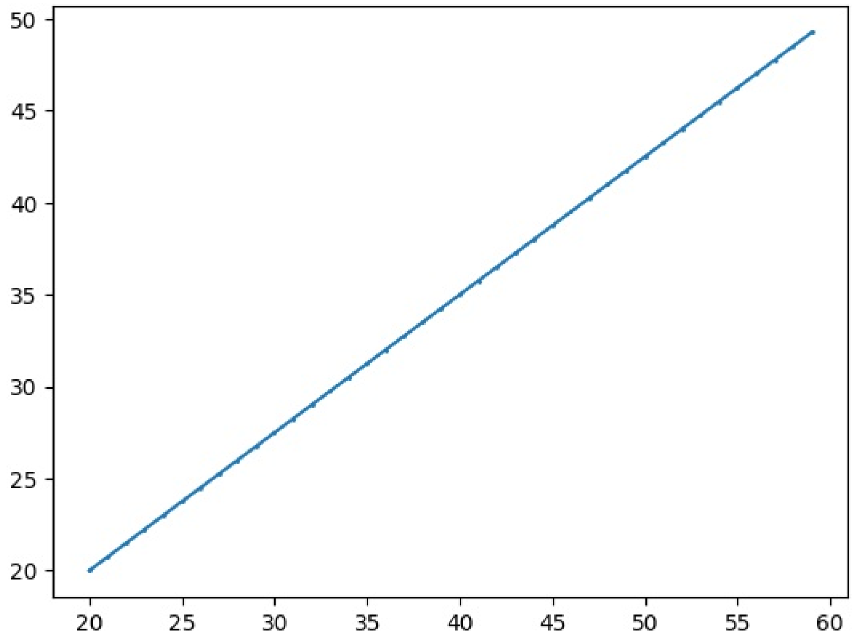
LAB-1

(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

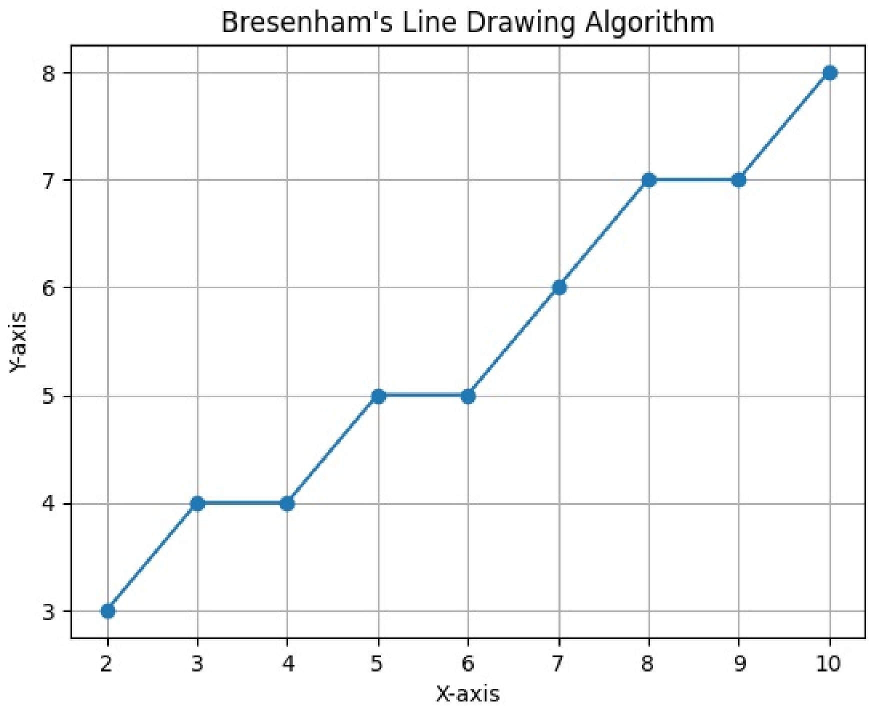
(Title: Bresenham’s drawing Algorithm)

- **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()
```

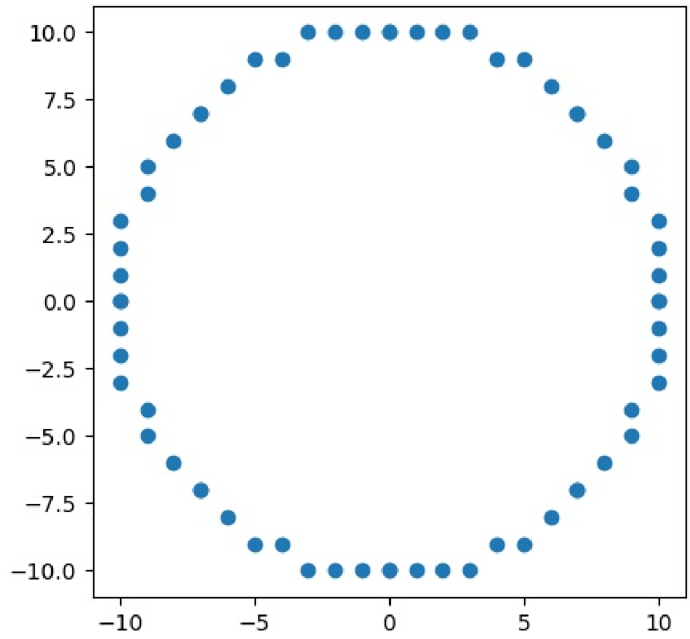
```
(2, 3)
(3, 4)
(4, 4)
(5, 5)
(6, 5)
(7, 6)
(8, 7)
(9, 7)
(10, 8)
```



- **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)
```



