**Index**

|  |  |  |
| --- | --- | --- |
| **S.No.** | **Title** | **Page No,** |
| **1** | Write a program to implement DDA and Bresenham's line drawing algorithm. | **3** |
| **2** | Write a program to implement mid-point circle drawing algorithm. | **5** |
| **3** | Write a program to clip a line using Cohen and Sutherland line clipping algorithm. | **7** |
| **4** | Write a program to clip a polygon using Sutherland Hodgeman algorithm. | **10** |
| **5** | Write a program to fill a polygon using Scan line fill algorithm. | **12** |
| **6** | Write a program to apply various 2D transformations on a 2D object (use homogenous Coordinates). | **13** |
| **7** | Write a program to apply various 3D transformations on a 3D object and then apply parallel and perspective projection on it. | **19** |
| **8** | Write a program to draw Hermite / Bezier curve. | **23** |

**Ques1:** Write a program to implement DDA and Bresenham's line drawing algorithm.

**CODE:-**

import matplotlib.pyplot as plt

def draw\_line\_DDA(x1, y1, x2, y2):

    dx = x2 - x1

    dy = y2 - y1

    steps = max(abs(dx), abs(dy))

    x\_increment = dx / steps

    y\_increment = dy / steps

    x = x1

    y = y1

    plt.plot(x, y, 'ro')

    print("Intermediate points using DDA:")

    for \_ in range(steps):

        x += x\_increment

        y += y\_increment

        plt.plot(round(x), round(y), 'ro')

        print(f"({round(x)}, {round(y)})")

    plt.plot([x1, x2], [y1, y2], 'b')

def draw\_line\_Bresenham(x1, y1, x2, y2):

    dx = abs(x2 - x1)

    dy = abs(y2 - y1)

    p = 2 \* dy - dx

    x = x1

    y = y1

    plt.plot(x, y, 'ro')

    print("Intermediate points using Bresenham:")

    if x1 < x2:

        x\_increment = 1

    else:

        x\_increment = -1

    if y1 < y2:

        y\_increment = 1

    else:

        y\_increment = -1

    while x != x2:

        x += x\_increment

        if p < 0:

            p += 2 \* dy

        else:

            y += y\_increment

            p += 2 \* (dy - dx)

        plt.plot(x, y, 'ro')

        print(f"({x}, {y})")

    plt.plot([x1, x2], [y1, y2], 'g')

# Test the functions

x1, y1 = 9, 18

x2, y2 = 14, 22

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

# Plotting using DDA

plt.subplot(1, 2, 1)

plt.title('DDA Algorithm')

draw\_line\_DDA(x1, y1, x2, y2)

plt.grid(True)

plt.axis('equal')

# Plotting using Bresenham's

plt.subplot(1, 2, 2)

