**LAB-1**

**(Title:** **Digital Differential Algorithm**)

* **Implement Digital Differential Algorithm (DDA). Also, print the output coordinates and plot the resultant line.**

from matplotlib import pyplot as plt

def DDA(x0, y0, x1, y1):

# find absolute differences

dx = abs(x0 - x1)

dy = abs(y0 - y1)

# find maximum difference

steps = max(dx, dy)

# calculate the increment in x and y

xinc = dx/steps

yinc = dy/steps

# start with 1st point

x = float(x0)

y = float(y0)

# make a list for coordinates

x\_coorinates = []

y\_coorinates = []

for i in range(steps):

# append the x,y coordinates in respective list

x\_coorinates.append(x)

y\_coorinates.append(y)

# increment the values

x = x + xinc

y = y + yinc

# plot the line with coordinates list

plt.plot(x\_coorinates, y\_coorinates, marker="o",

markersize=1, markerfacecolor="green")

plt.show()

# Driver code

if \_\_name\_\_ == "\_\_main\_\_":

# coordinates of 1st point

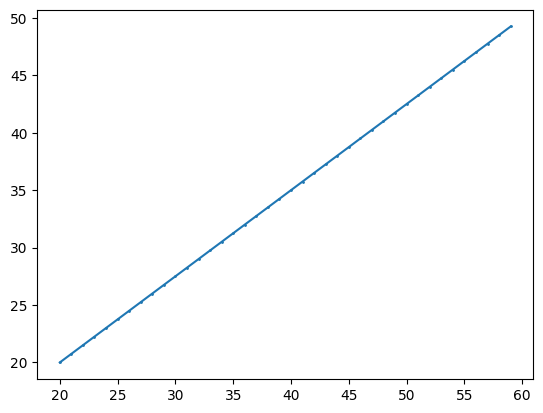
x0, y0 = 20, 20

# coordinates of 2nd point

x1, y1 = 60, 50

# Function call

DDA(x0, y0, x1, y1)



**Output to the  # Function call DDA(x0, y0, x1, y1)**

**LAB-2**

**(Title:** **Bresenham’s drawing Algorithm**)

* **Implementation of Bresenham's Line Drawing Algorithm: Print the Resultant Coordinates and Plot the Line Graph**

import matplotlib.pyplot as plt

def bresenham\_line(x1, y1, x2, y2):

points = []

dx = abs(x2 - x1)

dy = abs(y2 - y1)

sx = 1 if x1 < x2 else -1

sy = 1 if y1 < y2 else -1

err = dx - dy

while True:

points.append((x1, y1))

if x1 == x2 and y1 == y2:

break

e2 = 2 \* err

if e2 > -dy:

err -= dy

x1 += sx

if e2 < dx:

err += dx

y1 += sy

return points

# Example usage:

x1, y1 = 2, 3

x2, y2 = 10, 8

line\_points = bresenham\_line(x1, y1, x2, y2)

# Print the coordinates

for point in line\_points:

print(point)

# Plotting the line

x\_coords, y\_coords = zip(\*line\_points)

plt.plot(x\_coords, y\_coords, marker='o')

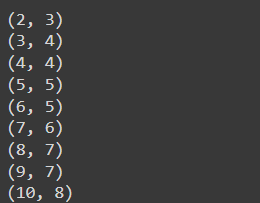
plt.title("Bresenham's Line Drawing Algorithm")

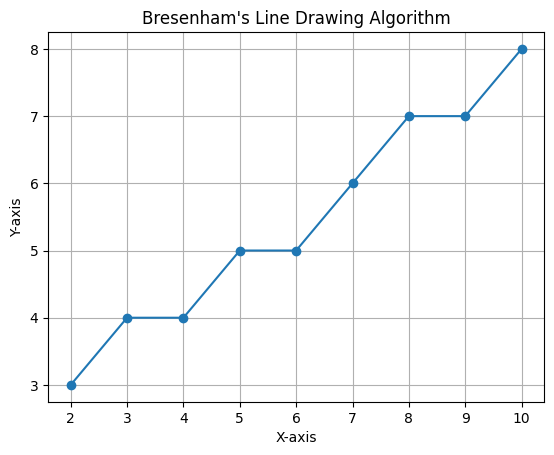
plt.xlabel('X-axis')

plt.ylabel('Y-axis')

plt.grid(True)

plt.show()





* **Implementation of Bresenham's Circle Drawing Algorithm with Symmetric Coordinates and Resultant Circle**

import matplotlib.pyplot as plt

def draw\_circle(x\_center, y\_center, radius):

    x = 0

    y = radius

    d = 1 - radius  # Decision parameter

    points = []

    def plot\_circle\_points(x\_center, y\_center, x, y):

        points.append((x\_center + x, y\_center + y))

        points.append((x\_center - x, y\_center + y))

        points.append((x\_center + x, y\_center - y))

        points.append((x\_center - x, y\_center - y))

        points.append((x\_center + y, y\_center + x))

        points.append((x\_center - y, y\_center + x))

        points.append((x\_center + y, y\_center - x))

        points.append((x\_center - y, y\_center - x))

    # Plot the initial set of points

    plot\_circle\_points(x\_center, y\_center, x, y)

    # Loop to calculate the points in the first quadrant

    while x < y:

        x += 1

        if d < 0:

            d += 2 \* x + 1

        else:

            y -= 1

            d += 2 \* (x - y) + 1

        plot\_circle\_points(x\_center, y\_center, x, y)

    return points

def plot\_circle(points):

    x\_values = [p[0] for p in points]

    y\_values = [p[1] for p in points]

    plt.scatter(x\_values, y\_values)

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

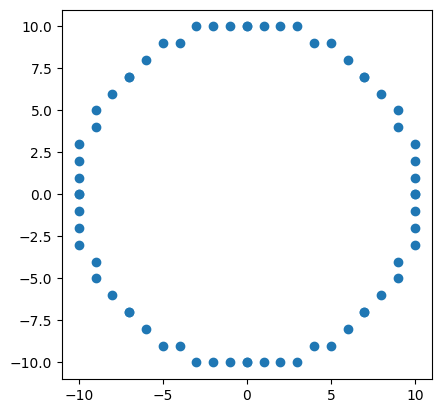
    plt.show()

# Example usage

x\_center, y\_center, radius = 0, 0, 10

circle\_points = draw\_circle(x\_center, y\_center, radius)

plot\_circle(circle\_points)



**LAB-3**

**(Title:** **Mid-Point Circle drawing algorithm**)

* **Implementation of Mid-Point Circle drawing algorithm. Also, show all the symmetrical octant coordinates along with resultant circle**

import matplotlib.pyplot as plt

def plot\_circle\_points(x\_center, y\_center, x, y):

# Plot the 8 symmetrical points for the given (x, y)

points = [

(x\_center + x, y\_center + y),

(x\_center - x, y\_center + y),

(x\_center + x, y\_center - y),

(x\_center - x, y\_center - y),

(x\_center + y, y\_center + x),

(x\_center - y, y\_center + x),

(x\_center + y, y\_center - x),

(x\_center - y, y\_center - x)

]

for point in points:

plt.plot(point[0], point[1], 'bo') # Plot each point

return points

def mid\_point\_circle(x\_center, y\_center, radius):

x = 0

y = radius

d = 1 - radius # Initial decision parameter

symmetric\_points = []

symmetric\_points.extend(plot\_circle\_points(x\_center, y\_center, x, y)) # Plot initial points

while x < y:

x += 1

if d < 0:

d = d + 2 \* x + 1

else:

y -= 1

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

symmetric\_points.extend(plot\_circle\_points(x\_center, y\_center, x, y))

return symmetric\_points

def draw\_circle\_with\_octants(x\_center, y\_center, radius):

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

symmetric\_points = mid\_point\_circle(x\_center, y\_center, radius)