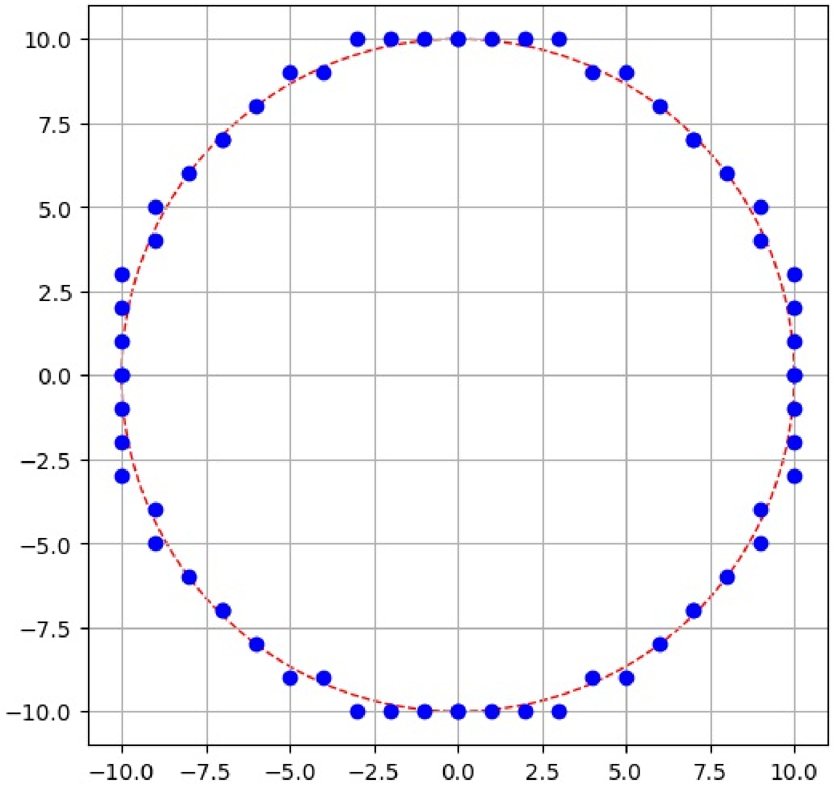
LAB-3

(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
```

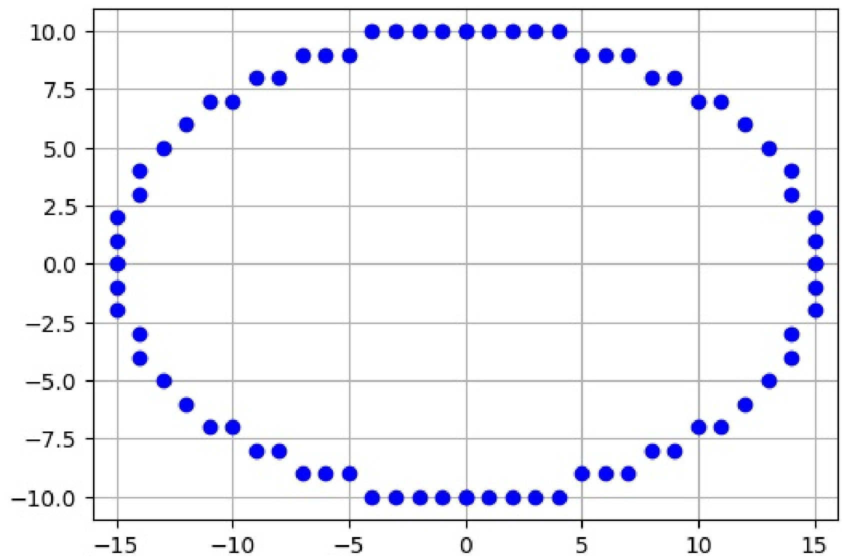
```
while dx < dy:
symmetric_points.extend(plot_ellipse_points(x_center, y_center, x, y))
if d1 < 0:
x += 1
dx += 2 * ry * ry
d1 += dx + (ry * ry)
else:
x += 1
y -= 1
dx += 2 * ry * ry
dy -= 2 * rx * rx
d1 += dx - dy + (ry * ry)
# Decision parameter for region 2
d2 = ((ry * ry) * (x + 0.5) * (x + 0.5)) + ((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}")
```

```
Symmetric Ellipse Coordinates:
1: (0, 10)
2: (0, 10)
3: (0, -10)
4: (0, -10)
5: (1, 10)
6: (-1, 10)
7: (1, -10)
8: (-1, -10)
9: (2, 10)
10: (-2, 10)
11: (2, -10)
12: (-2, -10)
13: (3, 10)
14: (-3, 10)
15: (3, -10)
16: (-3, -10)
17: (4, 10)
18: (-4, 10)
19: (4, -10)
20: (-4, -10)
21: (5, 9)
22: (-5, 9)
23: (5, -9)
24: (-5, -9)
```

```
25: (6, 9)
26: (-6, 9)
27: (6, -9)
28: (-6, -9)
29: (7, 9)
30: (-7, 9)
31: (7, -9)
32: (-7, -9)
33: (8, 8)
34: (-8, 8)
35: (8, -8)
36: (-8, -8)
37: (9, 8)
38: (-9, 8)
39: (9, -8)
40: (-9, -8)
41: (10, 7)
42: (-10, 7)
43: (10, -7)
44: (-10, -7)
45: (11, 7)
46: (-11, 7)
47: (11, -7)
48: (-11, -7)
49: (12, 6)
50: (-12, 6)
```

