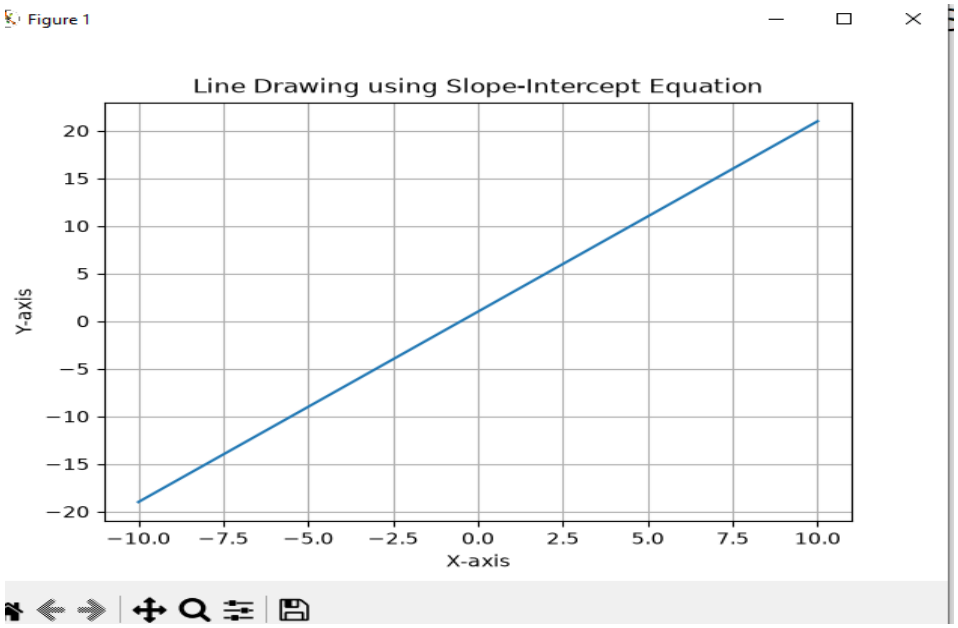
S.No	Title of Experiment
1.	Write a program to draw line using slope intercept equation
2.	Write a program to draw line using digital differential analyzer
3.	Write a program to implement Bresenham's line drawing algorithm
4.	Write a program to implement scan convert using polar co-ordinates
5.	Write a program to implement Bresenham's circle drawing algorithm
6.	Write a program to implement midpoint circle drawing algorithm
7.	To perform 2D Transformations such as Translation ,Rotation, Scaling, Reflection and Shearing.
8.	To implement Cohen-Sutherland 2D clipping and window view port mapping
9.	To implement Liang Barsky Line clipping algorithm
10.	To perform 3D Transformations such as Translation, Rotation and Scaling .
11.	Write a program to draw HUT or another Geometrical Figure
12.	Write a program to draw Decagon
13.	Write a program to demonstrate the polygon filling

1. Write a program to draw line using slope intercept equation

Source Code:

```
import matplotlib.pyplot as plt
# Given values
m = 2       # slope
b = 1       # y-intercept
# Generate x values
x = range(-10, 11)
# Calculate y using slope-intercept equation
y = [m * i + b for i in x]
# Draw the line
plt.plot(x, y)
# Add labels and title
plt.xlabel("X-axis")
plt.ylabel("Y-axis")
plt.title("Line Drawing using Slope-Intercept Equation")
# Show grid
plt.grid(True)
# Display the graph
plt.show()
```

OUTPUT:



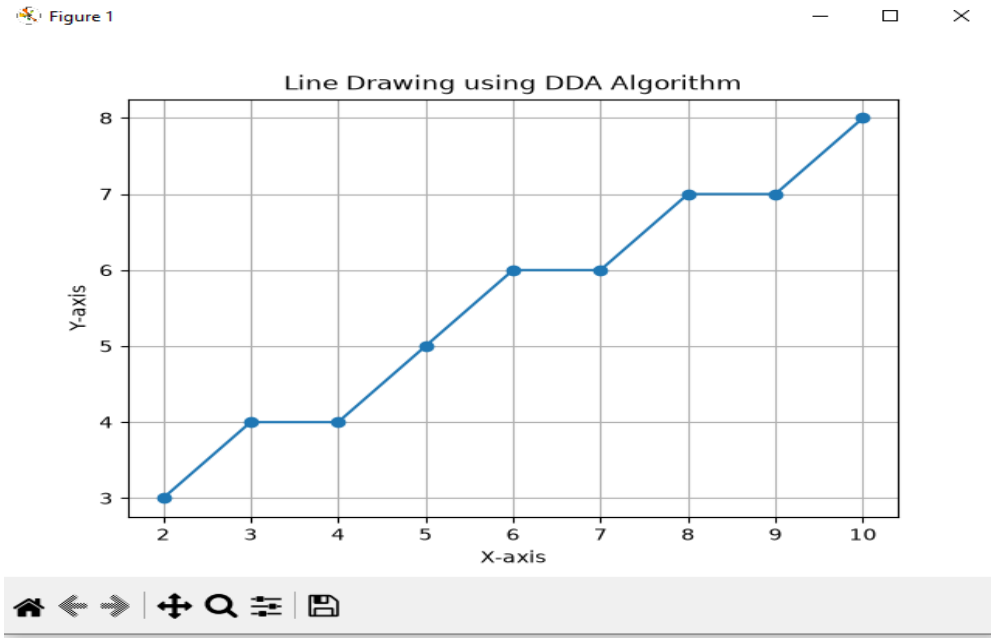
Result: The program was successfully executed and a straight line was drawn using the slope-intercept equation y = mx + b, with slope m = 2 and y-intercept b = 1.

2. Write a program to draw line using digital differential analyzer

Source Code:

```
import matplotlib.pyplot as plt
# Starting and ending points
x1 = 2
y1 = 3
x2 = 10
y2 = 8
# Calculate dx and dy
dx = x2 - x1
dy = y2 - y1
# Calculate number of steps
steps = max(abs(dx), abs(dy))
# Calculate increments
x_increment = dx / steps
y_increment = dy / steps
# Lists to store points
x_points = []
y_points = []
# Starting point
x = x1
y = y1
# DDA algorithm
for i in range(steps + 1):
    x_points.append(round(x))
    y_points.append(round(y))
    x = x + x_increment
    y = y + y_increment
# Draw the line
plt.plot(x_points, y_points, marker='o')
# Labels and title
plt.xlabel("X-axis")
plt.ylabel("Y-axis")
plt.title("Line Drawing using DDA Algorithm")
# Grid
plt.grid(True)
# Display graph
plt.show()
```