# Plotting the resultant circle using matplotlib's Circle

circle = plt.Circle((x\_center, y\_center), radius, color='r', fill=False, linestyle='--')

plt.gca().add\_patch(circle)

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

plt.grid(True)

plt.xlim(x\_center - radius - 1, x\_center + radius + 1)

plt.ylim(y\_center - radius - 1, y\_center + radius + 1)

plt.show()

return symmetric\_points

# Example usage:

x\_center = 0

y\_center = 0

radius = 10

# Draw the circle and get the octant points

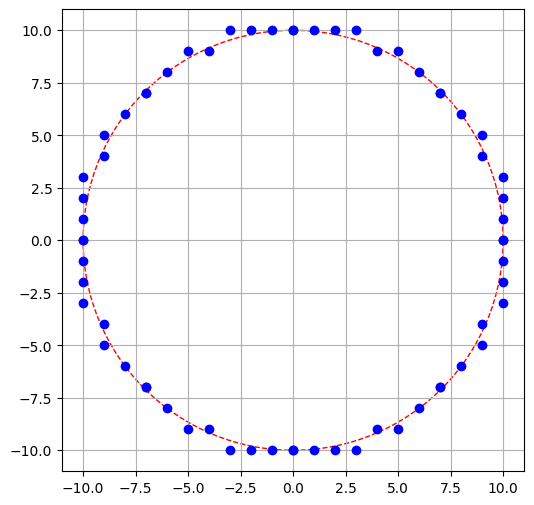
octant\_points = draw\_circle\_with\_octants(x\_center, y\_center, radius)

# Display the symmetric octant coordinates

print("Symmetric Octant Coordinates:")

for i, point in enumerate(octant\_points, start=1):

print(f"{i}: {point}")



**LAB-4**

**(Title:** **Mid-Point Ellipse drawing algorithm**)

* **Implementation of Mid-Point Ellipse drawing algorithm. Also, show all the symmetrical octant coordinates along with resultant circle**

import matplotlib.pyplot as plt

def plot\_ellipse\_points(x\_center, y\_center, x, y):

# Plot the 4 symmetrical points for the given (x, y)

points = [

(x\_center + x, y\_center + y),

(x\_center - x, y\_center + y),

(x\_center + x, y\_center - y),

(x\_center - x, y\_center - y)

]

for point in points:

plt.plot(point[0], point[1], 'bo') # Plot each point

return points

def midpoint\_ellipse(x\_center, y\_center, rx, ry):

x = 0

y = ry

# Decision parameter for region 1

d1 = (ry \* ry) - (rx \* rx \* ry) + (0.25 \* rx \* rx)

dx = 2 \* ry \* ry \* x

dy = 2 \* rx \* rx \* y

symmetric\_points = []

# Region 1

while dx < dy:

symmetric\_points.extend(plot\_ellipse\_points(x\_center, y\_center, x, y))

if d1 < 0:

x += 1

dx += 2 \* ry \* ry

d1 += dx + (ry \* ry)

else:

x += 1

y -= 1

dx += 2 \* ry \* ry

dy -= 2 \* rx \* rx

d1 += dx - dy + (ry \* ry)

# Decision parameter for region 2

d2 = ((ry \* ry) \* (x + 0.5) \* (x + 0.5)) + ((rx \* rx) \* (y - 1) \* (y - 1)) - (rx \* rx \* ry \* ry)

# Region 2

while y >= 0:

symmetric\_points.extend(plot\_ellipse\_points(x\_center, y\_center, x, y))

if d2 > 0:

y -= 1

dy -= 2 \* rx \* rx

d2 += (rx \* rx) - dy

else:

x += 1

y -= 1

dx += 2 \* ry \* ry

dy -= 2 \* rx \* rx

d2 += dx - dy + (rx \* rx)

return symmetric\_points

def draw\_ellipse(x\_center, y\_center, rx, ry):

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

symmetric\_points = midpoint\_ellipse(x\_center, y\_center, rx, ry)

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

plt.grid(True)

plt.xlim(x\_center - rx - 1, x\_center + rx + 1)

plt.ylim(y\_center - ry - 1, y\_center + ry + 1)

plt.show()

return symmetric\_points

# Example usage:

x\_center = 0

y\_center = 0

rx = 15

ry = 10

# Draw the ellipse and get the symmetric points

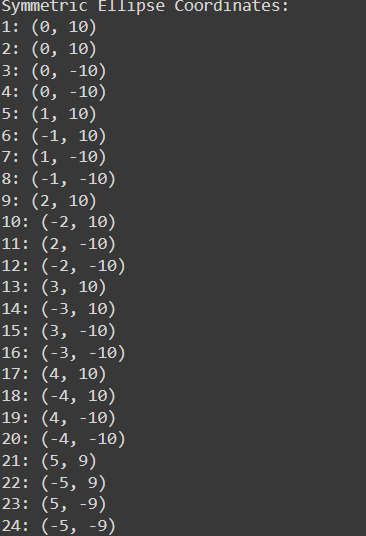
symmetric\_points = draw\_ellipse(x\_center, y\_center, rx, ry)

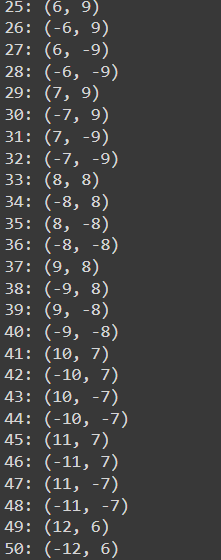
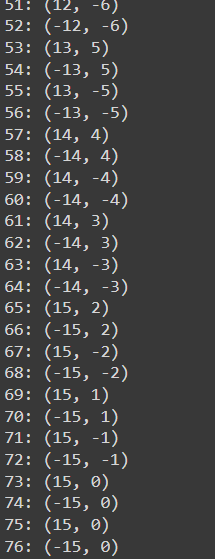
# Display the symmetric ellipse coordinates

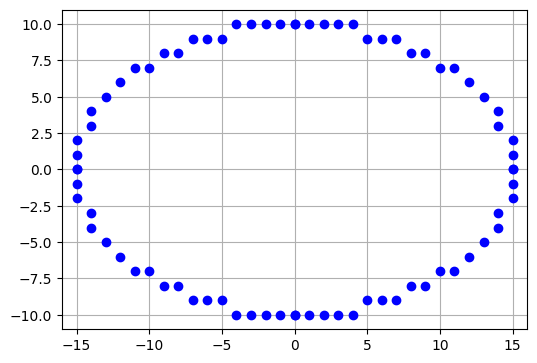
print("Symmetric Ellipse Coordinates:")

for i, point in enumerate(symmetric\_points, start=1):

print(f"{i}: {point}")





**LAB-5**

**(Title:** **2D transformations**)

**Implement 2D transformations of a rectangle.  
Step By Step Procedural Algorithm  
1. Enter the choice for transformation.  
2. Perform the translation, rotation and scaling of 2D object.  
3. Get the needed parameters for the transformation from the user.  
4. Incase of rotation, object can be rotated about x or y axis.  
5. Display the transmitted object in the screen along with new generated coordinates.**

import math

import matplotlib.pyplot as plt

# Define a Point class to store x and y coordinates

class Point:

    def \_\_init\_\_(self, x, y):

        self.x = x

        self.y = y

# Function to display rectangle points and plot them

def display\_rectangle(rect, title="Rectangle"):

    x\_coords = [point.x for point in rect] + [rect[0].x]

    y\_coords = [point.y for point in rect] + [rect[0].y]

    plt.plot(x\_coords, y\_coords, marker='o')

    plt.fill(x\_coords, y\_coords, "b", alpha=0.2)

    plt.title(title)

    plt.xlim(-10, 10)

    plt.ylim(-10, 10)

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

    plt.grid(True)

    plt.show()