```
51: (12, -6)
52: (-12, -6)
53: (13, 5)
54: (-13, 5)
55: (13, -5)
56: (-13, -5)
57: (14, 4)
58: (-14, 4)
59: (14, -4)
60: (-14, -4)
61: (14, 3)
62: (-14, 3)
63: (14, -3)
64: (-14, -3)
65: (15, 2)
66: (-15, 2)
67: (15, -2)
68: (-15, -2)
69: (15, 1)
70: (-15, 1)
71: (15, -1)
72: (-15, -1)
73: (15, 0)
74: (-15, 0)
75: (15, 0)
76: (-15, 0)
```



LAB-5

(Title: 2D transformations)

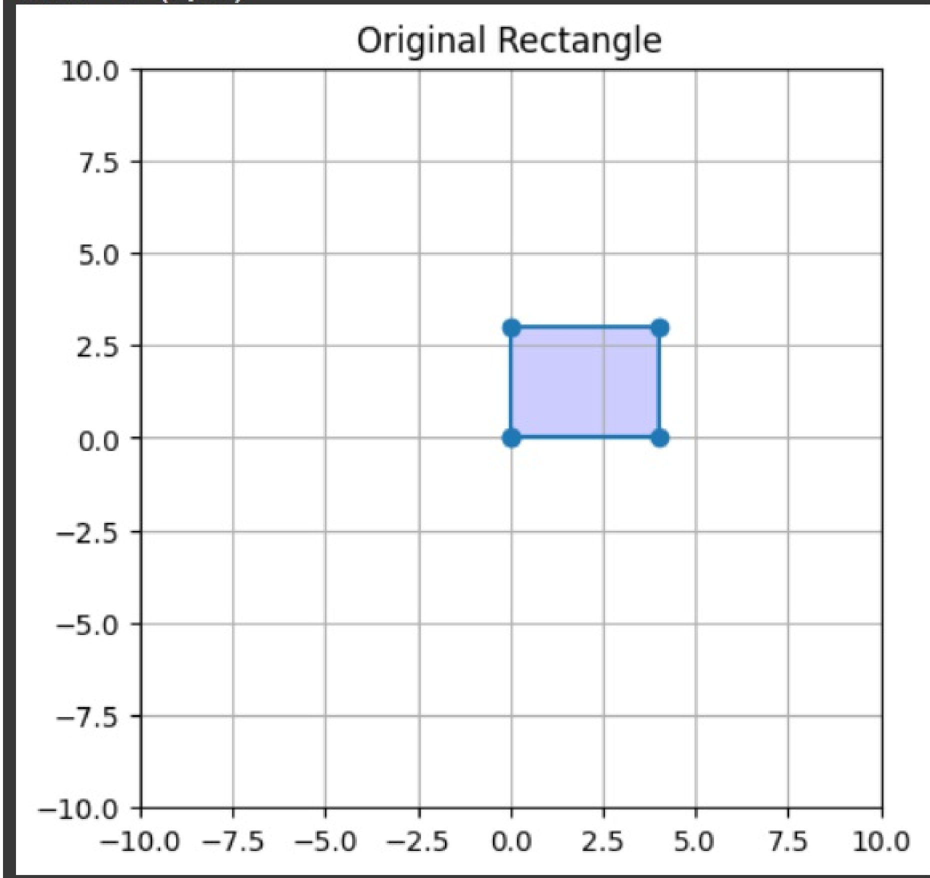
Implement 2D transformations of a rectangle.
Step By Step Procedural Algorithm
1. Enter the choice for transformation.
2. Perform the translation, rotation and scaling of 2D object.
3. Get the needed parameters for the transformation from the user.
4. Incase of rotation, object can be rotated about x 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:")
```