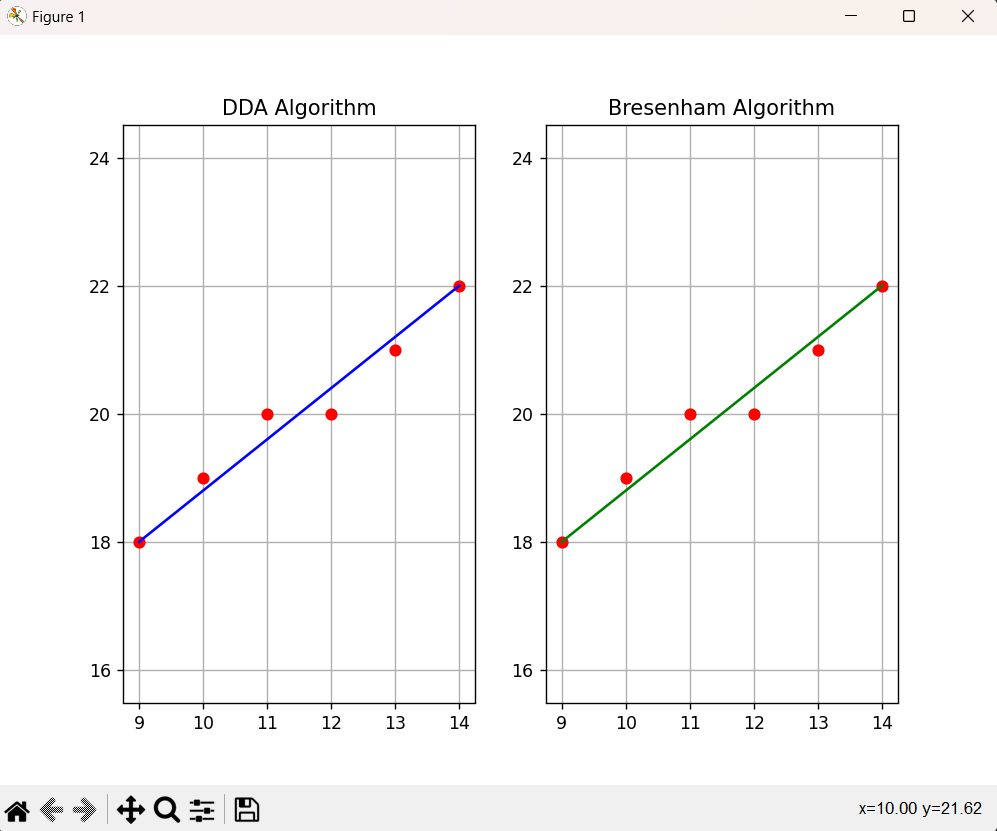
plt.title('Bresenham Algorithm')

draw\_line\_Bresenham(x1, y1, x2, y2)

plt.grid(True)

plt.axis('equal')

plt.show()

**OUTPUT:- **

**Ques2:** Write a program to implement mid-point circle drawing algorithm.

**CODE:-**

# mid point circle drawing algorithm

from graphics import GraphWin, Point

import time

def midPointCircleDraw(x\_centre, y\_centre, r):

    win = GraphWin("Mid Point Circle Generation Program Output", 600, 480)

    x = r

    y = 0

    P = 1 - r

    while x > y:

        x1 = x + x\_centre

        y1 = y + y\_centre

        x2 = -x + x\_centre

        y2 = y + y\_centre

        x3 = x + x\_centre

        y3 = -y + y\_centre

        x4 = -x + x\_centre

        y4 = -y + y\_centre

        PutPixel(win, x1, y1)

        PutPixel(win, x2, y2)

        PutPixel(win, x3, y3)

        PutPixel(win, x4, y4)

        x1 = y + x\_centre

        y1 = x + y\_centre

        x2 = -y + x\_centre

        y2 = x + y\_centre

        x3 = y + x\_centre

        y3 = -x + y\_centre

        x4 = -y + x\_centre

        y4 = -x + y\_centre

        PutPixel(win, x1, y1)

        PutPixel(win, x2, y2)

        PutPixel(win, x3, y3)

        PutPixel(win, x4, y4)

        y += 1

        if P <= 0:

            P = P + 2 \* y + 1

        else:

            x -= 1

            P = P + 2 \* y -2 \*x + 1

    win.getMouse()

    win.close()

def PutPixel(win, x, y):

    pt = Point(x,y)

    pt.draw(win)