# Function for translation

def translate(point, tx, ty):

    point.x += tx

    point.y += ty

# Function for rotation around a point (cx, cy)

def rotate(point, angle, cx, cy):

    rad = math.radians(angle)  # Convert to radians

    x\_new = cx + (point.x - cx) \* math.cos(rad) - (point.y - cy) \* math.sin(rad)

    y\_new = cy + (point.x - cx) \* math.sin(rad) + (point.y - cy) \* math.cos(rad)

    point.x, point.y = x\_new, y\_new

# Function for scaling

def scale(point, sx, sy, cx, cy):

    point.x = cx + (point.x - cx) \* sx

    point.y = cy + (point.y - cy) \* sy

# Main program

if \_\_name\_\_ == "\_\_main\_\_":

    # Define the initial coordinates of the rectangle

    rect = [Point(0, 0), Point(4, 0), Point(4, 3), Point(0, 3)]

    print("Original Rectangle Coordinates:")

    for i, point in enumerate(rect):

        print(f"Point {i + 1}: ({point.x}, {point.y})")

    # Display original rectangle

    display\_rectangle(rect, title="Original Rectangle")

    # Menu for selecting the transformation

    print("\nChoose Transformation:")

    print("1. Translation\n2. Rotation\n3. Scaling")

    choice = int(input())

    if choice == 1:

        # Translation

        tx = float(input("Enter translation tx: "))

        ty = float(input("Enter translation ty: "))

        for point in rect:

            translate(point, tx, ty)

    elif choice == 2:

        # Rotation

        angle = float(input("Enter rotation angle (in degrees): "))

        cx = float(input("Enter center of rotation cx: "))

        cy = float(input("Enter center of rotation cy: "))

        for point in rect:

            rotate(point, angle, cx, cy)

    elif choice == 3:

        # Scaling

        sx = float(input("Enter scaling factor sx: "))

        sy = float(input("Enter scaling factor sy: "))

        cx = float(input("Enter center of scaling cx: "))

        cy = float(input("Enter center of scaling cy: "))

        for point in rect:

            scale(point, sx, sy, cx, cy)

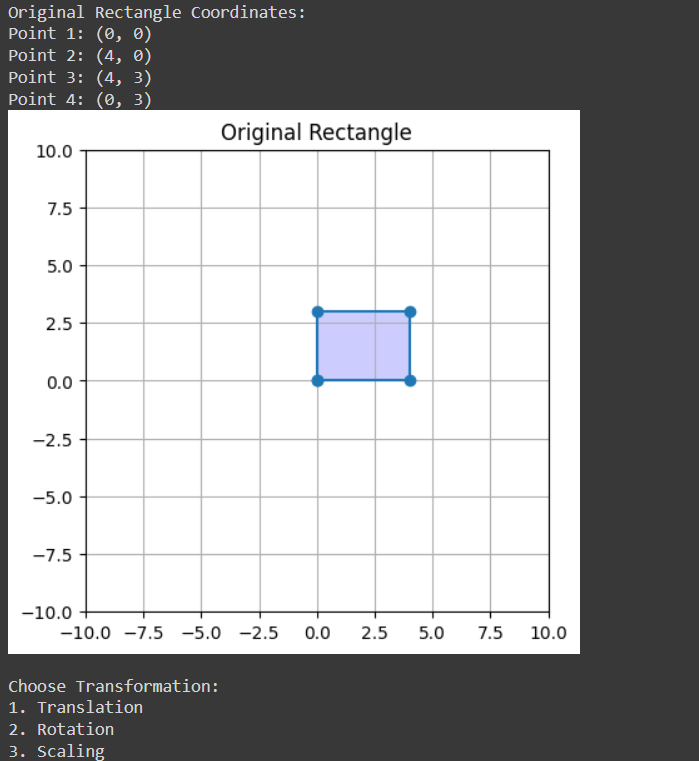
    # Display transformed rectangle

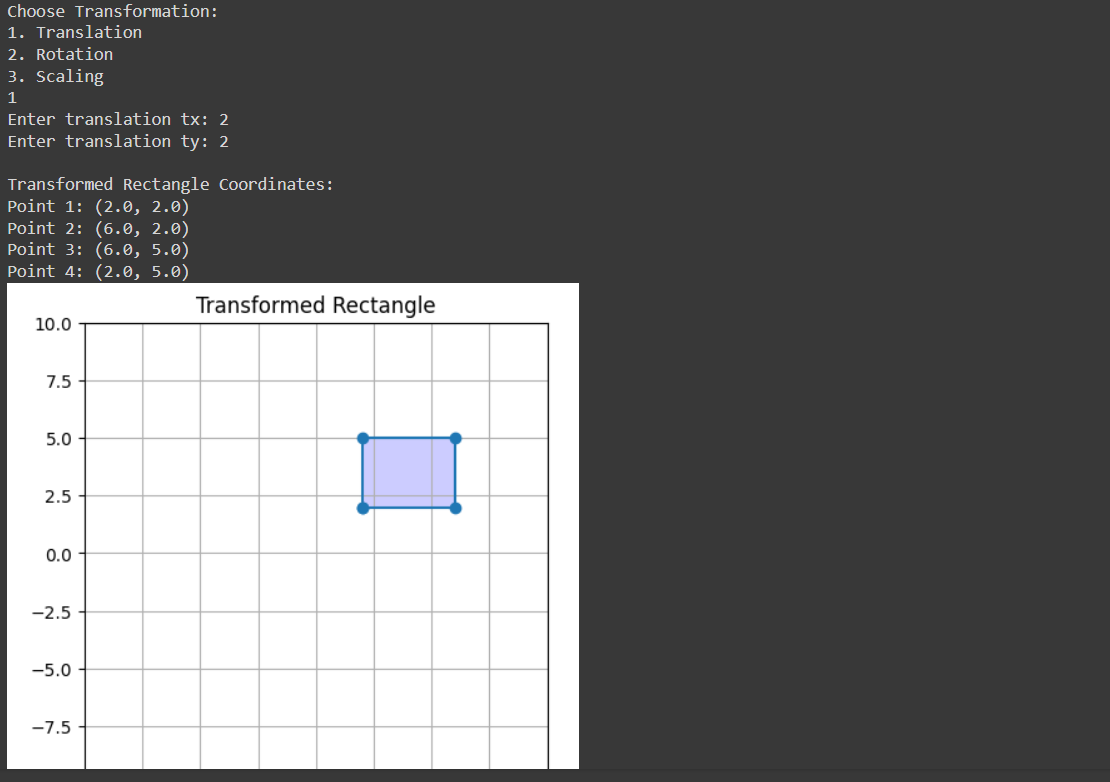
    print("\nTransformed Rectangle Coordinates:")