OUTPUT



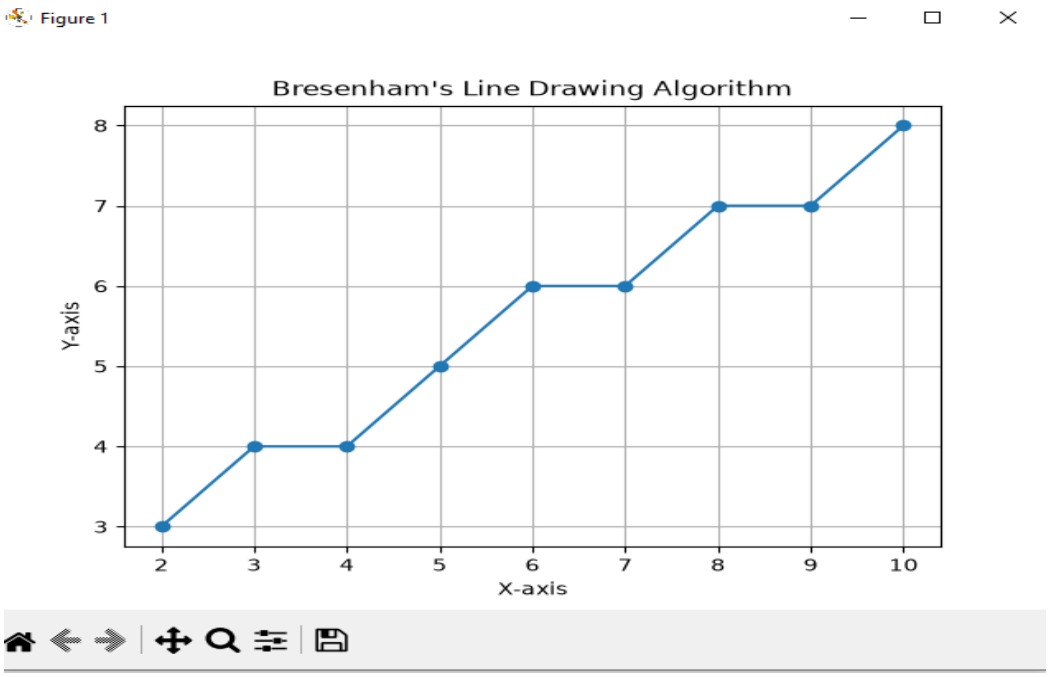
Result: The program was successfully executed and a straight line was drawn using the Digital Differential Analyzer (DDA) line drawing algorithm between the points (2, 3) and (10, 8). Thus, the required line was successfully displayed on the graph using the DDA algorithm.

3. Write a program to implement Bresenham's line drawing algorithm

Source Code:

```
import matplotlib.pyplot as plt
# Starting and ending points
x1 = 2
y1 = 3
x2 = 10
y2 = 8
# Calculate dx and dy
dx = x2 - x1
dy = y2 - y1
# Initial decision parameter
p = 2 * dy - dx
# Lists to store points
x_points = [x1]
y_points = [y1]
x = x1
y = y1
# Bresenham's algorithm
while x < x2:
    x = x + 1
    if p < 0:
        p = p + 2 * dy
    else:
        y = y + 1
        p = p + 2 * dy - 2 * dx
    x_points.append(x)
    y_points.append(y)
# Draw the line
plt.plot(x_points, y_points, marker='o')
# Labels and title
plt.xlabel("X-axis")
plt.ylabel("Y-axis")
plt.title("Bresenham's Line Drawing Algorithm")
# Display grid
plt.grid(True)
# Display the graph
plt.show()
```

OUTPUT:



Result: The program was successfully executed and a **straight line was drawn using Bresenham's Line Drawing Algorithm** between the points **(2, 3)** and **(10, 8)**. Thus, the required line was successfully displayed on the graph using the **Bresenham's Line Drawing Algorithm**.

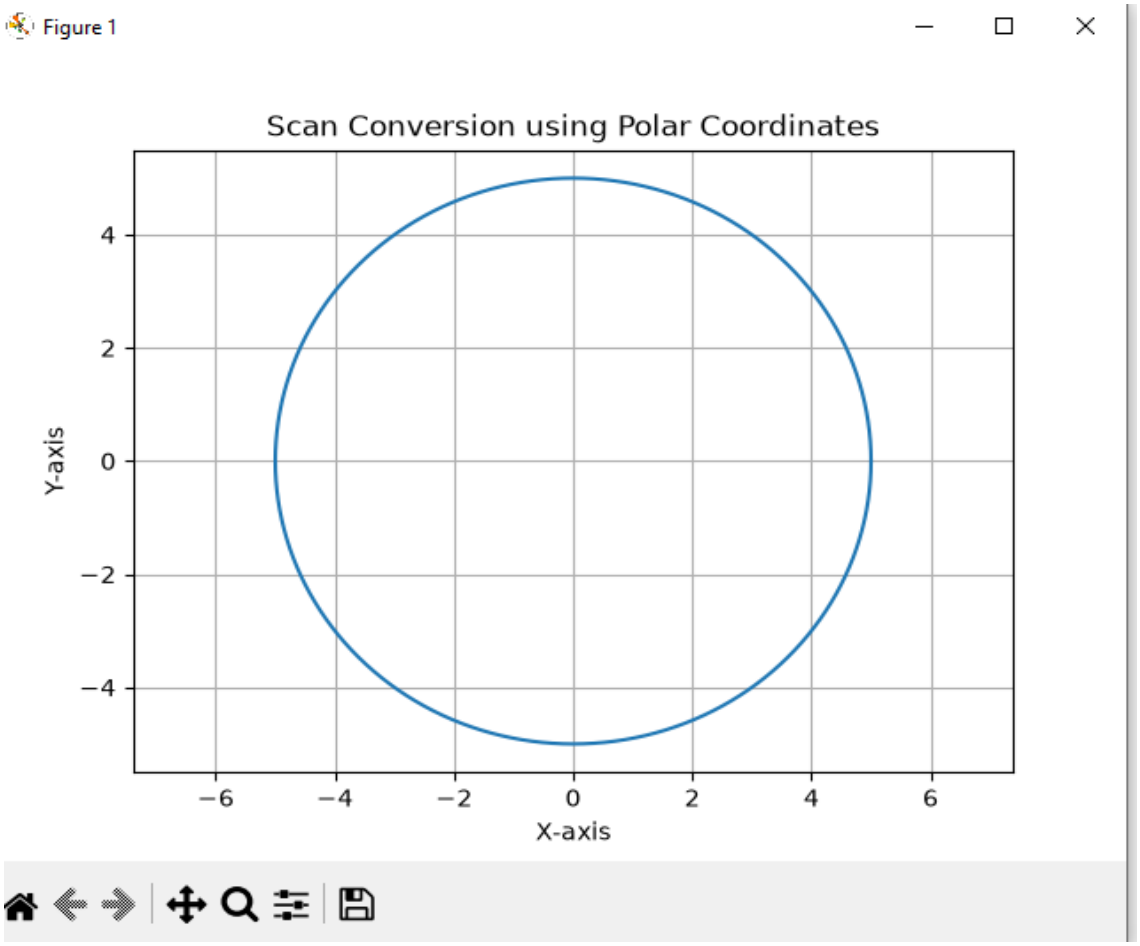
4. Write a program to implement scan convert using polar co-ordinates

Source Code:

```
import matplotlib.pyplot as plt
import math
# Center of the circle
xc = 0
yc = 0
# Radius
r = 5
# Lists to store points
x_points = []
y_points = []
# Scan convert using polar coordinates
for angle in range(0, 361):
    theta = math.radians(angle)
    x = xc + r * math.cos(theta)
    y = yc + r * math.sin(theta)
    x_points.append(x)
    y_points.append(y)
# Draw the circle
plt.plot(x_points, y_points)
# Labels and title
plt.xlabel("X-axis")
plt.ylabel("Y-axis")
plt.title("Scan Conversion using Polar Coordinates")
# Equal scale for X and Y axes
plt.axis("equal")

# Display grid
plt.grid(True)
# Display the graph
plt.show()
```

OUTPUT:



Result: The circle was successfully drawn using **scan conversion with polar coordinates**.

