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

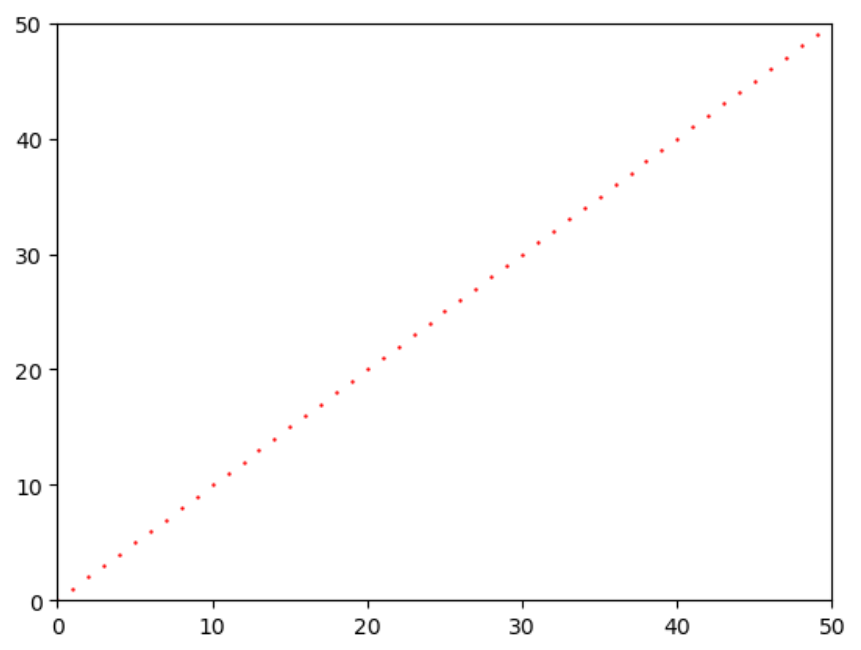
#Plotting the line

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:")

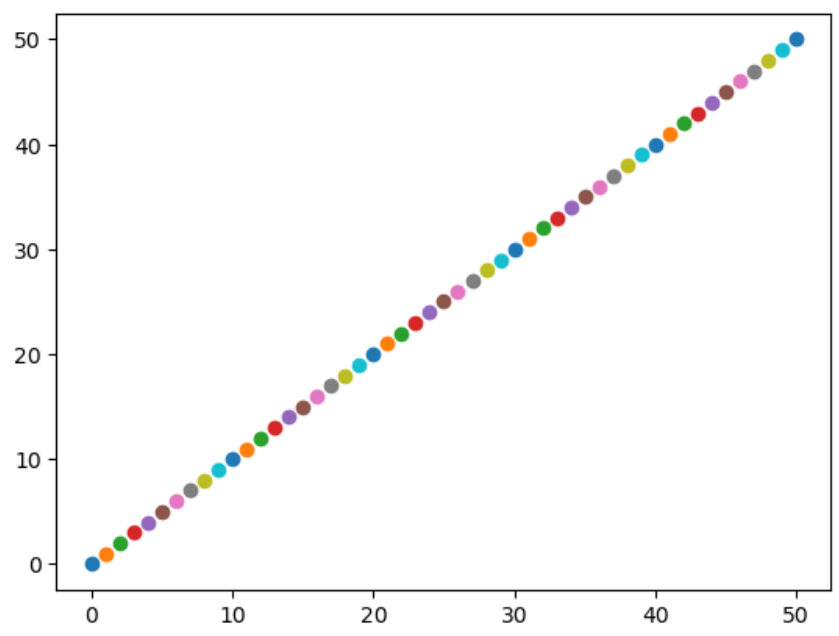
#for point in line\_points:

#    print(point)

for i in line\_points:

    plt.scatter(i[0], i[1])

plt.show()



1. 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')