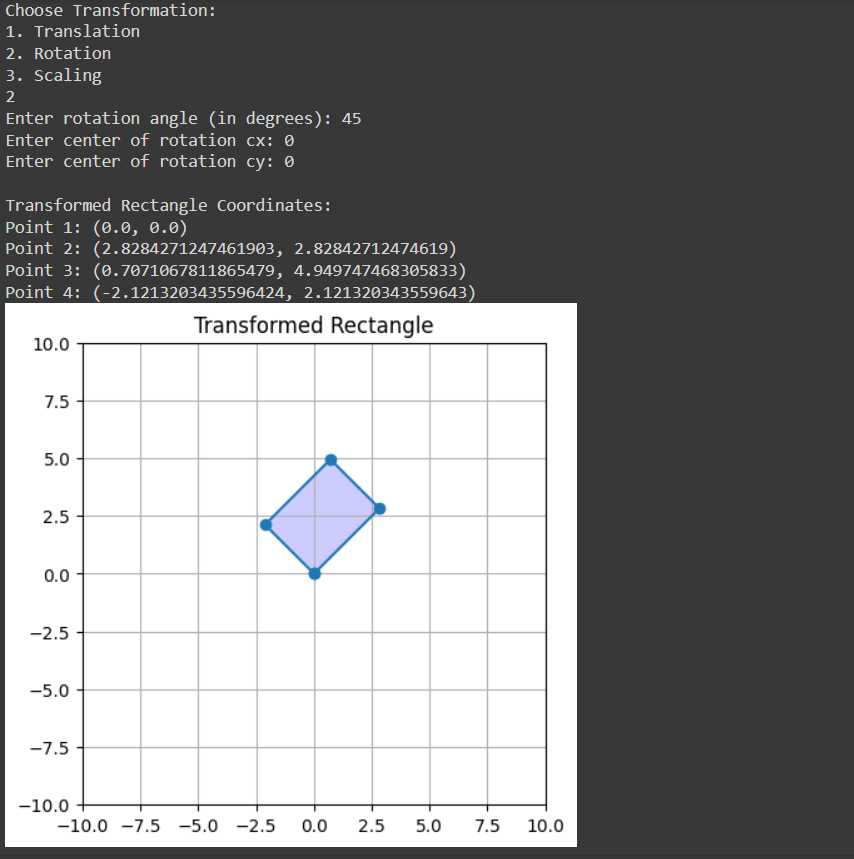
    for i, point in enumerate(rect):

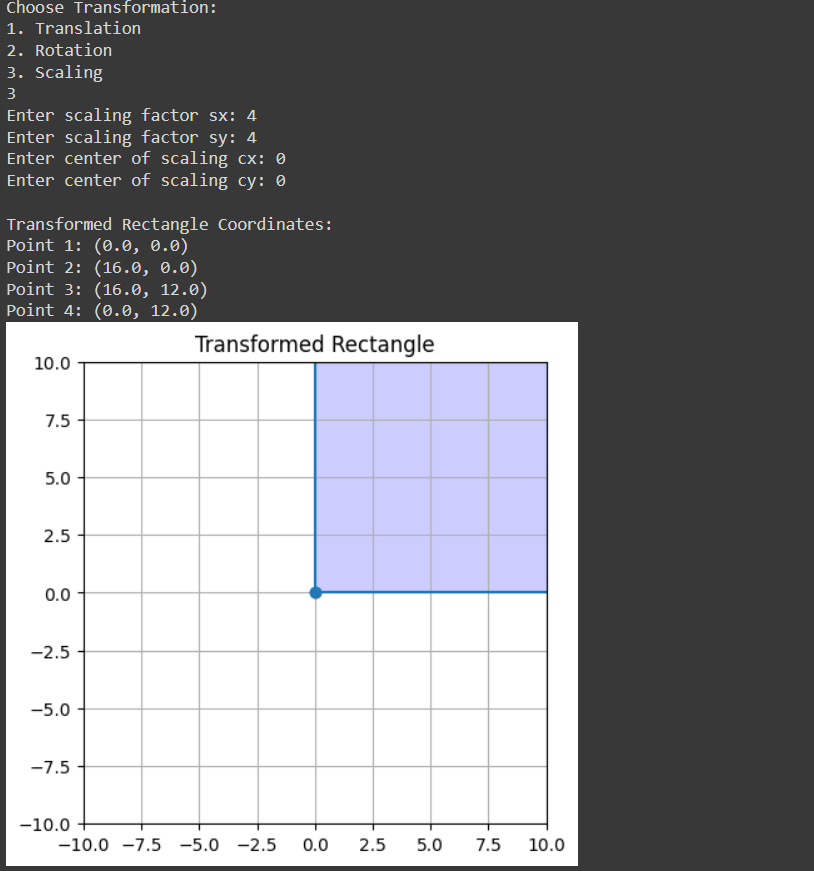
        print(f"Point {i + 1}: ({point.x}, {point.y})")

    display\_rectangle(rect, title="Transformed Rectangle")









**LAB-6**

**(Title:** **2D transformations**)

**Implement 2D reflection and shearing transformations on geometric shapes. Tasks:**

**Flip the point or shape across the x-axis.**

**Flip the point or shape across the y-axis.**

**Reflect the point or shape across a given line.**

**Skew the point or shape along the x-axis.**

**Skew the point or shape along the y-axis.**

import numpy as np

# Function to apply transformation

def apply\_transformation(point, transformation\_matrix):

    point = np.array([point[0], point[1], 1])

    result = np.dot(transformation\_matrix, point)

    return result[:2]

# Flip across X-axis

def flip\_across\_x\_axis(point):

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

                              [0, -1, 0],

                              [0, 0, 1]])

    return apply\_transformation(point, flip\_x\_matrix)

# Flip across Y-axis

def flip\_across\_y\_axis(point):

    flip\_y\_matrix = np.array([[-1, 0, 0],

                              [0, 1, 0],

                              [0, 0, 1]])

    return apply\_transformation(point, flip\_y\_matrix)

# Reflect across y=x

def reflect\_across\_line\_y\_equals\_x(point):

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

                               [1, 0, 0],

                               [0, 0, 1]])

    return apply\_transformation(point, reflect\_matrix)

# Skew along X-axis

def skew\_along\_x\_axis(point, skew\_factor):

    skew\_x\_matrix = np.array([[1, skew\_factor, 0],

                              [0, 1, 0],

                              [0, 0, 1]])

    return apply\_transformation(point, skew\_x\_matrix)

# Skew along Y-axis

def skew\_along\_y\_axis(point, skew\_factor):

    skew\_y\_matrix = np.array([[1, 0, 0],

                              [skew\_factor, 1, 0],

                              [0, 0, 1]])

    return apply\_transformation(point, skew\_y\_matrix)

# Example usage:

point = (2, 3)

print("Original point:", point)

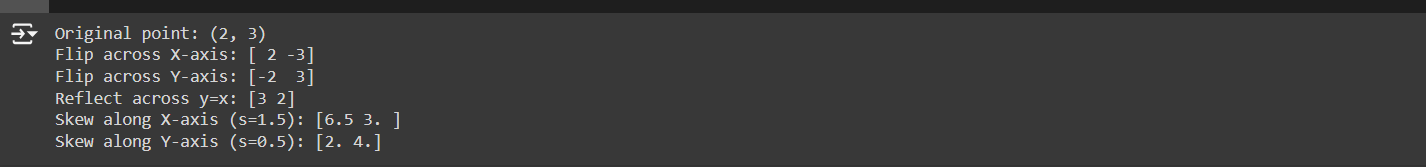
print("Flip across X-axis:", flip\_across\_x\_axis(point))

print("Flip across Y-axis:", flip\_across\_y\_axis(point))

print("Reflect across y=x:", reflect\_across\_line\_y\_equals\_x(point))

print("Skew along X-axis (s=1.5):", skew\_along\_x\_axis(point, 1.5))

print("Skew along Y-axis (s=0.5):", skew\_along\_y\_axis(point, 0.5))



**LAB-7**

**(Title:** **Cohen Sutherland clipping algorithm**)

**AIM:  
To write a program to implement the line clipping using Cohen Sutherland clipping algorithm.   
ALGORITHM:   
1. Get the clip window coordinates.   
2. Get the line end points.   
3. Draw the window and the line.   
4. Remove the line points which are plotted in outside the window.   
5. Draw the window with clipped line.**

import matplotlib.pyplot as plt

# Define region codes

INSIDE = 0 # 0000

LEFT = 1 # 0001

RIGHT = 2 # 0010

BOTTOM = 4 # 0100

TOP = 8 # 1000

# Define the clipping window boundaries

x\_min, y\_min = 100, 100

x\_max, y\_max = 300, 300

# Function to compute the region code of a point

def compute\_code(x, y):

code = INSIDE

if x < x\_min: # To the left of rectangle

code |= LEFT

elif x > x\_max: # To the right of rectangle

code |= RIGHT

if y < y\_min: # Below the rectangle

code |= BOTTOM

elif y > y\_max: # Above the rectangle

code |= TOP

return code

# Cohen-Sutherland clipping algorithm

def cohen\_sutherland\_clip(x1, y1, x2, y2):

code1 = compute\_code(x1, y1)

code2 = compute\_code(x2, y2)

accept = False

while True:

if code1 == 0 and code2 == 0:

accept = True

break

elif code1 & code2 != 0:

break

else:

x, y = 1.0, 1.0

code\_out = code1 if code1 != 0 else 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, y1 = x, y

code1 = compute\_code(x1, y1)

else:

x2, y2 = x, y

code2 = compute\_code(x2, y2)

if accept:

plt.plot([x1, x2], [y1, y2], color="green", linewidth=2)

else:

print("Line rejected.")