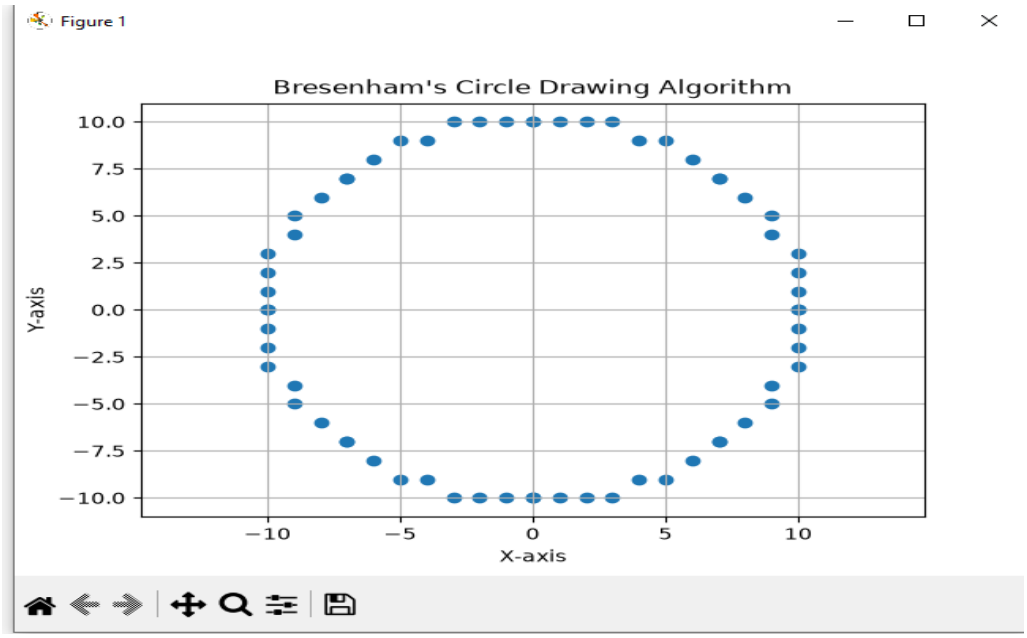
5. Write a program to implement Bresenham's circle drawing algorithm

Source Code:

```
import matplotlib.pyplot as plt
# Center and radius of the circle
xc = 0
yc = 0
r = 10
# Lists to store points
x_points = []
y_points = []
# Initial values
x = 0
y = r
p = 3 - 2 * r
# Bresenham's circle algorithm
while x <= y:
    # Calculate the 8 symmetric points
    points = [
        (xc + x, yc + y),
        (xc - x, yc + y),
        (xc + x, yc - y),
        (xc - x, yc - y),
        (xc + y, yc + x),
        (xc - y, yc + x),
        (xc + y, yc - x),
        (xc - y, yc - x)
    ]
    for px, py in points:
        x_points.append(px)
        y_points.append(py)
    # Decision parameter
    if p < 0:
        p = p + 4 * x + 6
    else:
        p = p + 4 * (x - y) + 10
        y = y - 1
    x = x + 1
# Draw the circle
plt.scatter(x_points, y_points)
# Labels and title
plt.xlabel("X-axis")
plt.ylabel("Y-axis")
plt.title("Bresenham's Circle Drawing Algorithm")
```

```
# Equal scale
plt.axis("equal")
# Grid
plt.grid(True)
# Display
plt.show()
```

OUTPUT:



Result:-The circle was successfully drawn using **Bresenham's Circle Drawing Algorithm**.

6. Write a program to implement midpoint circle drawing algorithm

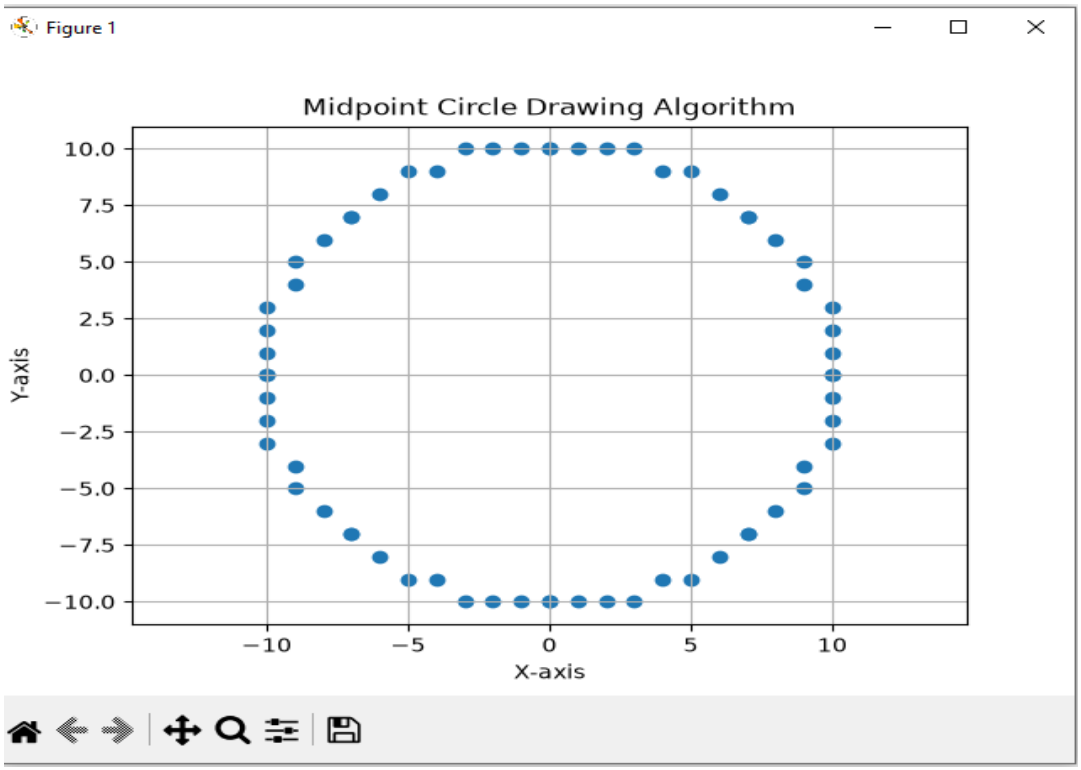
Source Code:

```
import matplotlib.pyplot as plt
# Center and radius of the circle
xc = 0
yc = 0
r = 10
# Lists to store points
x_points = []
y_points = []
# Initial values
x = 0
y = r
p = 1 - r
# Midpoint circle algorithm
while x <= y:
    # Store 8 symmetric points
    points = [
        (xc + x, yc + y),
        (xc - x, yc + y),
        (xc + x, yc - y),
        (xc - x, yc - y),
        (xc + y, yc + x),
        (xc - y, yc + x),
        (xc + y, yc - x),
        (xc - y, yc - x)
    ]

    for px, py in points:
        x_points.append(px)
        y_points.append(py)
    # Decision parameter
    if p < 0:
        p = p + 2 * x + 3
    else:
        p = p + 2 * (x - y) + 5
        y = y - 1
    x = x + 1
# Draw the circle
plt.scatter(x_points, y_points)
# Labels and title
plt.xlabel("X-axis")
```

```
plt.ylabel("Y-axis")
plt.title("Midpoint Circle Drawing Algorithm")
# Equal scale
plt.axis("equal")
# Grid
plt.grid(True)
# Display
plt.show()
```

OUTPUT:



Result: The circle was successfully drawn using the **Midpoint Circle Drawing Algorithm**.