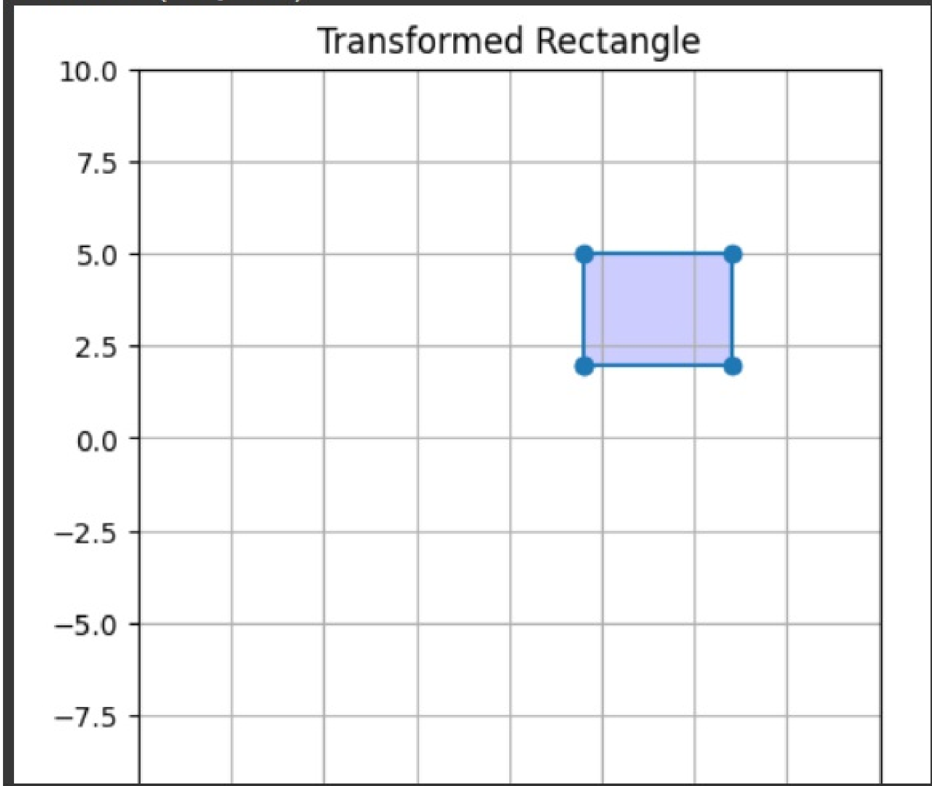
```
print("1. Translation\n2. Rotation\n3. Scaling")
choice = int(input())
if choice == 1:
    # Translation
    tx = float(input("Enter translation tx: "))
    ty = float(input("Enter translation ty: "))
    for point in rect:
        translate(point, tx, ty)
elif choice == 2:
    # Rotation
    angle = 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")
```

```
Original Rectangle Coordinates:
Point 1: (0, 0)
Point 2: (4, 0)
Point 3: (4, 3)
Point 4: (0, 3)
```

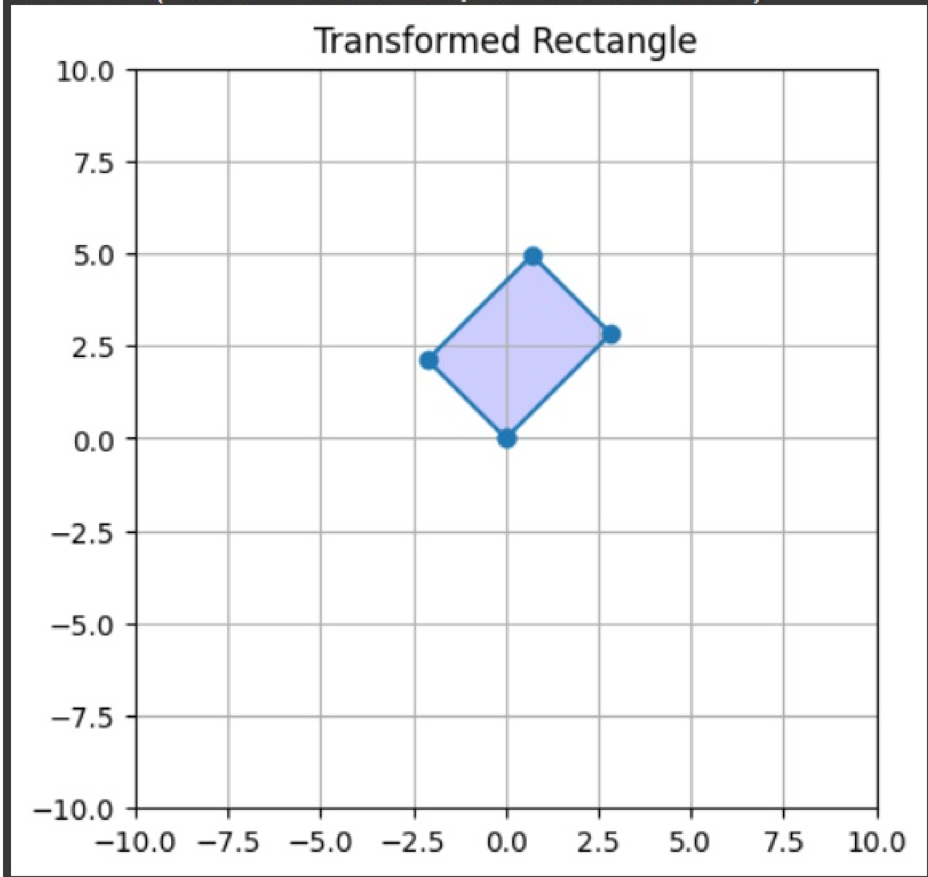



```
Choose Transformation:
1. Translation
2. Rotation
3. Scaling
```