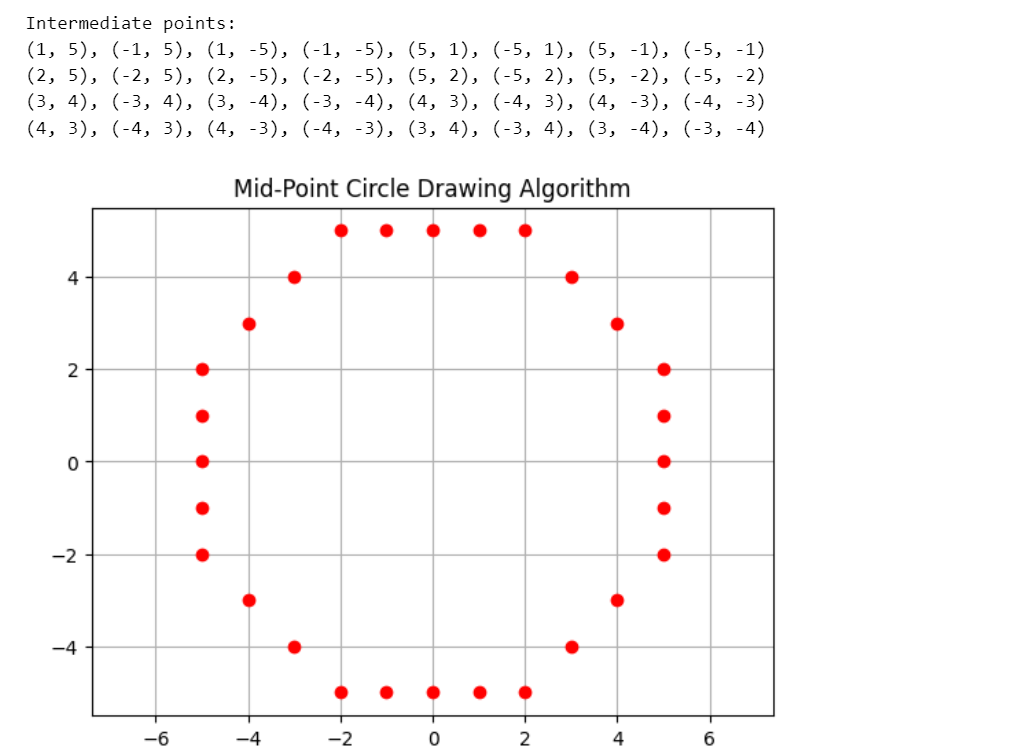
    plt.title('Mid-Point Circle Drawing Algorithm')

    plt.show()

# Test the function

radius = 5

draw\_circle\_midpoint(radius)



1. 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: "))

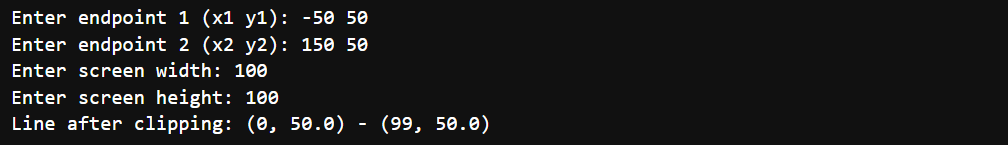
# Clipping window coordinates

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)



1. 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')