# Define the line end points

x1, y1 = 50, 150

x2, y2 = 350, 250

# 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")

# Plot the original line

plt.plot([x1, x2], [y1, y2], color="red", linestyle="--")

# Clip the line

cohen\_sutherland\_clip(x1, y1, x2, y2)

# Set up the plot

plt.xlim(0, 400)

plt.ylim(0, 400)

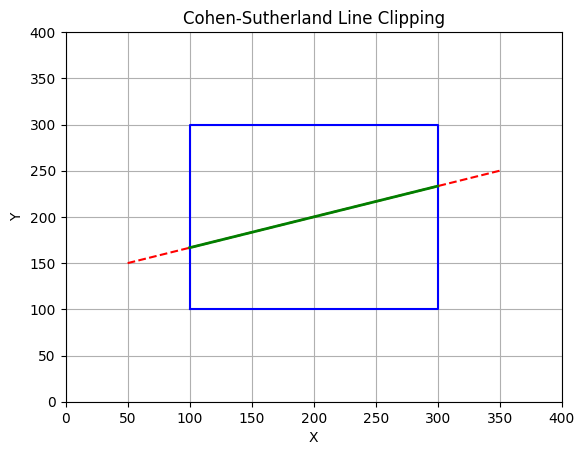
plt.title("Cohen-Sutherland Line Clipping")

plt.xlabel("X")

plt.ylabel("Y")

plt.grid()

plt.show()



**LAB-8**

**AIM:  
To write a program to implement the line clipping using liang Barsky line clipping algorithm.   
ALGORITHM:   
1. Get the clip window coordinates.   
2. Get the line end points.   
3. Draw the window and the line.   
4. Remove the line points which are plotted in outside the window.   
5. Draw the window with clipped line.**

import matplotlib.pyplot as plt

# Function to implement Liang-Barsky line clipping algorithm

def liang\_barsky(x\_min, y\_min, x\_max, y\_max, x1, y1, x2, y2):

    dx = x2 - x1

    dy = y2 - y1

    p = [-dx, dx, -dy, dy]

    q = [x1 - x\_min, x\_max - x1, y1 - y\_min, y\_max - y1]

    t\_enter = 0.0

    t\_exit = 1.0

    for i in range(4):

        if p[i] == 0:  # Check if line is parallel to the clipping boundary

            if q[i] < 0:

                return None  # Line is outside and parallel, so completely discarded

        else:

            t = q[i] / p[i]

            if p[i] < 0:

                if t > t\_enter:

                    t\_enter = t

            else:

                if t < t\_exit:

                    t\_exit = t

    if t\_enter > t\_exit:

        return None  # Line is completely outside

    x1\_clip = x1 + t\_enter \* dx

    y1\_clip = y1 + t\_enter \* dy

    x2\_clip = x1 + t\_exit \* dx

    y2\_clip = y1 + t\_exit \* dy

    return x1\_clip, y1\_clip, x2\_clip, y2\_clip

# Define the clipping window

x\_min, y\_min = 20, 20

x\_max, y\_max = 80, 80

# Define the line (starting and ending points) - you can adjust these coordinates

x1, y1 = 10, 30

x2, y2 = 90, 60

# Apply the Liang-Barsky algorithm to clip the line

clipped\_line = liang\_barsky(x\_min, y\_min, x\_max, y\_max, x1, y1, x2, y2)

# Plotting

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

# 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], 'b', label='Clipping Window')

if clipped\_line is not None:

    x1\_clip, y1\_clip, x2\_clip, y2\_clip = clipped\_line

    # Plot the original line

    plt.plot([x1, x2], [y1, y2], 'r', label='Original Line')

    # Plot the clipped line

    plt.plot([x1\_clip, x2\_clip], [y1\_clip, y2\_clip], 'g', label='Clipped Line')

    plt.title('Liang-Barsky Line Clipping Algorithm')

    plt.legend()

else:

    # The line is completely outside or parallel, so just plot the window

    plt.title('Line is outside the clipping window')

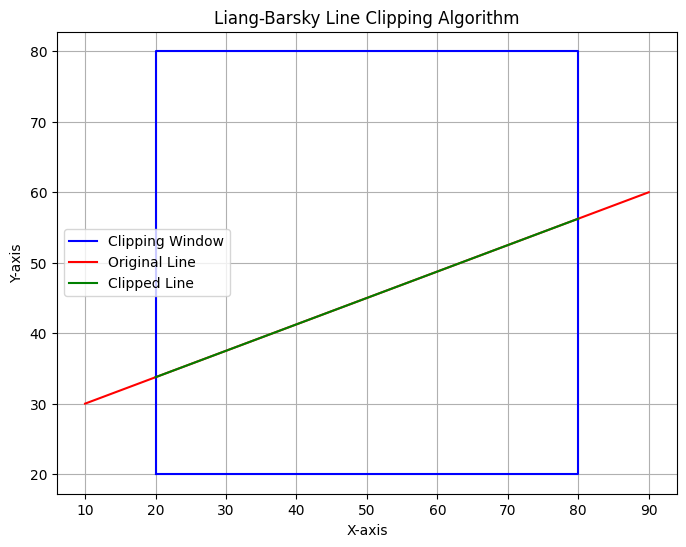
plt.xlabel('X-axis')

plt.ylabel('Y-axis')

plt.grid()

plt.axis('equal')

plt.show()



**AIM:  
To write a program to implement the polygon clipping using Sutherland Hodgeman polygon clipping algorithm.**

import numpy as np

import matplotlib.pyplot as plt

# Maximum number of points in a polygon

MAX\_POINTS = 20

# Function to return the x-value of the intersection point of two lines

def x\_intersect(x1, y1, x2, y2, x3, y3, x4, y4):

    num = (x1\*y2 - y1\*x2) \* (x3 - x4) - (x1 - x2) \* (x3\*y4 - y3\*x4)

    den = (x1 - x2) \* (y3 - y4) - (y1 - y2) \* (x3 - x4)

    return num / den

# Function to return the y-value of the intersection point of two lines

def y\_intersect(x1, y1, x2, y2, x3, y3, x4, y4):

    num = (x1\*y2 - y1\*x2) \* (y3 - y4) - (y1 - y2) \* (x3\*y4 - y3\*x4)

    den = (x1 - x2) \* (y3 - y4) - (y1 - y2) \* (x3 - x4)

    return num / den

# Function to clip all edges with respect to one clip edge of the clipping area

def clip(poly\_points, poly\_size, x1, y1, x2, y2):

    new\_points = np.zeros((MAX\_POINTS, 2), dtype=int)

    new\_poly\_size = 0

    for i in range(poly\_size):

        k = (i + 1) % poly\_size

        ix, iy = poly\_points[i]

        kx, ky = poly\_points[k]

        # Calculating position of points w.r.t. clipper line

        i\_pos = (x2 - x1) \* (iy - y1) - (y2 - y1) \* (ix - x1)

        k\_pos = (x2 - x1) \* (ky - y1) - (y2 - y1) \* (kx - x1)