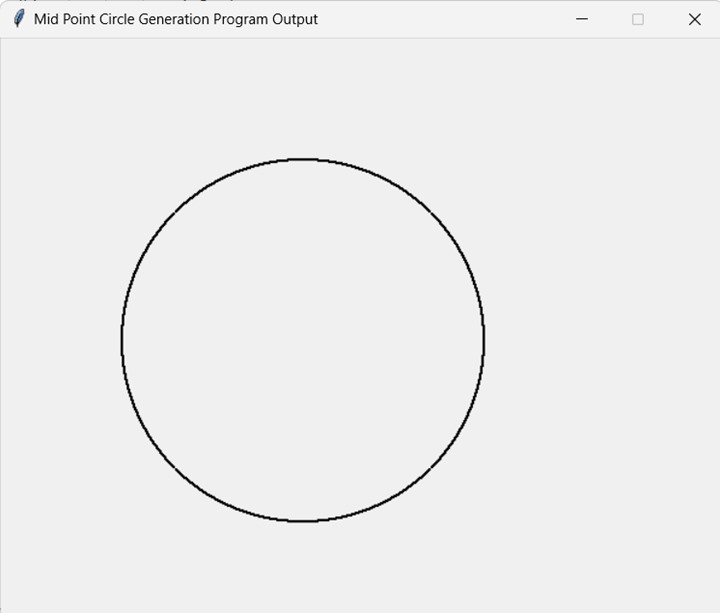
if \_\_name\_\_ == '\_\_main\_\_':

    x,y = map(int, input("Enter center co-ordinates: ").split())

    r = int(input("Enter radius: "))

    midPointCircleDraw(x,y,r)

**OUTPUT:-**

****

**Ques3:** Write a program to clip a line using Cohen and Sutherland line clipping algorithm.

**CODE:-**

import matplotlib.pyplot as plt

INSIDE = 0  # 0000

LEFT = 1    # 0001

RIGHT = 2   # 0010

BOTTOM = 4  # 0100

TOP = 8     # 1000

def computeCode(x, y):

    code = INSIDE

    if x < x\_min:      # left of rectangle

        code |= LEFT

    elif x > x\_max:    # right of rectangle

        code |= RIGHT

    if y < y\_min:      # below

        code |= BOTTOM

    elif y > y\_max:    # above

        code |= TOP

    return code

def cohenSutherlandClip(x1, y1, x2, y2):

    code1 = computeCode(x1, y1)

    code2 = computeCode(x2, y2)

    accept = False

    while True:

        if code1 == 0 and code2 == 0:

            accept = True

            break

        elif (code1 & code2) != 0:

            break

        else:

            x = 1.0

            y = 1.0

            if code1 != 0:

                code\_out = code1

            else:

                code\_out = 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 = x

                y1 = y

                code1 = computeCode(x1, y1)

            else:

                x2 = x

                y2 = y

                code2 = computeCode(x2, y2)

    if accept:

        print(f"Line accepted from {x1:.2f}, {y1:.2f} to {x2:.2f}, {y2:.2f}")

        plt.plot([x1, x2], [y1, y2], color='black', linewidth=2)  # Plot the accepted line

    else:

        print("Line rejected")

x\_max = 10.0

y\_max = 8.0

x\_min = 4.0

y\_min = 4.0

# 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', linewidth=2)

# Test lines

cohenSutherlandClip(3, 5, 5, 7)

plt.plot([3, 5], [5, 7], color='green', linewidth=1)

cohenSutherlandClip(8, 2, 8, 6)

plt.plot([8, 8], [2, 6], color='green', linewidth=1)

cohenSutherlandClip(1, 5, 4, 1)

plt.plot([1, 5], [4, 1], color='green', linewidth=1)