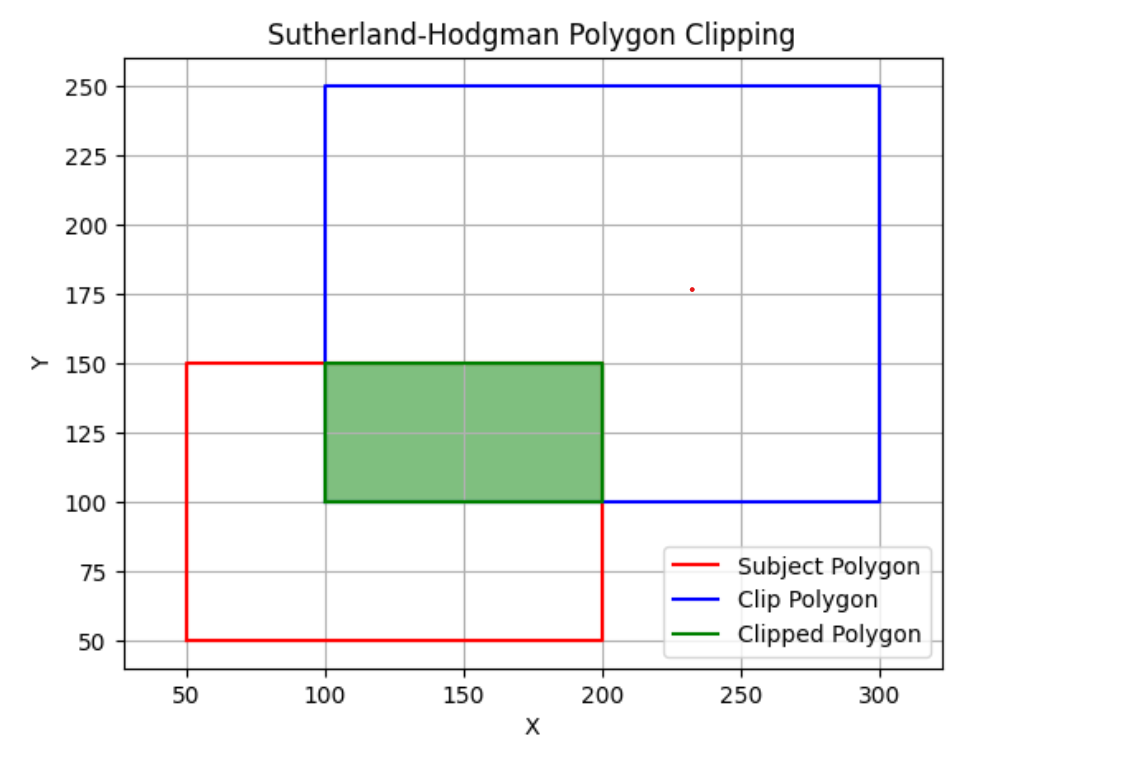
plt.ylabel('Y')

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

plt.axis('equal')

plt.grid(True)

plt.show()



1. 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()

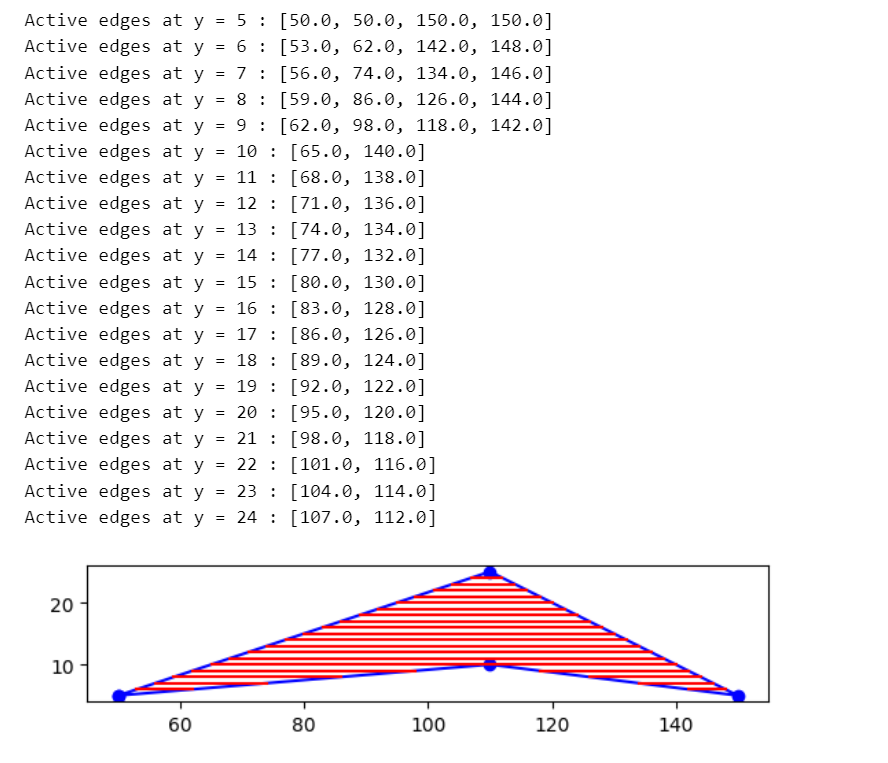
plt.gca().set\_aspect('equal', adjustable='box')

