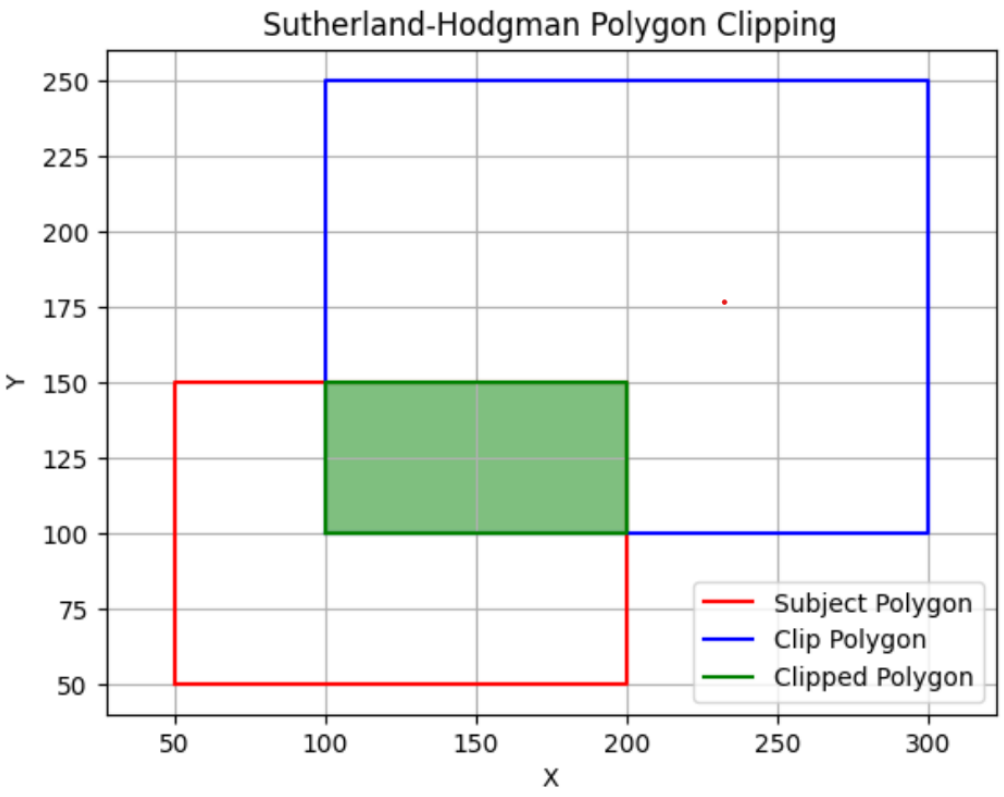
```
Enter endpoint 1 (x1 y1): -50 50
Enter endpoint 2 (x2 y2): 150 50
Enter screen width: 100
Enter screen height: 100
Line after clipping: (0, 50.0) - (99, 50.0)
```

4. 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()
```



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

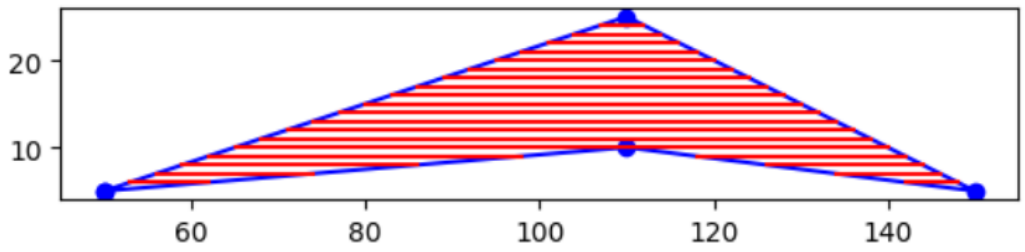
#CODE

```
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')
            print("Active edges at y =", y, ":", active_edges)

        active_edges = [x for x in active_edges if x not in scanline_points[y]]

# Example usage
vertices = [(50, 5), (110, 10), (150, 5), (110, 25),(50,5)]
plt.figure()
plt.gca().set_aspect('equal', adjustable='box')
plt.plot(*zip(*vertices), marker='o', color='b')
fill_polygon(vertices)
plt.show()
```

#OUTPUT