        # Case 1: Both points are inside

        if i\_pos < 0 and k\_pos < 0:

            new\_points[new\_poly\_size] = [kx, ky]

            new\_poly\_size += 1

        # Case 2: First point is outside, second point is inside

        elif i\_pos >= 0 and k\_pos < 0:

            new\_points[new\_poly\_size] = [x\_intersect(x1, y1, x2, y2, ix, iy, kx, ky),

                                         y\_intersect(x1, y1, x2, y2, ix, iy, kx, ky)]

            new\_poly\_size += 1

            new\_points[new\_poly\_size] = [kx, ky]

            new\_poly\_size += 1

        # Case 3: First point is inside, second point is outside

        elif i\_pos < 0 and k\_pos >= 0:

            new\_points[new\_poly\_size] = [x\_intersect(x1, y1, x2, y2, ix, iy, kx, ky),

                                         y\_intersect(x1, y1, x2, y2, ix, iy, kx, ky)]

            new\_poly\_size += 1

    # Copying new points into a separate array

    clipped\_poly\_points = np.zeros((new\_poly\_size, 2), dtype=int)

    for i in range(new\_poly\_size):

        clipped\_poly\_points[i] = new\_points[i]

    return clipped\_poly\_points, new\_poly\_size

# Function to implement the Sutherland–Hodgman algorithm

def suthHodgClip(poly\_points, poly\_size, clipper\_points, clipper\_size):

    for i in range(clipper\_size):

        k = (i + 1) % clipper\_size

        poly\_points, poly\_size = clip(poly\_points, poly\_size, clipper\_points[i][0],

                                      clipper\_points[i][1], clipper\_points[k][0],

                                      clipper\_points[k][1])

    return poly\_points, poly\_size

# Function to plot the polygons

def plot\_polygon(points, color, label):

    points = np.vstack([points, points[0]])  # Closing the polygon

    plt.plot(points[:, 0], points[:, 1], color=color, label=label)

    plt.fill(points[:, 0], points[:, 1], alpha=0.3, color=color)

# Driver code

if \_\_name\_\_ == "\_\_main\_\_":

    # Defining the polygon vertices

    poly\_points = np.array([[100, 150], [200, 250], [300, 200]])

    poly\_size = len(poly\_points)

    # Defining the clipper polygon vertices

    clipper\_points = np.array([[150, 150], [150, 200], [200, 200], [200, 150]])

    clipper\_size = len(clipper\_points)

    # Plotting the original polygon and clipper

    plt.figure()

    plot\_polygon(poly\_points, 'blue', 'Original Polygon')

    plot\_polygon(clipper\_points, 'red', 'Clipper Polygon')

    # Clipping the polygon

    clipped\_poly\_points, clipped\_poly\_size = suthHodgClip(poly\_points, poly\_size, clipper\_points, clipper\_size)

    # Plotting the clipped polygon

    plot\_polygon(clipped\_poly\_points, 'green', 'Clipped Polygon')

    # Displaying the plot

    plt.legend()

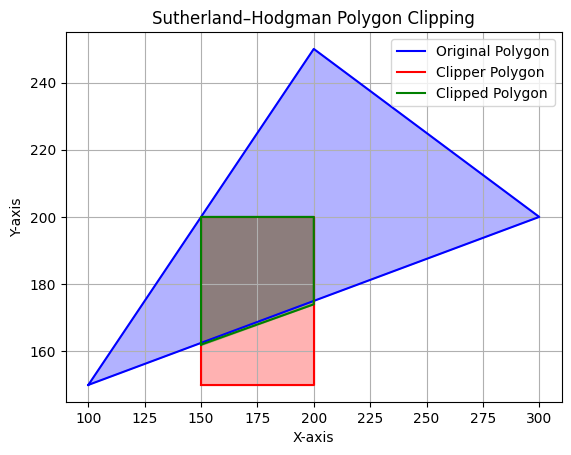
    plt.title('Sutherland–Hodgman Polygon Clipping')

    plt.xlabel('X-axis')

    plt.ylabel('Y-axis')

    plt.grid(True)

    plt.show()



**LAB-9**

**AIM:  
Implement 3D transformations of a object.  
Step By Step Procedural Algorithm  
1. Enter the choice for transformation.  
2. Perform the translation, rotation and scaling of 3d object.  
3. Get the needed parameters for the transformation from the user.  
4. Incase of rotation, object can be rotated about x or y axis.  
5. Display the transmitted object in the screen along with new generated coordinates.**

import numpy as np

import math

import matplotlib.pyplot as plt

from mpl\_toolkits.mplot3d.art3d import Poly3DCollection

import ipywidgets as widgets

from IPython.display import display, clear\_output

rectangle\_points = 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]

])

def translate(points, Tx, Ty, Tz):

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

                                   [0, 1, 0, Ty],

                                   [0, 0, 1, Tz],

                                   [0, 0, 0, 1]])

    return apply\_transformation(points, translation\_matrix)

def scale(points, Sx, Sy, Sz):

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

                               [0, Sy, 0, 0],

                               [0, 0, Sz, 0],

                               [0, 0, 0, 1]])

    return apply\_transformation(points, scaling\_matrix)

def rotate\_x(points, theta):

    theta\_rad = math.radians(theta)

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

                                  [0, math.cos(theta\_rad), -math.sin(theta\_rad), 0],

                                  [0, math.sin(theta\_rad), math.cos(theta\_rad), 0],

                                  [0, 0, 0, 1]])

    return apply\_transformation(points, rotation\_matrix\_x)

def rotate\_y(points, theta):

    theta\_rad = math.radians(theta)

    rotation\_matrix\_y = np.array([[math.cos(theta\_rad), 0, math.sin(theta\_rad), 0],

                                  [0, 1, 0, 0],

                                  [-math.sin(theta\_rad), 0, math.cos(theta\_rad), 0],

                                  [0, 0, 0, 1]])

    return apply\_transformation(points, rotation\_matrix\_y)

def apply\_transformation(points, transformation\_matrix):

    homogeneous\_points = np.hstack((points, np.ones((points.shape[0], 1))))

    transformed\_points = homogeneous\_points.dot(transformation\_matrix.T)

    return transformed\_points[:, :3]

def display\_coordinates(points):

    print("Coordinates of the Rectangle:")

    for i, point in enumerate(points, start=1):

        print(f"Point {i}: {point}")

def plot\_rectangle(points, title="3D Rectangle"):

    fig = plt.figure()

    ax = fig.add\_subplot(111, projection='3d')

    ax.set\_title(title)

    verts = [[points[0], points[1], points[2], points[3]],

             [points[4], points[5], points[6], points[7]],

             [points[0], points[1], points[5], points[4]],

             [points[2], points[3], points[7], points[6]],

             [points[1], points[2], points[6], points[5]],

             [points[4], points[7], points[3], points[0]]]

    ax.add\_collection3d(Poly3DCollection(verts, color="cyan", edgecolor="black", alpha=0.3))

    max\_range = np.array([points[:, 0].max() - points[:, 0].min(),

                          points[:, 1].max() - points[:, 1].min(),

                          points[:, 2].max() - points[:, 2].min()]).max() / 2.0

    mid\_x = (points[:, 0].max() + points[:, 0].min()) \* 0.5

    mid\_y = (points[:, 1].max() + points[:, 1].min()) \* 0.5

    mid\_z = (points[:, 2].max() + points[:, 2].min()) \* 0.5

    ax.set\_xlim(mid\_x - max\_range, mid\_x + max\_range)

    ax.set\_ylim(mid\_y - max\_range, mid\_y + max\_range)

    ax.set\_zlim(mid\_z - max\_range, mid\_z + max\_range)

    plt.show()

def on\_translate(Tx, Ty, Tz):

    clear\_output(wait=True)