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

fill\_polygon(vertices)

plt.show()

#OUTPUT



1. 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)

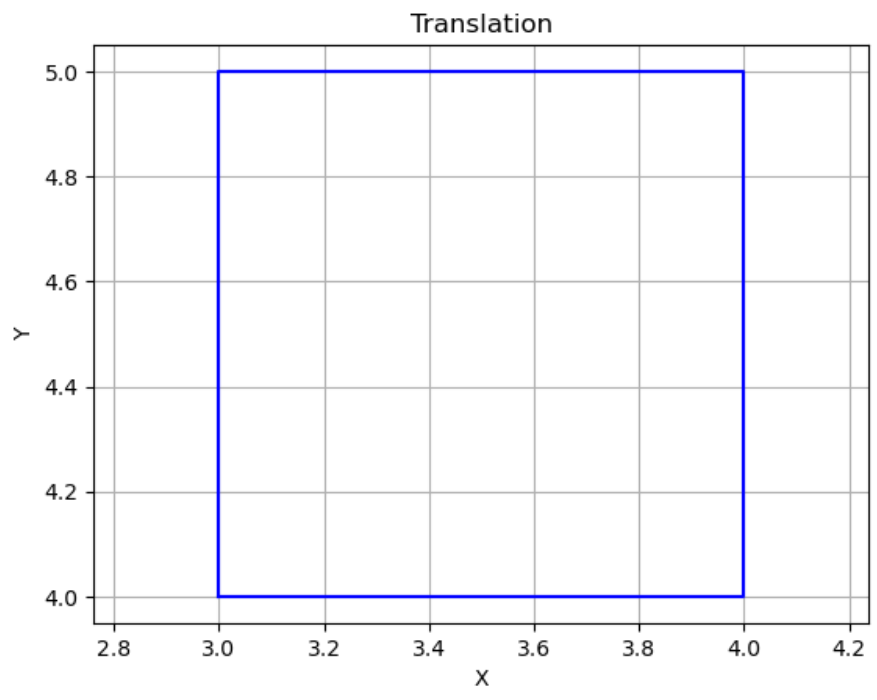
transformer.plot(rotated\_points, 'Rotation')