```
Active edges at y = 5 : [50.0, 50.0, 150.0, 150.0]
Active edges at y = 6 : [53.0, 62.0, 142.0, 148.0]
Active edges at y = 7 : [56.0, 74.0, 134.0, 146.0]
Active edges at y = 8 : [59.0, 86.0, 126.0, 144.0]
Active edges at y = 9 : [62.0, 98.0, 118.0, 142.0]
Active edges at y = 10 : [65.0, 140.0]
Active edges at y = 11 : [68.0, 138.0]
Active edges at y = 12 : [71.0, 136.0]
Active edges at y = 13 : [74.0, 134.0]
Active edges at y = 14 : [77.0, 132.0]
Active edges at y = 15 : [80.0, 130.0]
Active edges at y = 16 : [83.0, 128.0]
Active edges at y = 17 : [86.0, 126.0]
Active edges at y = 18 : [89.0, 124.0]
Active edges at y = 19 : [92.0, 122.0]
Active edges at y = 20 : [95.0, 120.0]
Active edges at y = 21 : [98.0, 118.0]
Active edges at y = 22 : [101.0, 116.0]
Active edges at y = 23 : [104.0, 114.0]
Active edges at y = 24 : [107.0, 112.0]
```



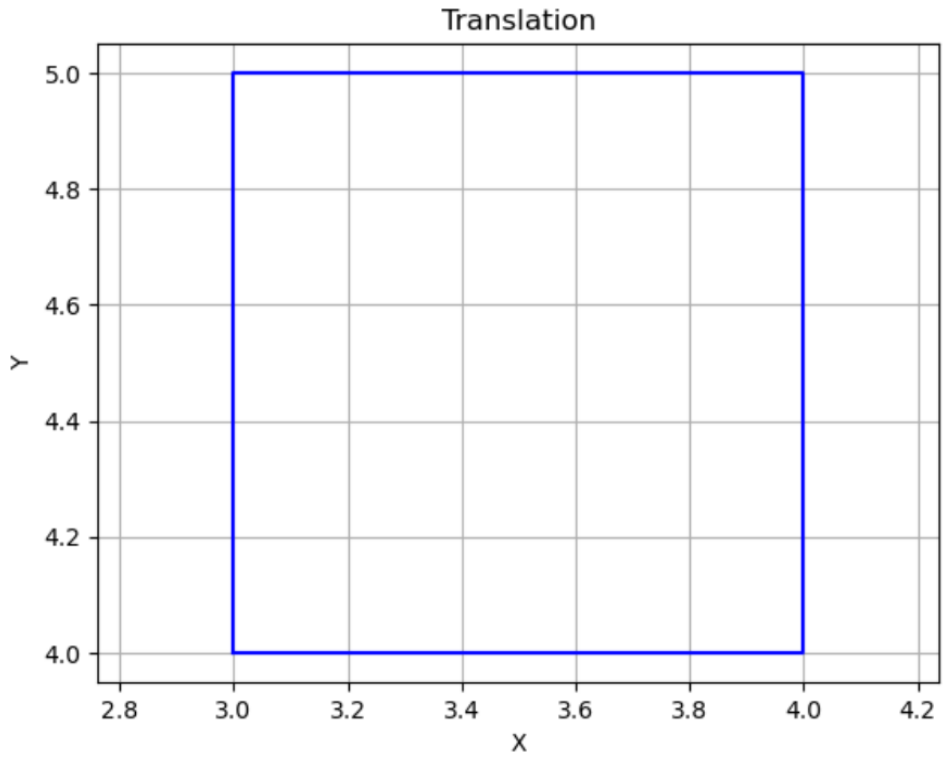
6. Write a program to apply various 2D transformations on a 2D object (use homogenous Coordinates).

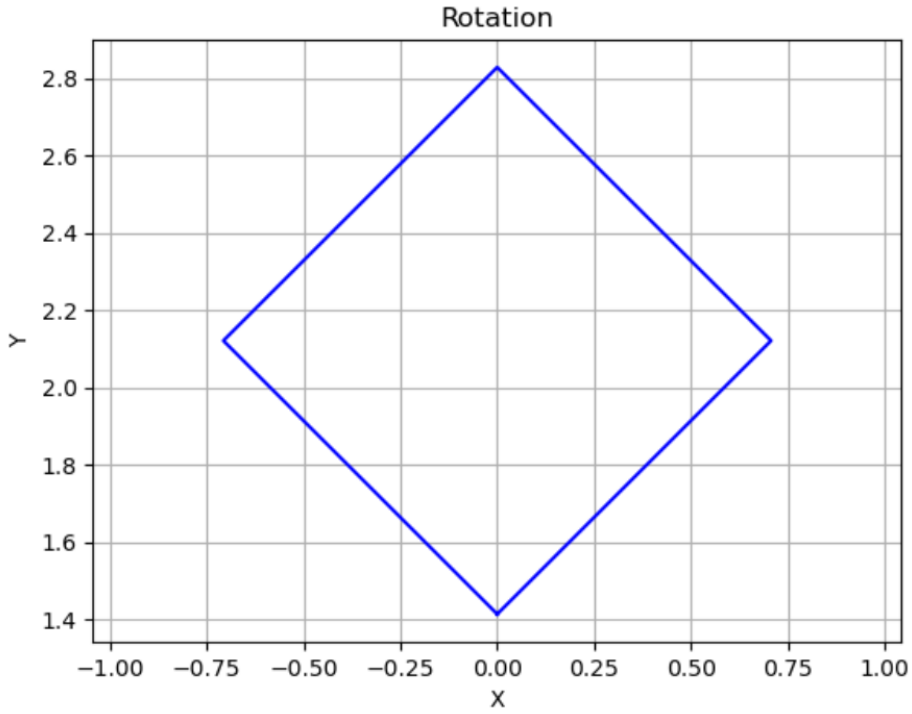
#CODE

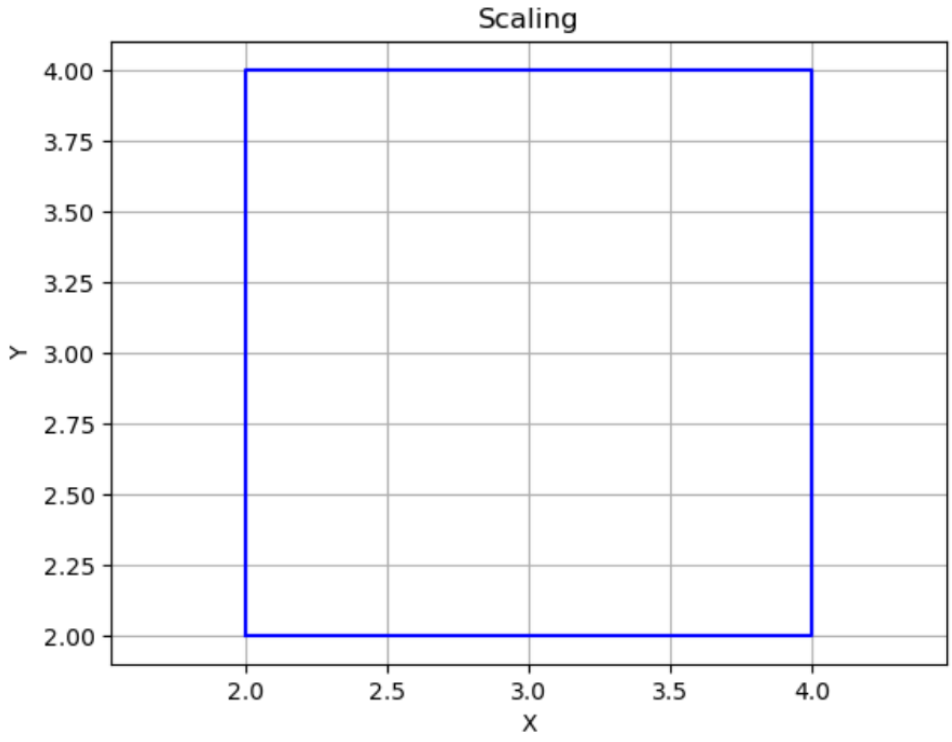
```
import numpy as np
import matplotlib.pyplot as plt