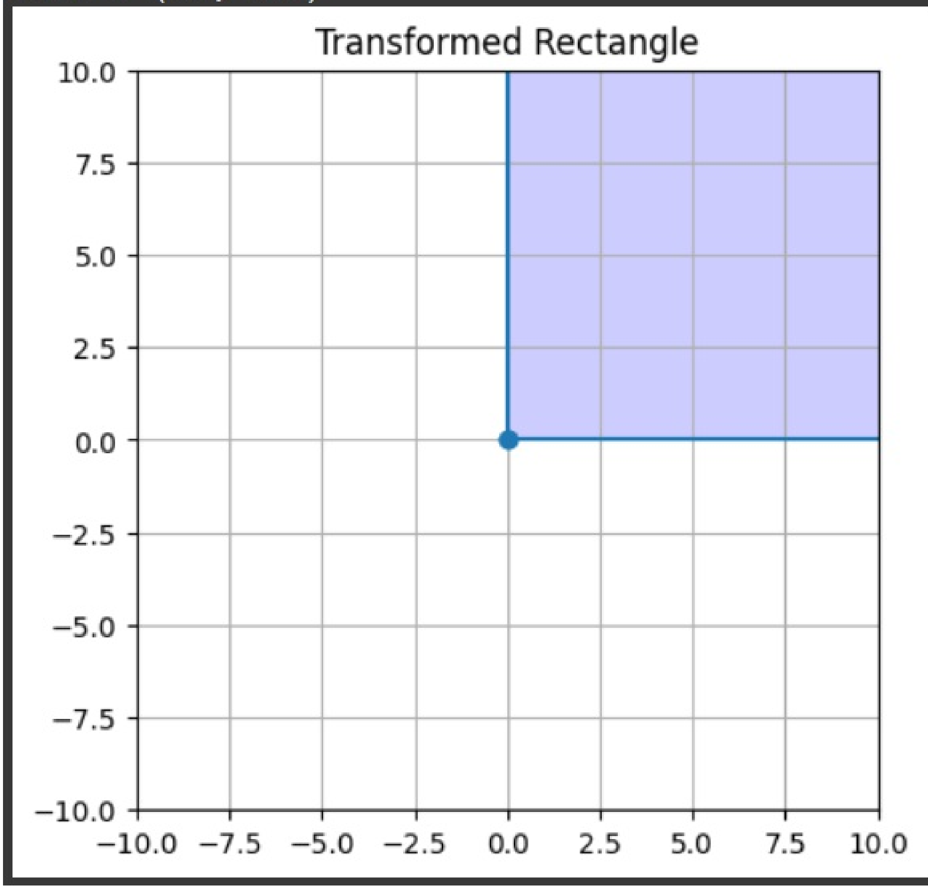
```
Choose Transformation:
1. Translation
2. Rotation
3. Scaling
1
Enter translation tx: 2
Enter translation ty: 2

Transformed Rectangle Coordinates:
Point 1: (2.0, 2.0)
Point 2: (6.0, 2.0)
Point 3: (6.0, 5.0)
Point 4: (2.0, 5.0)
```



```
Choose Transformation:
1. Translation
2. Rotation
3. Scaling
2
Enter rotation angle (in degrees): 45
Enter center of rotation cx: 0
Enter center of rotation cy: 0

Transformed Rectangle Coordinates:
Point 1: (0.0, 0.0)
Point 2: (2.8284271247461903, 2.82842712474619)
Point 3: (0.7071067811865479, 4.949747468305833)
Point 4: (-2.1213203435596424, 2.121320343559643)
```



```
Choose Transformation:
1. Translation
2. Rotation
3. Scaling
3
Enter scaling factor sx: 4
Enter scaling factor sy: 4
Enter center of scaling cx: 0
Enter center of scaling cy: 0

Transformed Rectangle Coordinates:
Point 1: (0.0, 0.0)
Point 2: (16.0, 0.0)
Point 3: (16.0, 12.0)
Point 4: (0.0, 12.0)
```