7. To perform 2D Transformations such as Translation ,Rotation, Scaling, Reflection and Shearing.

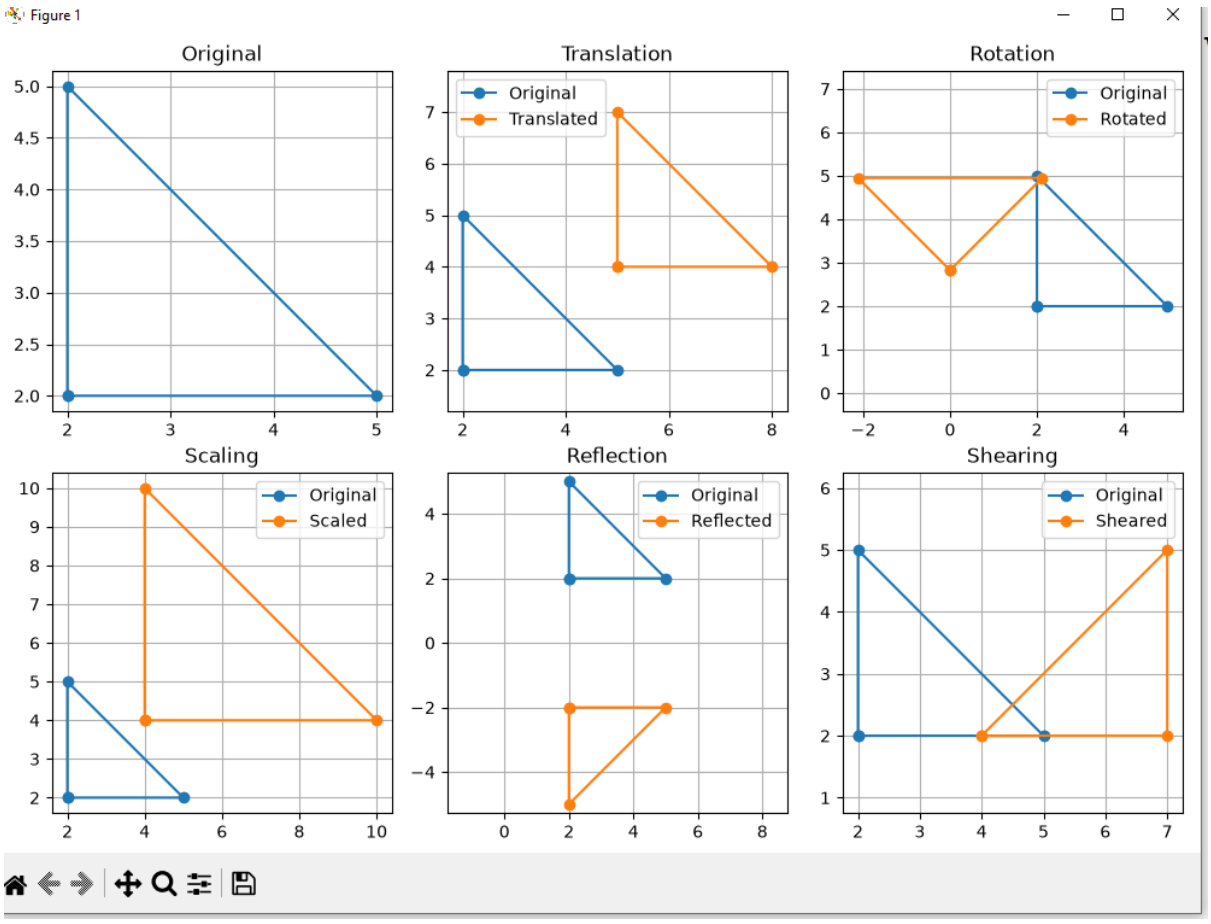
Source Code:

```
import matplotlib.pyplot as plt
import numpy as np
# Original triangle coordinates
x = np.array([2, 5, 2, 2])
y = np.array([2, 2, 5, 2])
# ------------------------------
# 1. Translation
# ------------------------------
tx = 3
ty = 2
x_translation = x + tx
y_translation = y + ty
# ------------------------------
# 2. Rotation
# ------------------------------
angle = 45
theta = np.radians(angle)
x_rotation = x * np.cos(theta) - y * np.sin(theta)
y_rotation = x * np.sin(theta) + y * np.cos(theta)
# ------------------------------
# 3. Scaling
# ------------------------------
sx = 2
sy = 2
x_scaling = x * sx
y_scaling = y * sy
# ------------------------------
# 4. Reflection
# ------------------------------
x_reflection = x
y_reflection = -y
# ------------------------------
# 5. Shearing
# ------------------------------
shx = 1
shy = 0
x_shearing = x + shx * y
y_shearing = y + shy * x
# ------------------------------
```

```
# Display all transformations
# ----------------------------
plt.figure(figsize=(10, 8))
# Original
plt.subplot(2, 3, 1)
plt.plot(x, y, marker='o')
plt.title("Original")
plt.grid(True)
plt.axis("equal")
# Translation
plt.subplot(2, 3, 2)
plt.plot(x, y, marker='o', label="Original")
plt.plot(x_translation, y_translation, marker='o', label="Translated")
plt.title("Translation")
plt.grid(True)
plt.axis("equal")
plt.legend()
# Rotation
plt.subplot(2, 3, 3)
plt.plot(x, y, marker='o', label="Original")
plt.plot(x_rotation, y_rotation, marker='o', label="Rotated")
plt.title("Rotation")
plt.grid(True)
plt.axis("equal")
plt.legend()
# Scaling
plt.subplot(2, 3, 4)
plt.plot(x, y, marker='o', label="Original")
plt.plot(x_scaling, y_scaling, marker='o', label="Scaled")
plt.title("Scaling")
plt.grid(True)
plt.axis("equal")
plt.legend()
# Reflection
plt.subplot(2, 3, 5)
plt.plot(x, y, marker='o', label="Original")
plt.plot(x_reflection, y_reflection, marker='o', label="Reflected")
plt.title("Reflection")
plt.grid(True)
plt.axis("equal")
plt.legend()
# Shearing
plt.subplot(2, 3, 6)
plt.plot(x, y, marker='o', label="Original")
```

```
plt.plot(x_shearing, y_shearing, marker='o', label="Sheared")
plt.title("Shearing")
plt.grid(True)
plt.axis("equal")
plt.legend()
plt.tight_layout()
plt.show()
```

OUTPUT:



Result: The program was successfully executed and the **2D** transformations—Translation, Rotation, Scaling, Reflection, and Shearing—were successfully performed and displayed on the triangle.