class Transformation2D:
    def __init__(self, points):
        self.points = np.array(points)
        self.homogeneous_points = np.concatenate((self.points,
np.ones((len(points), 1))), axis=1)

    def translate(self, tx, ty):
        translation_matrix = np.array([[1, 0, tx],
                                       [0, 1, ty],
                                       [0, 0, 1]])
        transformed_points = np.dot(translation_matrix,
self.homogeneous_points.T).T
        return transformed_points[:, :-1]

    def rotate(self, theta):
        rotation_matrix = np.array([[np.cos(theta), -np.sin(theta), 0],
                                    [np.sin(theta), np.cos(theta), 0],
                                    [0, 0, 1]])
        transformed_points = np.dot(rotation_matrix, self.homogeneous_points.T).T
        return transformed_points[:, :-1]

    def scale(self, sx, sy):
        scaling_matrix = np.array([[sx, 0, 0],
                                   [0, sy, 0],
                                   [0, 0, 1]])
        transformed_points = np.dot(scaling_matrix, self.homogeneous_points.T).T
        return transformed_points[:, :-1]

    def plot(self, points, title):
        plt.figure()
        plt.plot(points[:, 0], points[:, 1], 'b-')
        plt.title(title)
        plt.xlabel('X')
        plt.ylabel('Y')
        plt.grid(True)
        plt.axis('equal')
        plt.show()
```

```
# Define initial points
points = [[1, 1], [2, 1], [2, 2], [1, 2], [1, 1]]
transformer = Transformation2D(points)

# Translation
translated_points = transformer.translate(2, 3)
transformer.plot(translated_points, 'Translation')

# Rotation
rotated_points = transformer.rotate(np.pi/4)
transformer.plot(rotated_points, 'Rotation')

# Scaling
scaled_points = transformer.scale(2, 2)
transformer.plot(scaled_points, 'Scaling')
```

#OUTPUT




Rotation
2.8
2.6
2.4
2.2
2.0
1.8
1.6
1.4
Y
−1.00
−0.75
−0.50
−0.25
0.00
0.25
0.50
0.75
1.00
X


Scaling
4.00
3.75
3.50
3.25
3.00
2.75
2.50
2.25
2.00
Y
2.0
2.5
3.0
3.5
4.0
X

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

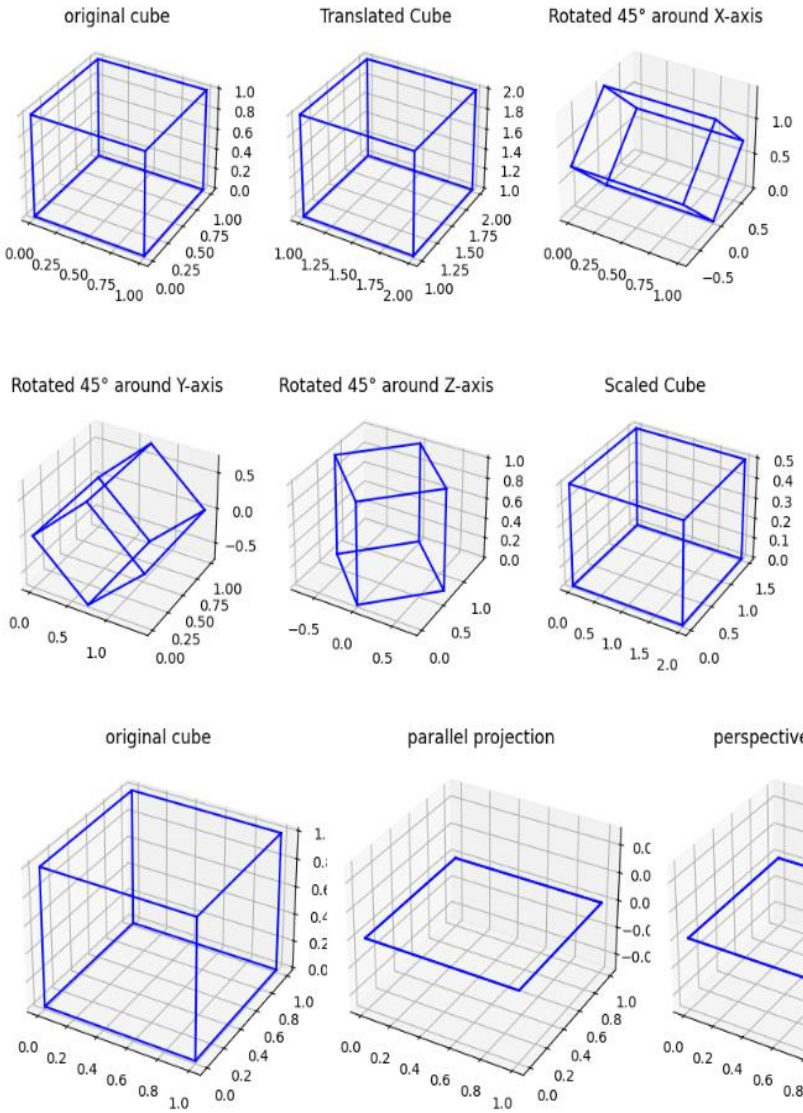
#CODE