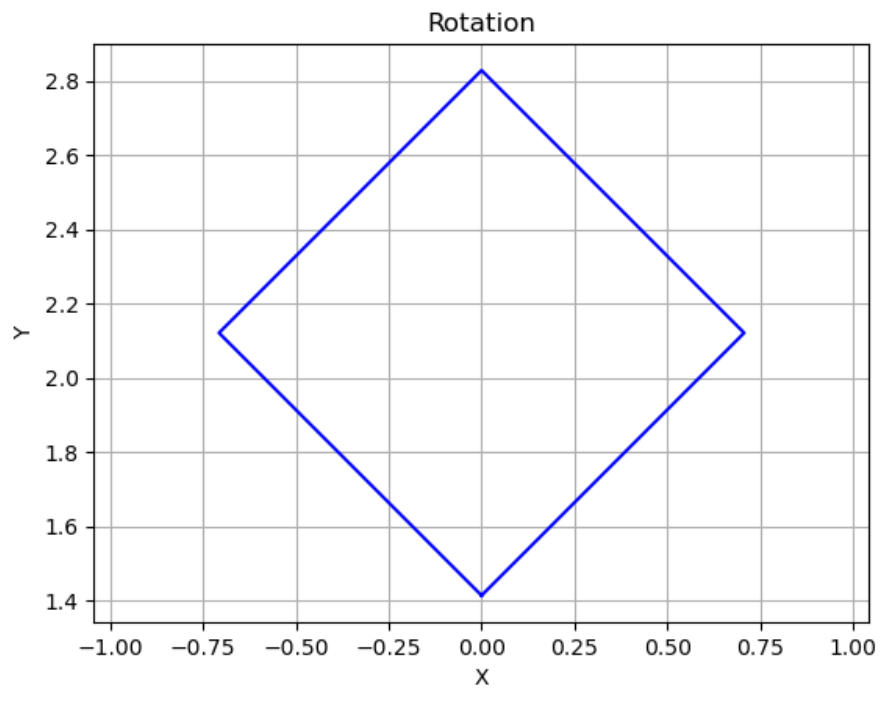
# Scaling

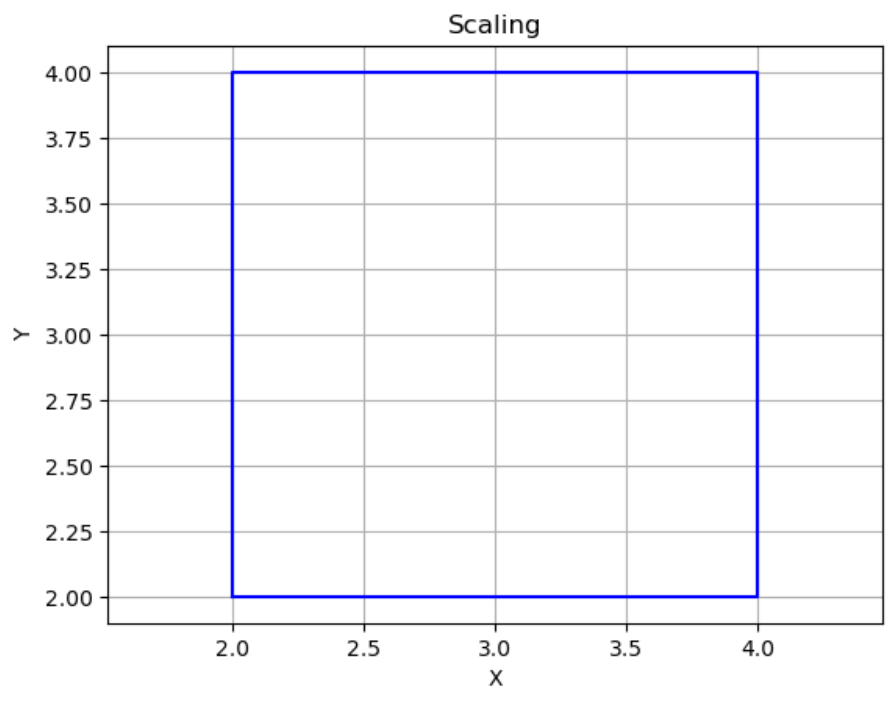
scaled\_points = transformer.scale(2, 2)

transformer.plot(scaled\_points, 'Scaling')

#OUTPUT







1. 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')