8. To implement Cohen-Sutherland 2D clipping and window view port mapping

Source Code:

```
import matplotlib.pyplot as plt
# Clipping window
xmin = 2
ymin = 2
xmax = 8
ymax = 6
# Line coordinates
x1 = 1
y1 = 1
x2 = 10
y2 = 8
# Region codes
INSIDE = 0
LEFT = 1
RIGHT = 2
BOTTOM = 4
TOP = 8
# Calculate region code
def compute_code(x, y):
    code = INSIDE
    if x < xmin:
        code |= LEFT
    elif x > xmax:
        code |= RIGHT
    if y < ymin:
        code |= BOTTOM
    elif y > ymax:
        code |= TOP
    return code
# Cohen-Sutherland Line Clipping
def cohen_sutherland(x1, y1, x2, y2):
    code1 = compute_code(x1, y1)
    code2 = compute_code(x2, y2)
    accept = False
    while True:
        # Both points are inside
        if code1 == 0 and code2 == 0:
            accept = True
            break
        # Both points are outside in the same region
        elif (code1 & code2) != 0:
            break
        else:
            # Select point outside
            if code1 != 0:
                code_out = code1
```

```
            else:
                code_out = code2
            # Find intersection point
            if code_out & TOP:
                x = x1 + (x2 - x1) * (ymax - y1) / (y2 - y1)
                y = ymax

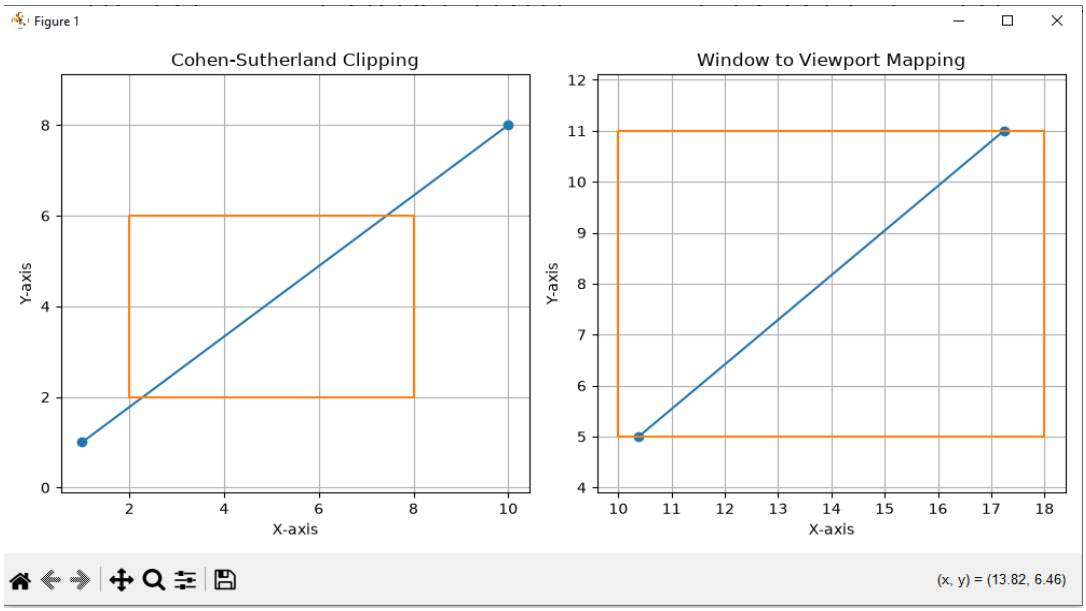
            elif code_out & BOTTOM:
                x = x1 + (x2 - x1) * (ymin - y1) / (y2 - y1)
                y = ymin
            elif code_out & RIGHT:
                y = y1 + (y2 - y1) * (xmax - x1) / (x2 - x1)
                x = xmax
            elif code_out & LEFT:
                y = y1 + (y2 - y1) * (xmin - x1) / (x2 - x1)
                x = xmin
            # Replace outside point
            if code_out == code1:
                x1 = x
                y1 = y
                code1 = compute_code(x1, y1)
            else:
                x2 = x
                y2 = y
                code2 = compute_code(x2, y2)
    if accept:
        return x1, y1, x2, y2
    else:
        return None
# Perform clipping
result = cohen_sutherland(x1, y1, x2, y2)
# Window to Viewport Mapping
if result:

    cx1, cy1, cx2, cy2 = result
    # Viewport coordinates
    vxmin = 10
    vymin = 5
    vxmax = 18
    vymax = 11
    # Map first point
    vx1 = vxmin + (cx1 - xmin) * (vxmax - vxmin) / (xmax - xmin)
    vy1 = vymin + (cy1 - ymin) * (vymax - vymin) / (ymax - ymin)
    # Map second point
    vx2 = vxmin + (cx2 - xmin) * (vxmax - vxmin) / (xmax - xmin)
    vy2 = vymin + (cy2 - ymin) * (vymax - vymin) / (ymax - ymin)
    # Create graphs
    plt.figure(figsize=(10, 5))
    # Clipping Window
    plt.subplot(1, 2, 1)
```

```
    plt.plot([x1, x2], [y1, y2], marker='o')
    plt.plot(
        [xmin, xmax, xmax, xmin, xmin],
        [ymin, ymin, ymax, ymax, ymin]
    )
    plt.title("Cohen-Sutherland Clipping")
    plt.xlabel("X-axis")
    plt.ylabel("Y-axis")
    plt.grid(True)
    plt.axis("equal")

    # Viewport
    plt.subplot(1, 2, 2)
    plt.plot([vx1, vx2], [vy1, vy2], marker='o')
    plt.plot(
        [vxmin, vxmax, vxmax, vxmin, vxmin],
        [vymin, vymin, vymax, vymax, vymin]
    )
    plt.title("Window to Viewport Mapping")
    plt.xlabel("X-axis")
    plt.ylabel("Y-axis")
    plt.grid(True)
    plt.axis("equal")
    plt.tight_layout()
    plt.show()
else:
    print("Line is completely outside the clipping window.")
```

OUTPUT:



Result: The program was successfully executed and **Cohen-Sutherland 2D line clipping and window-to-viewport mapping were successfully performed and displayed.**