```
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()
```

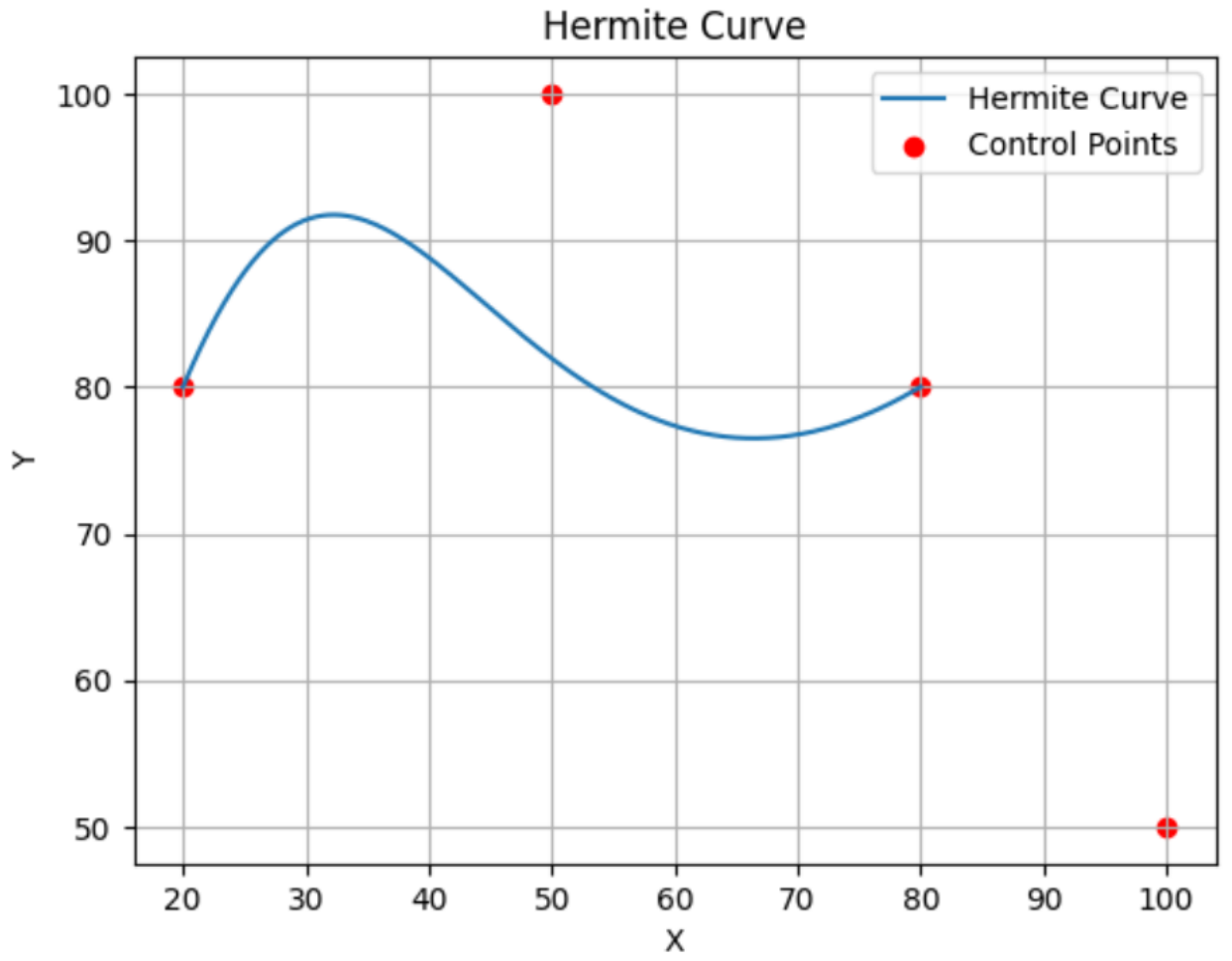
#OUTPUT


original cube
Translated Cube
Rotated 45° around X-axis
Rotated 45° around Y-axis
Rotated 45° around Z-axis
Scaled Cube
original cube
parallel projection
perspective projection

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

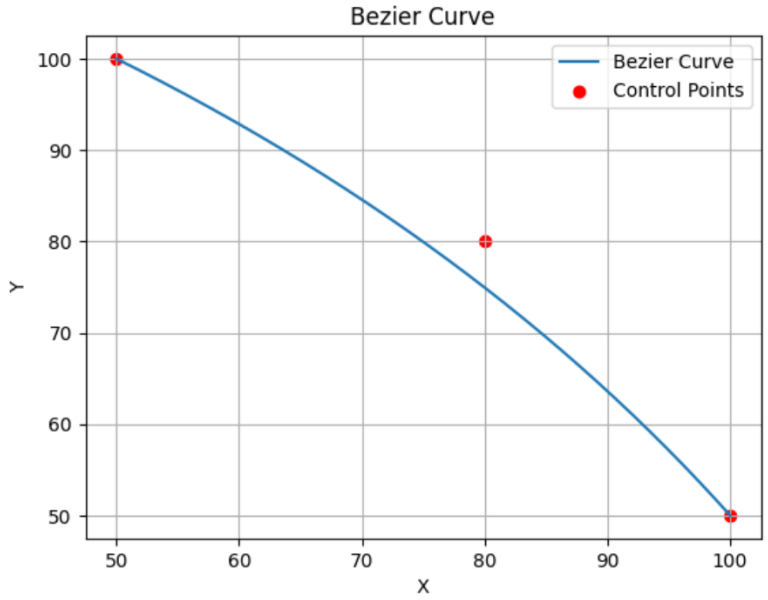
```
#CODE
#HERMITE CURVE