    Tx, Ty, Tz = float(Tx), float(Ty), float(Tz)

    transformed\_points = translate(rectangle\_points, Tx, Ty, Tz)

    display\_coordinates(transformed\_points)

    plot\_rectangle(transformed\_points, "Translated Rectangle")

def on\_scale(Sx, Sy, Sz):

    clear\_output(wait=True)

    Sx, Sy, Sz = float(Sx), float(Sy), float(Sz)

    transformed\_points = scale(rectangle\_points, Sx, Sy, Sz)

    display\_coordinates(transformed\_points)

    plot\_rectangle(transformed\_points, "Scaled Rectangle")

def on\_rotate(axis, theta):

    clear\_output(wait=True)

    theta = float(theta)

    if axis == 'x':

        transformed\_points = rotate\_x(rectangle\_points, theta)

    elif axis == 'y':

        transformed\_points = rotate\_y(rectangle\_points, theta)

    display\_coordinates(transformed\_points)

    plot\_rectangle(transformed\_points, f"Rotated Rectangle ({axis.upper()}-axis)")

def interactive\_interface():

    print("Original Rectangle:")

    display\_coordinates(rectangle\_points)

    plot\_rectangle(rectangle\_points, "Initial Rectangle")

    Tx = widgets.Text(value="0", description="Tx:")

    Ty = widgets.Text(value="0", description="Ty:")

    Tz = widgets.Text(value="0", description="Tz:")

    display(widgets.interactive(on\_translate, Tx=Tx, Ty=Ty, Tz=Tz))

    Sx = widgets.Text(value="1", description="Sx:")

    Sy = widgets.Text(value="1", description="Sy:")

    Sz = widgets.Text(value="1", description="Sz:")

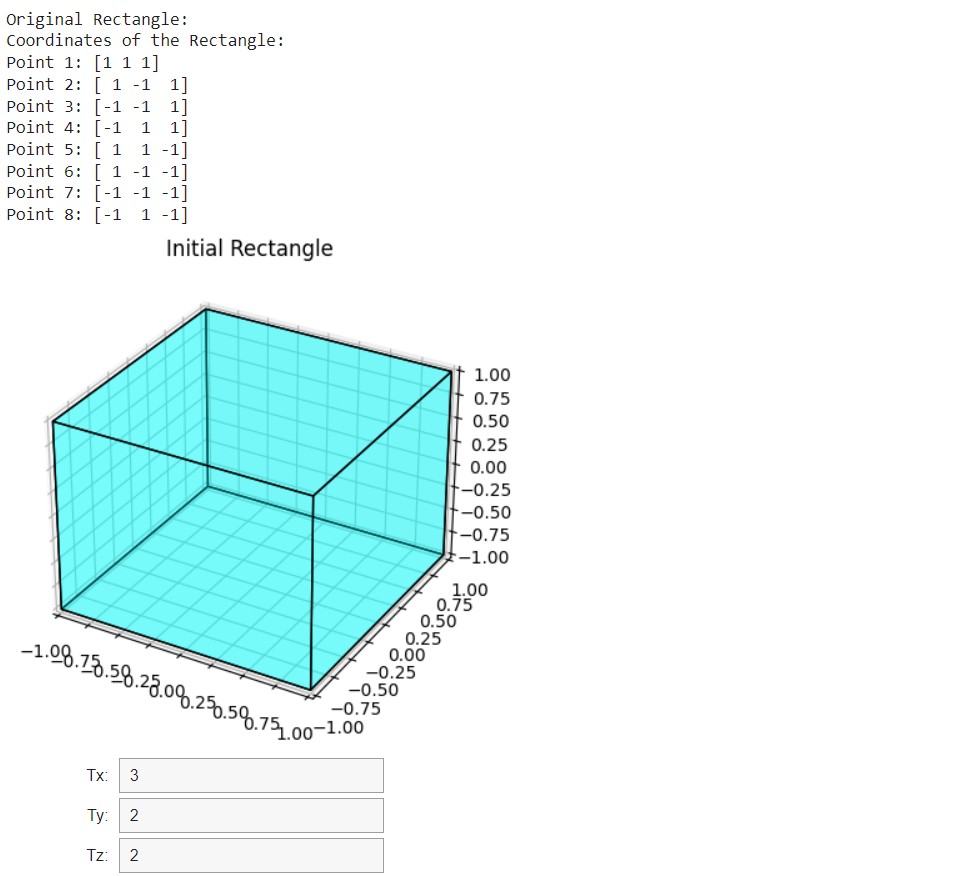
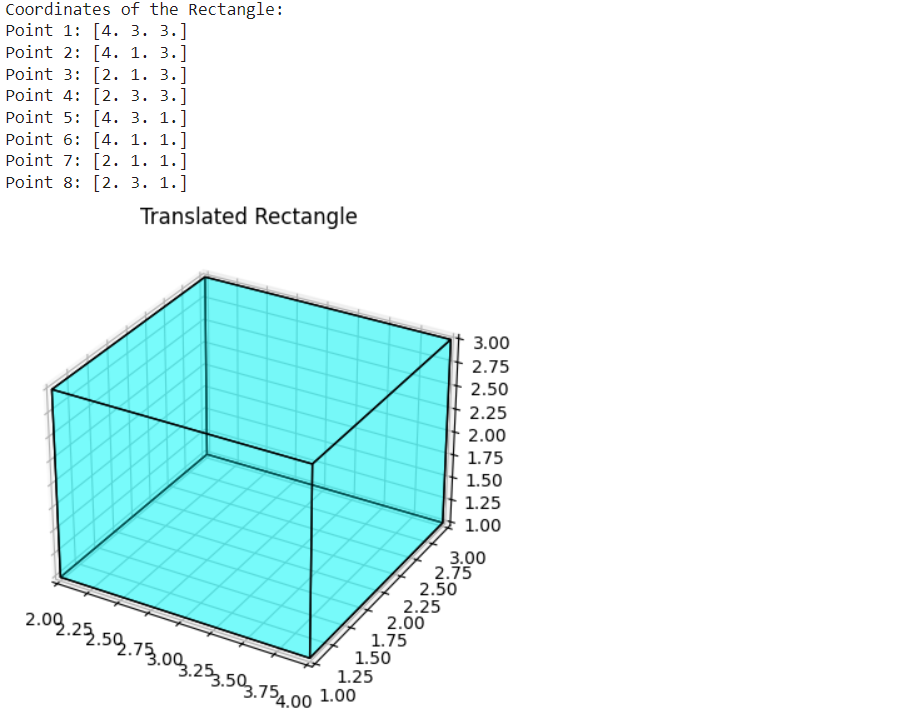
    display(widgets.interactive(on\_scale, Sx=Sx, Sy=Sy, Sz=Sz))