9. To implement Liang Barsky Line clipping algorithm

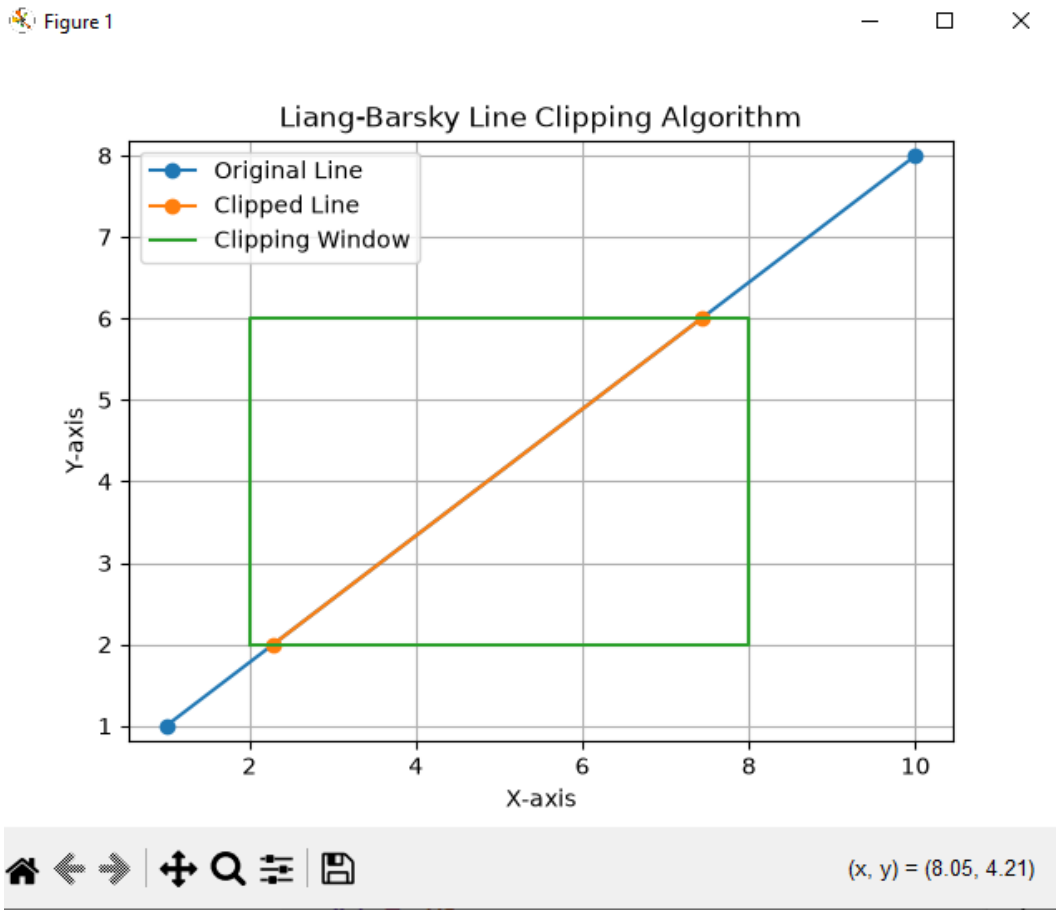
Source Code:

```
import matplotlib.pyplot as plt
# Clipping window
xmin = 2
ymin = 2
xmax = 8
ymax = 6
# Line coordinates
x1 = 1
y1 = 1
x2 = 10
y2 = 8
# Liang-Barsky Line Clipping
def liang_barsky(x1, y1, x2, y2):
    dx = x2 - x1
    dy = y2 - y1
    p = [-dx, dx, -dy, dy]
    q = [x1 - xmin, xmax - x1, y1 - ymin, ymax - y1]
    u1 = 0
    u2 = 1
    for i in range(4):
        if p[i] == 0:
            if q[i] < 0:
                return None
        else:
            u = q[i] / p[i]
            if p[i] < 0:
                u1 = max(u1, u)
            else:
                u2 = min(u2, u)
    if u1 > u2:
        return None
    # Calculate clipped line coordinates
    cx1 = x1 + u1 * dx
    cy1 = y1 + u1 * dy
    cx2 = x1 + u2 * dx
    cy2 = y1 + u2 * dy
    return cx1, cy1, cx2, cy2
# Perform clipping
result = liang_barsky(x1, y1, x2, y2)
# Display result
```

```
if result:
    cx1, cy1, cx2, cy2 = result
    print("Clipped Line Coordinates:")
    print("Start Point:", (cx1, cy1))
    print("End Point:", (cx2, cy2))
    # Draw graph
    plt.plot(
        [x1, x2],
        [y1, y2],
        marker='o',
        label="Original Line"
    )

    plt.plot(
        [cx1, cx2],
        [cy1, cy2],
        marker='o',
        label="Clipped Line"
    )
    # Draw clipping window
    plt.plot(
        [xmin, xmax, xmax, xmin, xmin],
        [ymin, ymin, ymax, ymax, ymin],
        label="Clipping Window"
    )
    plt.xlabel("X-axis")
    plt.ylabel("Y-axis")
    plt.title("Liang-Barsky Line Clipping Algorithm")
    plt.grid(True)
    plt.axis("equal")
    plt.legend()
    plt.show()
else:
    print("Line is completely outside the clipping window.")
```

OUTPUT:



Result: The program was successfully executed and the **Liang-Barsky Line Clipping Algorithm** was successfully implemented to clip the given line against the specified clipping window.

10. To perform 3D Transformations such as Translation, Rotation and Scaling .

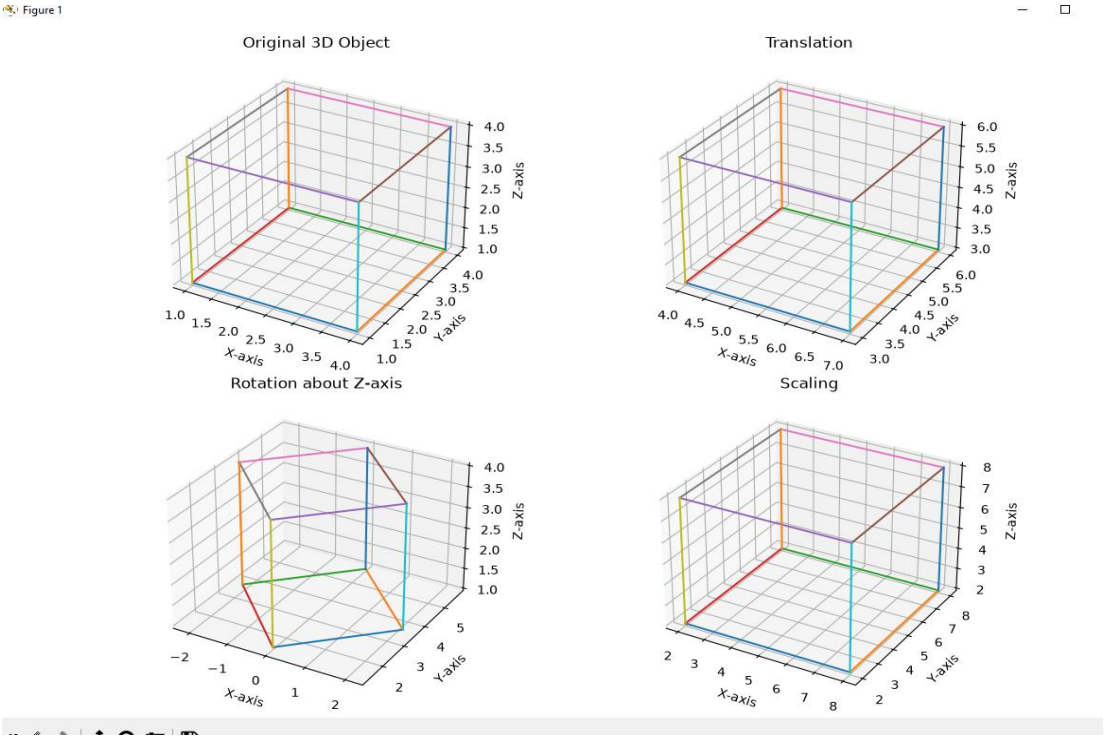
Source Code:

```
import matplotlib.pyplot as plt
import numpy as np
# Original 3D cube vertices
vertices = np.array([
    [1, 1, 1],
    [4, 1, 1],
    [4, 4, 1],
    [1, 4, 1],
    [1, 1, 4],
    [4, 1, 4],
    [4, 4, 4],
    [1, 4, 4]
])
# Cube edges
edges = [
    (0, 1), (1, 2), (2, 3), (3, 0),
    (4, 5), (5, 6), (6, 7), (7, 4),
    (0, 4), (1, 5), (2, 6), (3, 7)
]
# Function to draw cube
def draw_cube(ax, points, title):
    for edge in edges:
        p1 = points[edge[0]]
        p2 = points[edge[1]]
        ax.plot(
            [p1[0], p2[0]],
            [p1[1], p2[1]],
            [p1[2], p2[2]]
        )
    ax.set_title(title)
    ax.set_xlabel("X-axis")
    ax.set_ylabel("Y-axis")
    ax.set_zlabel("Z-axis")
# --------------------------------
# 1. Translation
# --------------------------------
tx = 3
ty = 2
tz = 2
translation = np.array([tx, ty, tz])
translated = vertices + translation
```

```
# --------------------------------
# 2. Rotation about Z-axis
# --------------------------------
angle = 45
theta = np.radians(angle)
rotation_matrix = np.array([
   [np.cos(theta), -np.sin(theta), 0],
   [np.sin(theta),  np.cos(theta), 0],
   [0,             0,              1]
])
rotated = vertices @ rotation_matrix.T

# --------------------------------
# 3. Scaling
# --------------------------------
sx = 2
sy = 2
sz = 2
scaling = np.array([sx, sy, sz])
scaled = vertices * scaling
# --------------------------------
# Display transformations
# --------------------------------
fig = plt.figure(figsize=(12, 8))
# Original
ax1 = fig.add_subplot(2, 2, 1, projection='3d')
draw_cube(ax1, vertices, "Original 3D Object")
# Translation
ax2 = fig.add_subplot(2, 2, 2, projection='3d')
draw_cube(ax2, translated, "Translation")
# Rotation
ax3 = fig.add_subplot(2, 2, 3, projection='3d')
draw_cube(ax3, rotated, "Rotation about Z-axis")
# Scaling
ax4 = fig.add_subplot(2, 2, 4, projection='3d')
draw_cube(ax4, scaled, "Scaling")
plt.tight_layout()
plt.show()
```