import numpy as np
import matplotlib.pyplot as plt

def make_hermite(xys):
    n = len(xys)
    def hermite(ts):
        result = []
        for t in ts:
            h00 = 2 * t**3 - 3 * t**2 + 1
            h10 = t**3 - 2 * t**2 + t
            h01 = -2 * t**3 + 3 * t**2
            h11 = t**3 - t**2
            result.append(tuple(sum([h * p for h, p in zip([h00, h10, h01, h11],
ps)]) for ps in zip(*xys)))
        return result

    return hermite

if __name__ == '__main__':
    ts = np.linspace(0.0, 1.0, 1000)
    xys = [(20, 80), (50, 100), (80, 80), (100, 50)]  # Example control points
    hermite = make_hermite(xys)
    points = hermite(ts)

    plt.plot(*zip(*points), label='Hermite Curve')
    plt.scatter(*zip(*xys), color='red', label='Control Points')
    plt.legend()
    plt.xlabel('X')
    plt.ylabel('Y')
    plt.title('Hermite Curve')
    plt.grid(True)
    plt.show()
```

#OUTPUT


Hermite Curve
Hermite Curve
Control Points
100
90
80
70
60
50
Y
20
30
40
50
60
70
80
90
100
X

```
#BEZIER CURVE

def make_bezier(xys):
    n = len(xys)
    def bezier(ts):
        result = []
        for t in ts:
            tpowers = (t**i for i in range(n))
            upowers = reversed([(1 - t)**i for i in range(n)])
            coefs = [c * a * b for c, a, b in zip(pascal_row(n - 1), tpowers,
upowers)]
            result.append(tuple(sum([coef * p for coef, p in zip(coefs, ps)]) for
ps in zip(*xys)))
        return result

    return bezier

def pascal_row(n):
    result = [1]
    x, numerator = 1, n
    for denominator in range(1, n // 2 + 1):
        x *= numerator
        x /= denominator
        result.append(x)
        numerator -= 1
    if n & 1 == 0:  # n is even
        result.extend(reversed(result[:-1]))
    else:
        result.extend(reversed(result))
    return result

if __name__ == '__main__':
    ts = np.linspace(0.0, 1.0, 1000)
    xys = [(50, 100), (80, 80), (100, 50)]  # Example control points
    bezier = make_bezier(xys)
    points = bezier(ts)

    plt.plot(*zip(*points), label='Bezier Curve')
    plt.scatter(*zip(*xys), color='red', label='Control Points')
    plt.legend()
    plt.xlabel('X')
    plt.ylabel('Y')
    plt.title('Bezier Curve')
    plt.grid(True)
    plt.show()
```

#OUTPUT


Bezier Curve
Bezier Curve
Control Points
100
90
80
70
60
50
Y
50
60
70
80
90
100
X