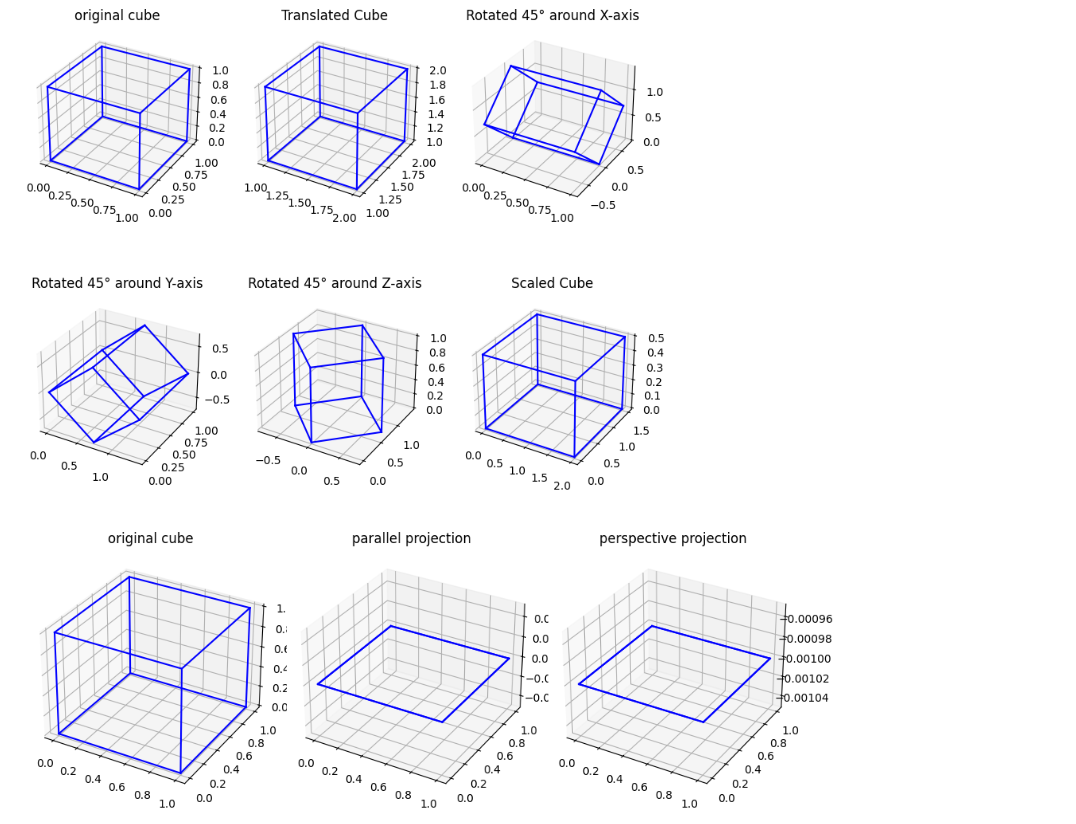
plot\_cube(perspective\_cube, cube\_edges, ax3)

ax3.set\_title('perspective projection')

plt.tight\_layout()

plt.show()

#OUTPUT



1. 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')

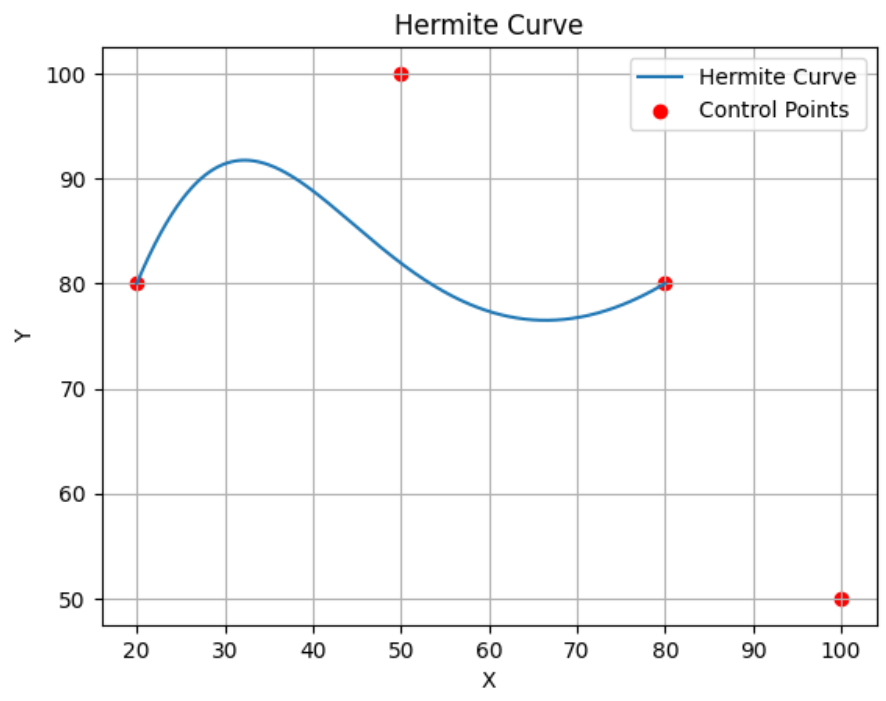
    plt.ylabel('Y')

    plt.title('Hermite Curve')

    plt.grid(True)

    plt.show()

#OUTPUT



#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