OUTPUT:



Result: The program was successfully executed and the 3D transformations—Translation, Rotation about the Z-axis, and Scaling—were successfully performed and displayed on the 3D cube.

11. Write a program to draw HUT or another Geometrical Figure

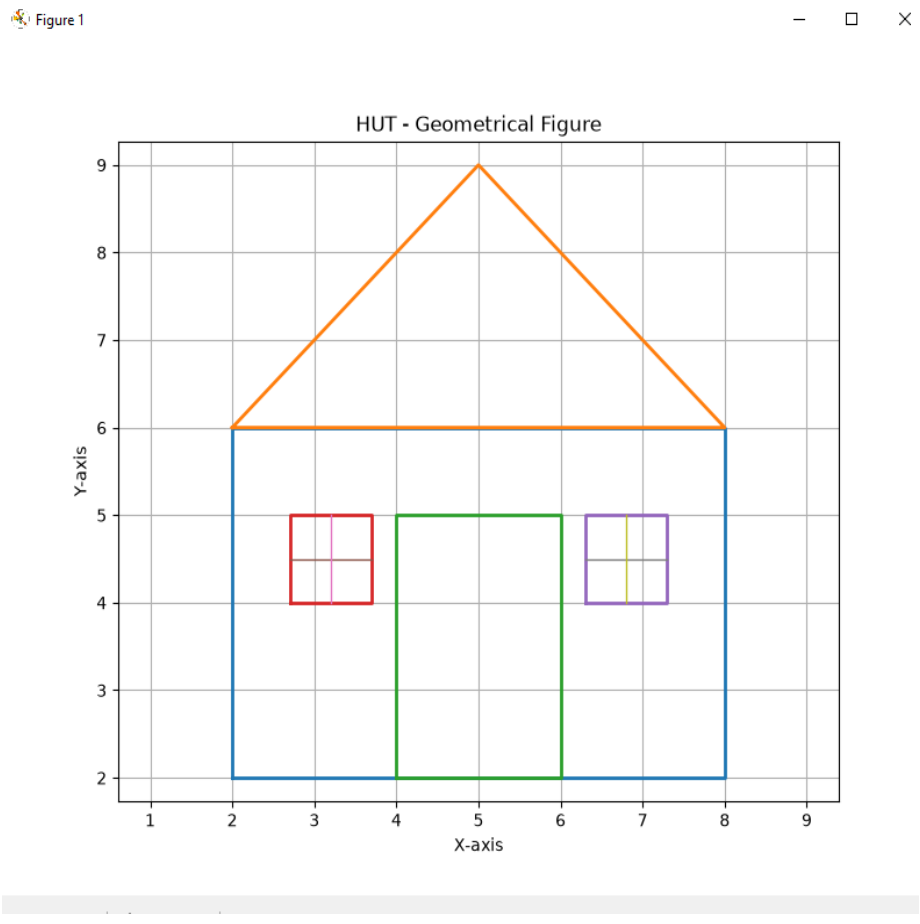
Source Code:

```
import matplotlib.pyplot as plt
# Create figure
plt.figure(figsize=(8, 8))
# ------------------------------
# House Body
# ------------------------------
x_body = [2, 8, 8, 2, 2]
y_body = [2, 2, 6, 6, 2]
plt.plot(x_body, y_body, linewidth=2)
# ------------------------------
# Roof
# ------------------------------
x_roof = [2, 5, 8, 2]
y_roof = [6, 9, 6, 6]
plt.plot(x_roof, y_roof, linewidth=2)
# ------------------------------
# Door
# ------------------------------
x_door = [4, 6, 6, 4, 4]
y_door = [2, 2, 5, 5, 2]
plt.plot(x_door, y_door, linewidth=2)
# ------------------------------
# Left Window
# ------------------------------
x_window1 = [2.7, 3.7, 3.7, 2.7, 2.7]
y_window1 = [4, 4, 5, 5, 4]

plt.plot(x_window1, y_window1, linewidth=2)
# ------------------------------
# Right Window
# ------------------------------
x_window2 = [6.3, 7.3, 7.3, 6.3, 6.3]
y_window2 = [4, 4, 5, 5, 4]
plt.plot(x_window2, y_window2, linewidth=2)
# ------------------------------
# Window Cross Lines
# ------------------------------
plt.plot([2.7, 3.7], [4.5, 4.5], linewidth=1)
plt.plot([3.2, 3.2], [4, 5], linewidth=1)
plt.plot([6.3, 7.3], [4.5, 4.5], linewidth=1)
```

```
plt.plot([6.8, 6.8], [4, 5], linewidth=1)
# ------------------------------
# Labels
# ------------------------------
plt.title("HUT - Geometrical Figure")
plt.xlabel("X-axis")
plt.ylabel("Y-axis")
# Grid
plt.grid(True)
# Equal scale
plt.axis("equal")
# Display
plt.show()
```

OUTPUT:



Result: The program was successfully executed and the **Hut (House) geometrical figure was successfully drawn and displayed using Python.**