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))
```

```
Original point: (2, 3)
Flip across X-axis: [ 2 -3]
Flip across Y-axis: [-2  3]
Reflect across y=x: [3 2]
Skew along X-axis (s=1.5): [6.5 3. ]
Skew along Y-axis (s=0.5): [2. 4.]
```

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)

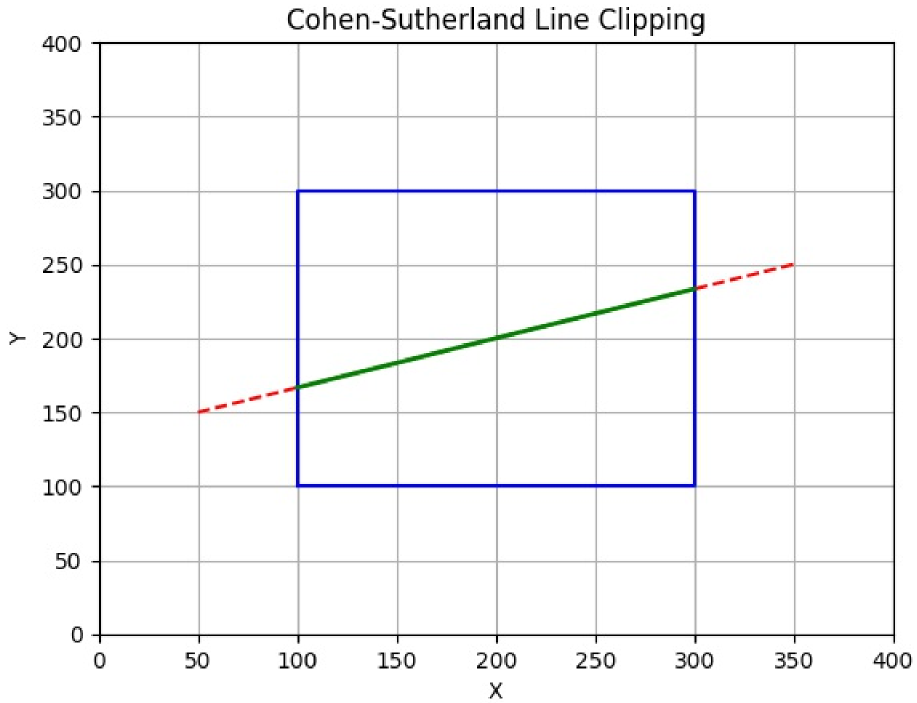
plt.title("Cohen-Sutherland Line Clipping")

plt.xlabel("X")

plt.ylabel("Y")

plt.grid()

plt.show()



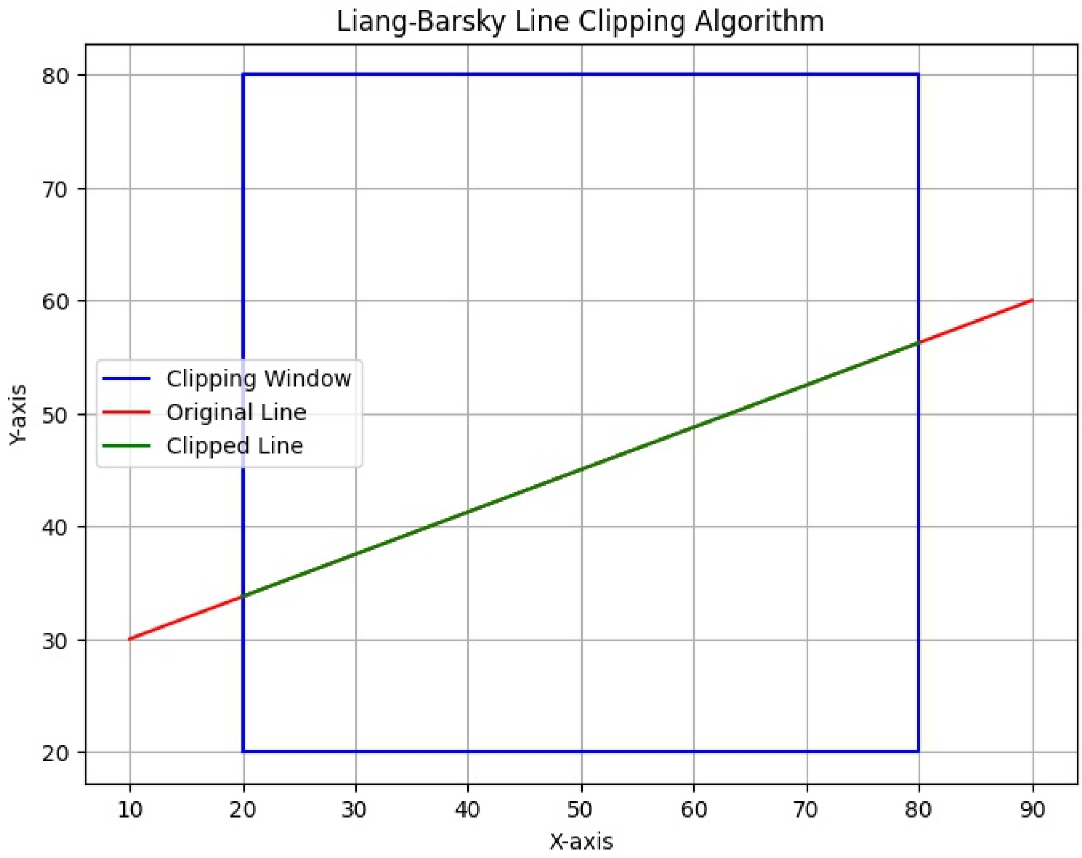
LAB-8

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()
```

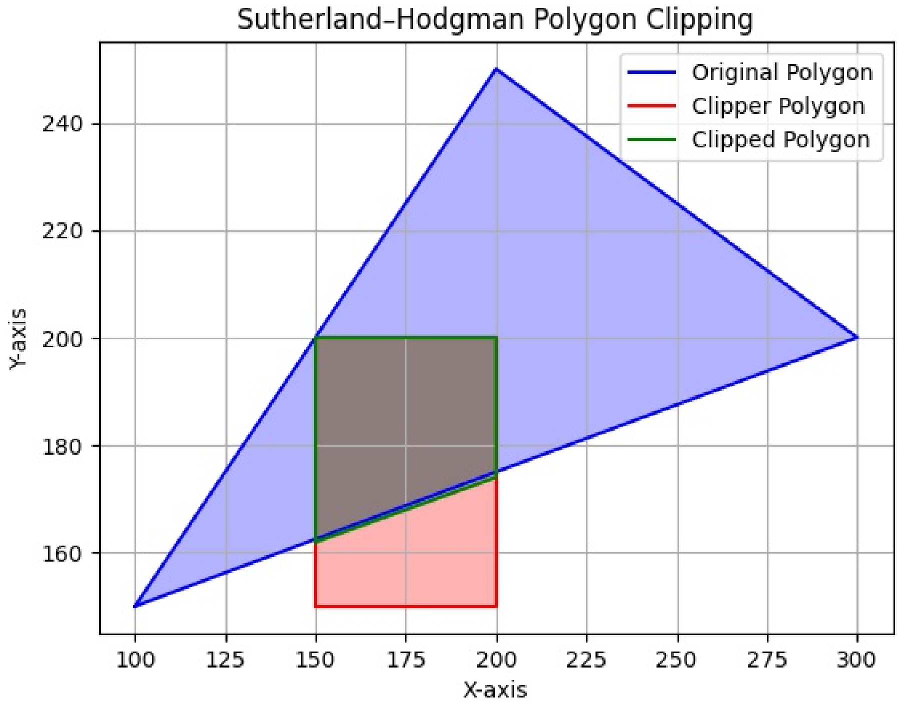


AIM:
To write a program to implement the polygon clipping using Sutherland Hodgeman polygon clipping algorithm.

```
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],
```