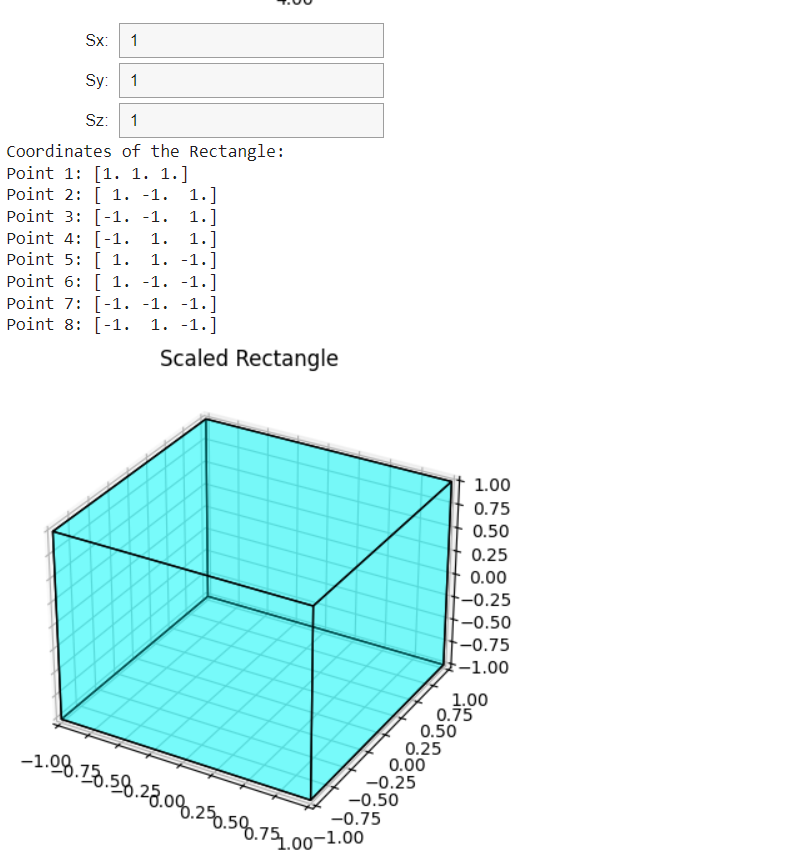
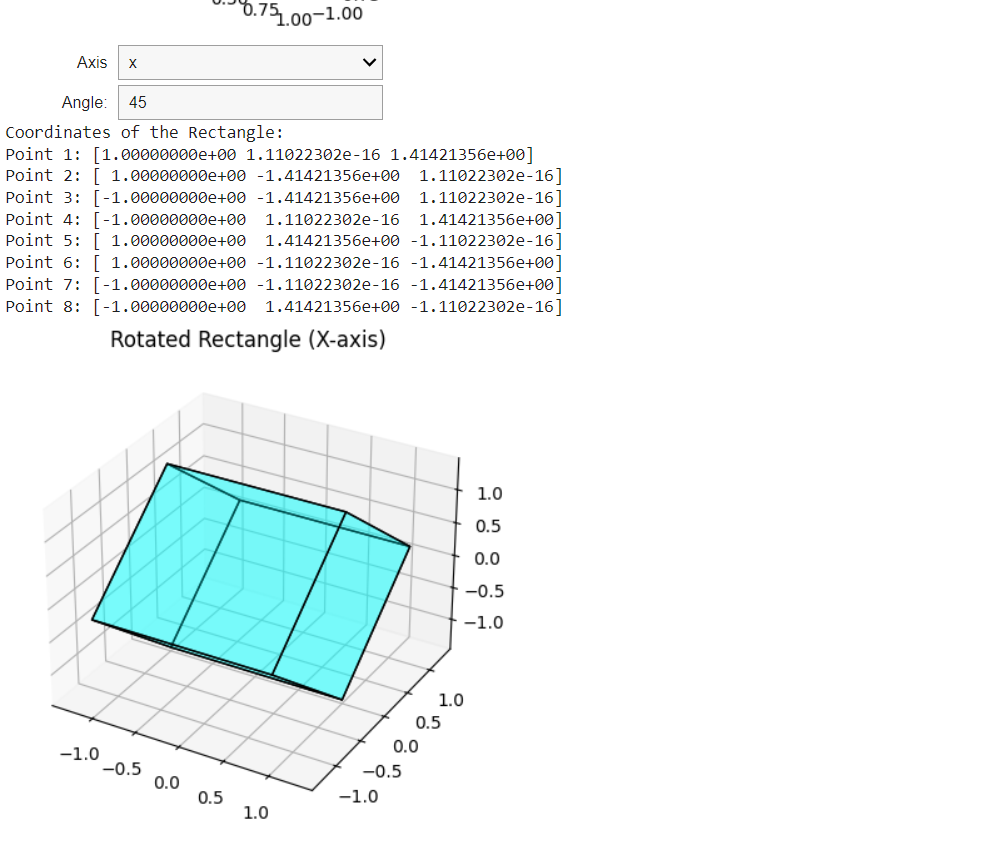
    axis = widgets.Dropdown(options=['x', 'y'], description='Axis')

    theta = widgets.Text(value="0", description="Angle:")

    display(widgets.interactive(on\_rotate, axis=axis, theta=theta))

interactive\_interface()

****



**Internal Evaluation**

**Implement the Cohen-Sutherland line clipping algorithm to handle multiple lines and clip them against an arbitrary clipping window. Your implementation should visualize:The original lines.The clipping window.The clipped segments (if accepted).The rejected lines.Your implementation must:**

* Use the Cohen-Sutherland algorithm to determine whether each line is .
* Calculate the intersection points for clipped lines.
* Display both the original and processed lines graphically.

**Enhance the implementation to:**Allow the user to input additional lines interactively.  
Handle edge cases such as:

* where the endpoints are the same.

import numpy as np

import pandas as pd

import matplotlib.pyplot as plt

INSIDE = 0

LEFT = 1

RIGHT = 2

BOTTOM = 4

TOP = 8

clip\_window = {'xmin': 50, 'ymin': 50, 'xmax': 200, 'ymax': 200}

def calc\_code(x, y, window):

    code = INSIDE

    if x < window['xmin']:

        code |= LEFT

    elif x > window['xmax']:

        code |= RIGHT

    if y < window['ymin']:

        code |= BOTTOM

    elif y > window['ymax']:

        code |= TOP

    return code

def vis(code,arr):

  if arr.find(code)==arr.end():

    return false

  return true

def cohen(x1, y1, x2, y2, window):

    code1 = calc\_code(x1, y1, window)

    code2 = calc\_code(x2, y2, window)

    accept = False

    while True:

        if code1 == 0 and code2 == 0:

            accept = True

            break

        elif code1 & code2 != 0:

            break

        else:

            code\_out = code1 if code1 != 0 else code2

            x, y = 0, 0

            if code\_out & TOP:

                x = x1 + (x2 - x1) \* (window['ymax'] - y1) / (y2 - y1)

                y = window['ymax']

            elif code\_out & BOTTOM:

                x = x1 + (x2 - x1) \* (window['ymin'] - y1) / (y2 - y1)

                y = window['ymin']

            elif code\_out & RIGHT:

                y = y1 + (y2 - y1) \* (window['xmax'] - x1) / (x2 - x1)

                x = window['xmax']

            elif code\_out & LEFT:

                y = y1 + (y2 - y1) \* (window['xmin'] - x1) / (x2 - x1)

                x = window['xmin']

            if code\_out == code1:

                x1, y1 = x, y

                code1 = calc\_code(x1, y1, window)

            else:

                x2, y2 = x, y

                code2 = calc\_code(x2, y2, window)

    if accept:

        return (x1, y1, x2, y2)

    else:

        return None

def visualize(lines, clipped\_lines, window):

    fig, ax = plt.subplots()

    rect = plt.Rectangle((window['xmin'], window['ymin']),window['xmax'] - window['xmin'],window['ymax'] - window['ymin'],edgecolor='black', facecolor='none', linewidth=2)

    ax.add\_patch(rect)

    for line in lines:

        x1, y1, x2, y2 = line

        ax.plot([x1, x2], [y1, y2], color='blue', linestyle='--', label='Original Line' if line == lines[0] else "")

    for line in clipped\_lines:

        if line:

            x1, y1, x2, y2 = line

            ax.plot([x1, x2], [y1, y2], color='green', label='Clipped Line' if line == clipped\_lines[0] else "")

    ax.legend(loc='upper right')

    ax.set\_xlim(0, 300)

    ax.set\_ylim(0, 300)

    ax.set\_aspect('equal')

    ax.set\_title('Cohen-Sutherland Line Clipping')

    plt.show()

def main():

    lines=[(30, 20, 180, 150), (100, 300, 150, 100), (60, 60, 260, 60), (150, 150, 150, 150)]

    clipped\_lines=[]

    for line in lines:

        result=cohen(\*line, clip\_window)

        clipped\_lines.append(result)

    visualize(lines, clipped\_lines, clip\_window)

if \_\_name\_\_ == "\_\_main\_\_":

    main()