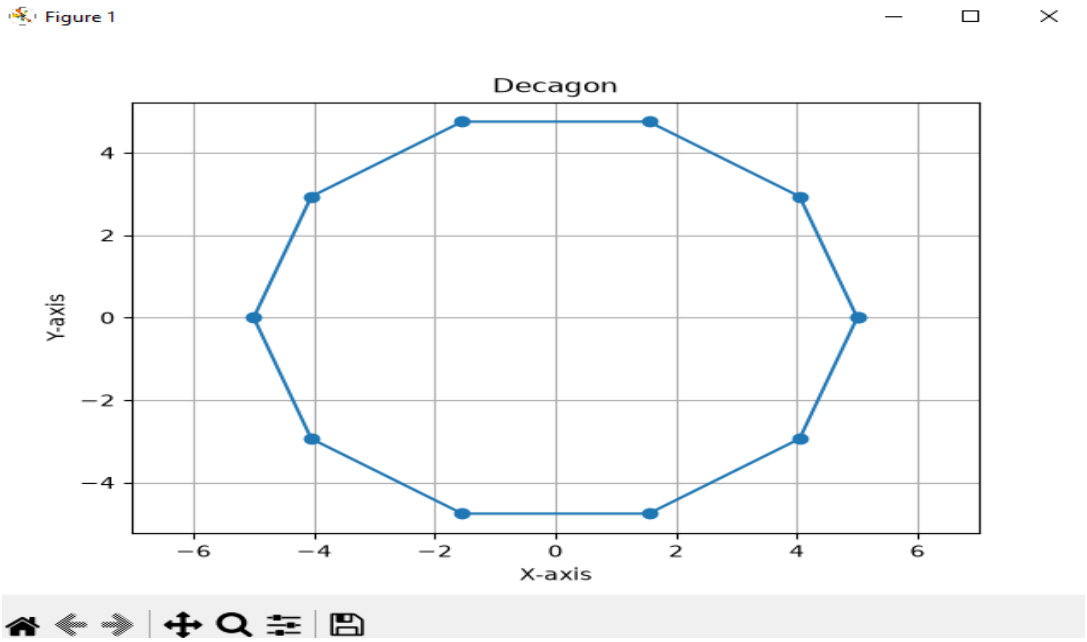
12. Write a program to draw Decagon

Source Code:

```
import matplotlib.pyplot as plt
import math
# Number of sides
n = 10
# Radius
r = 5
# Lists to store coordinates
x_points = []
y_points = []
# Calculate the vertices of the decagon
for i in range(n + 1):
    angle = 2 * math.pi * i / n
    x = r * math.cos(angle)
    y = r * math.sin(angle)
    x_points.append(x)
    y_points.append(y)
# Draw the decagon
plt.plot(x_points, y_points, marker='o')
# Title and labels
plt.title("Decagon")
plt.xlabel("X-axis")
plt.ylabel("Y-axis")
# Grid
plt.grid(True)
# Equal scale
plt.axis("equal")

# Display
plt.show()
```

OUTPUT:



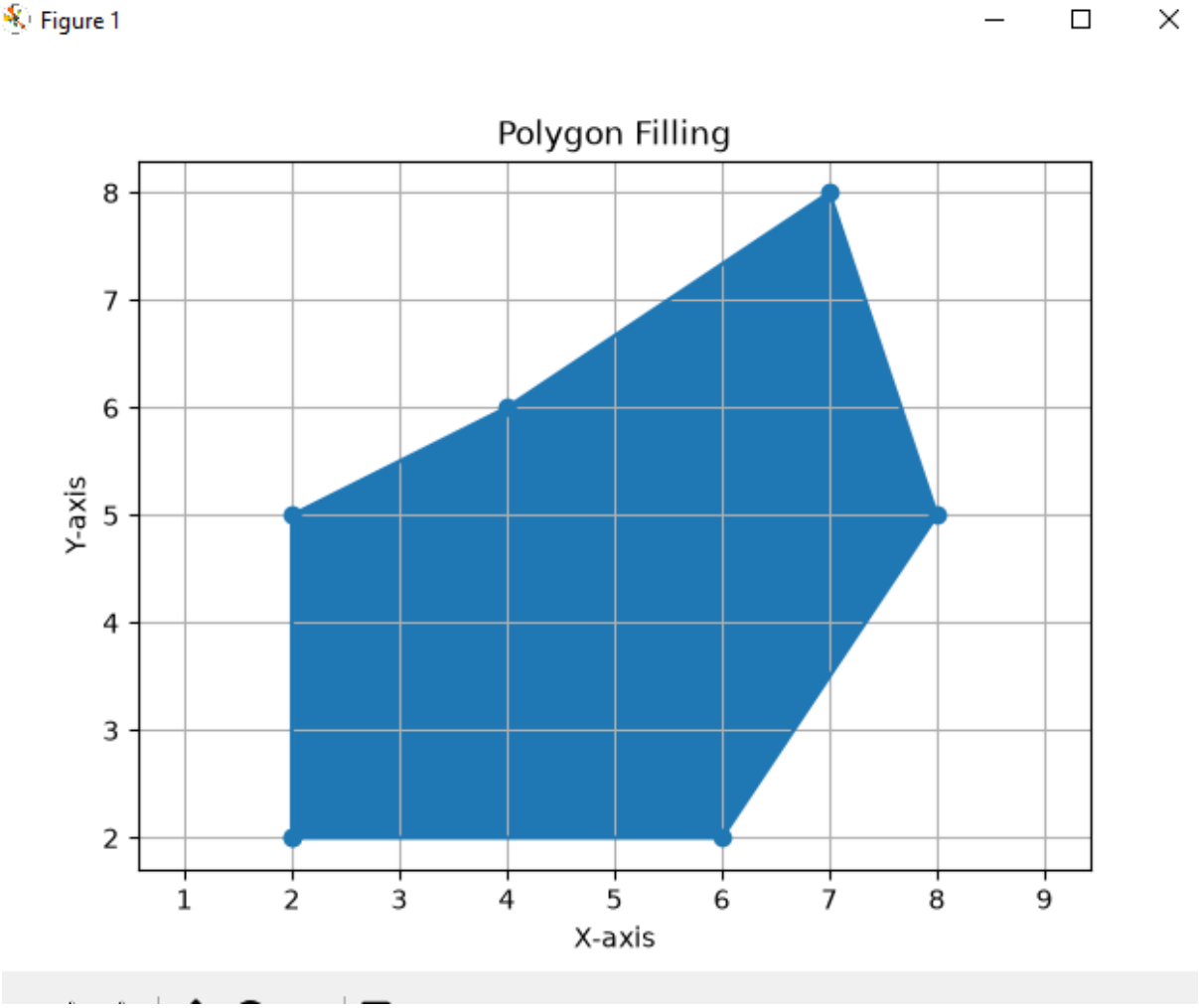
Result: The program was successfully executed and a **Decagon with 10 sides** was successfully drawn and displayed using Python.

13. Write a program to demonstrate the polygon filling

Source Code:

```
import matplotlib.pyplot as plt
# Polygon vertices
x = [2, 6, 8, 7, 4, 2]
y = [2, 2, 5, 8, 6, 5]
# Fill the polygon
plt.fill(x, y)
# Draw the boundary
plt.plot(x + [x[0]], y + [y[0]], marker='o')
# Title and labels
plt.title("Polygon Filling")
plt.xlabel("X-axis")
plt.ylabel("Y-axis")
# Grid
plt.grid(True)
# Equal scale
plt.axis("equal")
# Display
plt.show()
```

OUTPUT:



Result: The program was successfully executed and the **polygon was filled and displayed with its boundary and vertices using Python.**