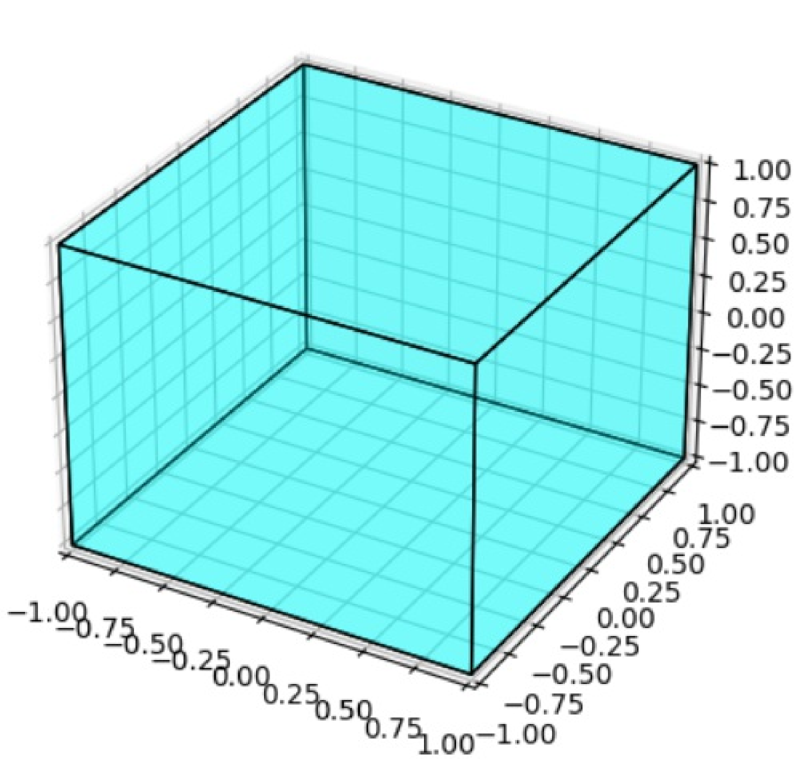
```
                              [0, 0, 0, 1]])
    return apply_transformation(points, translation_matrix)
def scale(points, Sx, Sy, Sz):
    scaling_matrix = np.array([[Sx, 0, 0, 0],
                               [0, Sy, 0, 0],
                               [0, 0, Sz, 0],
                               [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()
```

```
Original Rectangle:
Coordinates of the Rectangle:
Point 1: [1 1 1]
Point 2: [ 1 -1  1]
Point 3: [-1 -1  1]
Point 4: [-1  1  1]
Point 5: [ 1  1 -1]
Point 6: [ 1 -1 -1]
Point 7: [-1 -1 -1]
Point 8: [-1  1 -1]
```

Initial Rectangle

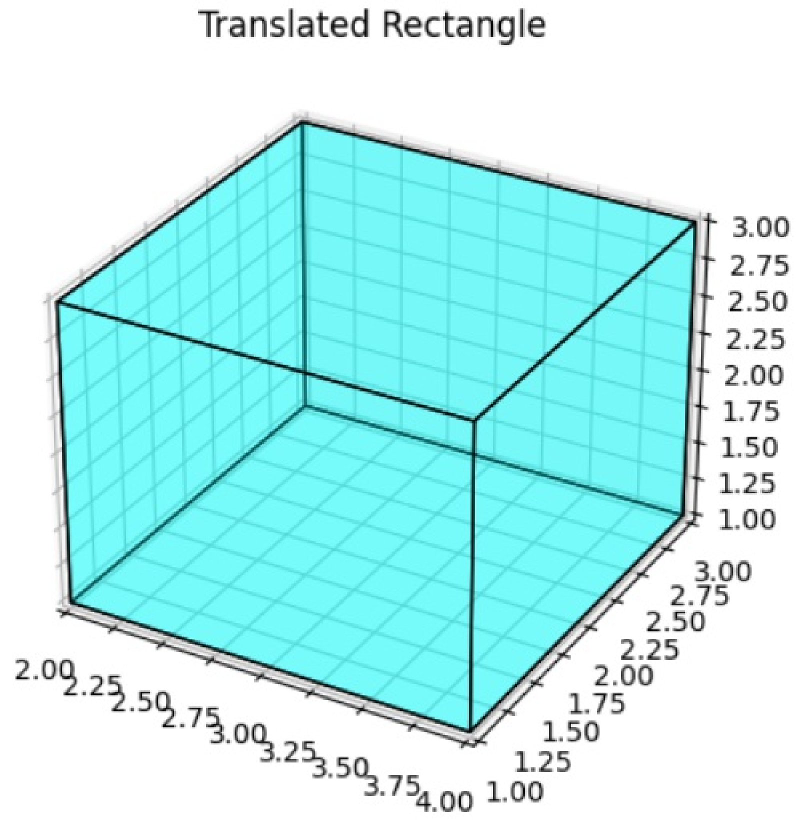


Tx: 3

Ty: 2

Tz: 2

```
Coordinates of the Rectangle:
Point 1: [4. 3. 3.]
Point 2: [4. 1. 3.]
Point 3: [2. 1. 3.]
Point 4: [2. 3. 3.]
Point 5: [4. 3. 1.]
Point 6: [4. 1. 1.]
Point 7: [2. 1. 1.]
Point 8: [2. 3. 1.]
```

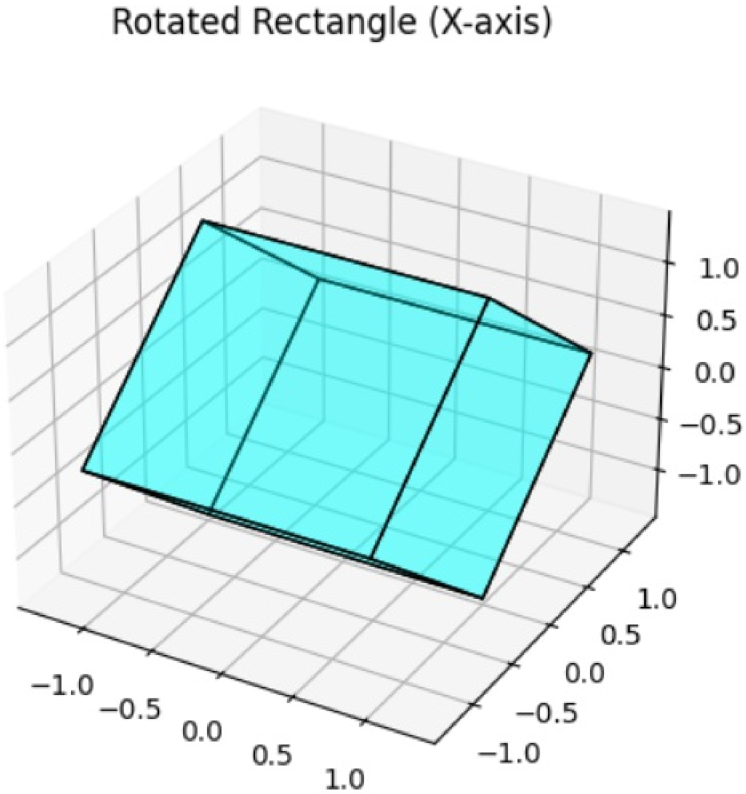


0.75 1.00−1.00

Axis: x

Angle: 45

```
Coordinates of the Rectangle:
Point 1: [1.00000000e+00 1.11022302e-16 1.41421356e+00]
Point 2: [ 1.00000000e+00 -1.41421356e+00  1.11022302e-16]
Point 3: [-1.00000000e+00 -1.41421356e+00  1.11022302e-16]
Point 4: [-1.00000000e+00  1.11022302e-16  1.41421356e+00]
Point 5: [ 1.00000000e+00  1.41421356e+00 -1.11022302e-16]
Point 6: [ 1.00000000e+00 -1.11022302e-16 -1.41421356e+00]
Point 7: [-1.00000000e+00 -1.11022302e-16 -1.41421356e+00]
Point 8: [-1.00000000e+00  1.41421356e+00 -1.11022302e-16]
```

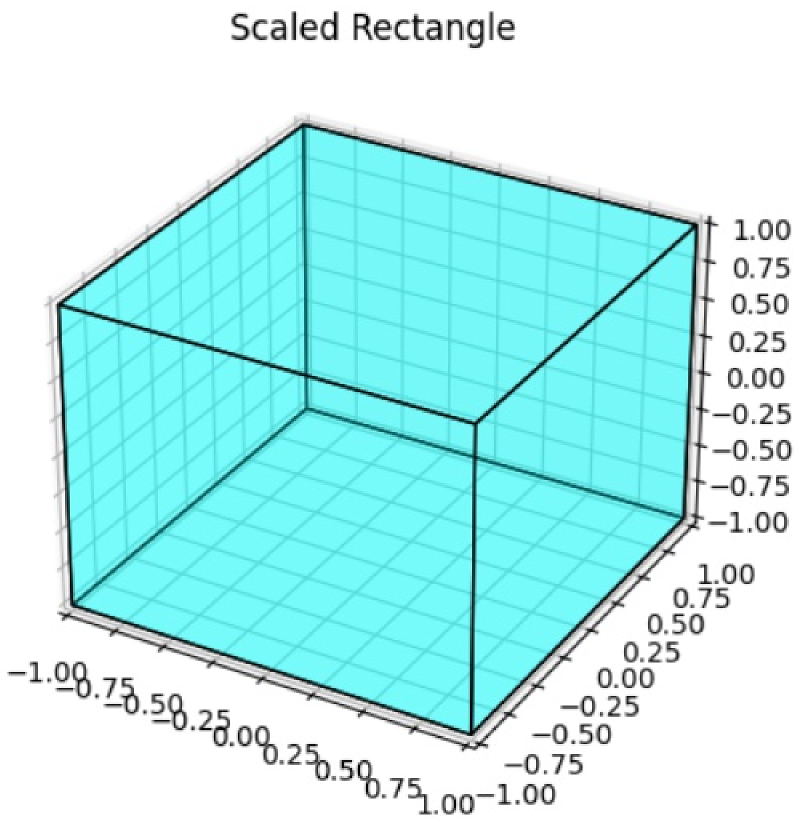


Sx: 1

Sy: 1

Sz: 1

```
Coordinates of the Rectangle:
Point 1: [1. 1. 1.]
Point 2: [ 1. -1.  1.]
Point 3: [-1. -1.  1.]
Point 4: [-1.  1.  1.]
Point 5: [ 1.  1. -1.]
Point 6: [ 1. -1. -1.]
Point 7: [-1. -1. -1.]
Point 8: [-1.  1. -1.]
```



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()
```