# Set plot limits and aspect ratio

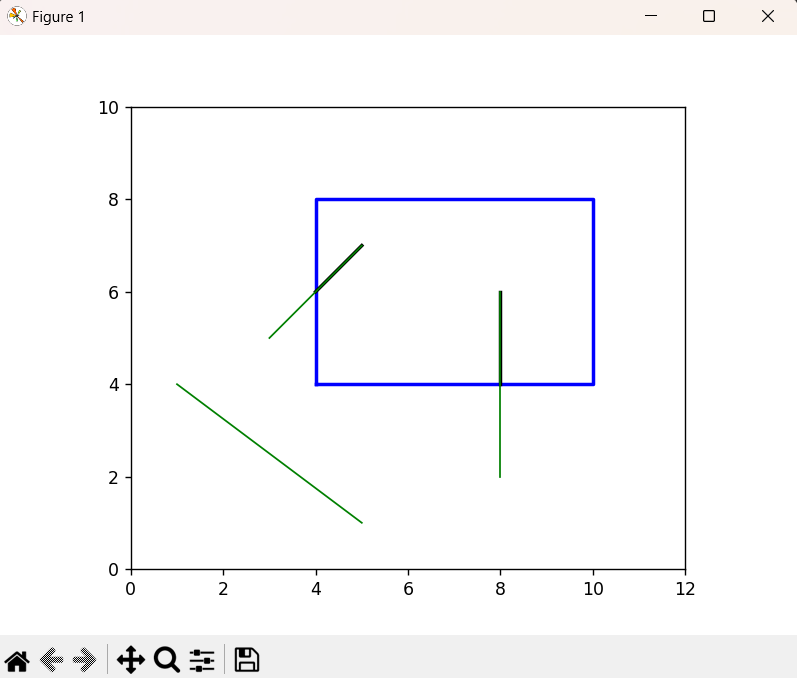
plt.xlim(0, 12)

plt.ylim(0, 10)

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

plt.show()

**OUTPUT:-**



**Ques4:** Write a program to clip a polygon using Sutherland Hodgeman algorithm.

**CODE:-**

def inside(point, edge):

    # Check if a point is inside or outside an edge

    x, y = point

    x1, y1, x2, y2 = edge

    if (x2 - x1) \* (y - y1) > (y2 - y1) \* (x - x1):

        return True  # Point is inside the edge

    return False  # Point is outside the edge

def intersect(point1, point2, edge):

    # Calculate intersection point of a line segment with an edge

    x1, y1 = point1

    x2, y2 = point2

    x3, y3, x4, y4 = edge

    dx1, dy1 = x2 - x1, y2 - y1

    dx2, dy2 = x4 - x3, y4 - y3

    denominator = dx1 \* dy2 - dy1 \* dx2

    if denominator == 0:

        return None  # Lines are parallel or coincident

    t = ((x1 - x3) \* dy2 - (y1 - y3) \* dx2) / denominator

    if 0 <= t <= 1:

        ix, iy = x1 + t \* dx1, y1 + t \* dy1

        return ix, iy  # Intersection point

    return None  # Intersection point is outside the segment

def clip\_polygon(polygon, window):

    # Clip a polygon against a rectangular window

    output = []

    for edge in window:

        if len(edge) != 4:

            raise ValueError("Window edges must be defined by four coordinates (x1, y1, x2, y2).")

        input\_polygon = output.copy() if output else polygon

        output = []

        prev\_point = input\_polygon[-1]

        for point in input\_polygon:

            if len(point) != 2:

                raise ValueError("Polygon points must be defined by two coordinates (x, y).")

            if inside(point, edge):

                if not inside(prev\_point, edge):

                    intersection = intersect(prev\_point, point, edge)

                    if intersection:

                        output.append(intersection)

                output.append(point)

            elif inside(prev\_point, edge):

                intersection = intersect(prev\_point, point, edge)

                if intersection:

                    output.append(intersection)

            prev\_point = point

    return output

polygon = [(50, 150), (200, 50), (350, 150), (350, 300), (250, 300), (200, 250), (150, 300), (50, 300)]

window = [(100, 100, 300, 200)]  # Rectangle window (x1, y1, x2, y2)

try:

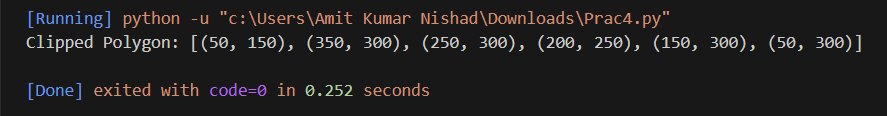
    clipped\_polygon = clip\_polygon(polygon, window)

    print("Clipped Polygon:", clipped\_polygon)

except ValueError as e:

    print("Error:", e)

**OUTPUT:-**



**Ques5:** Write a program to fill a polygon using Scan line fill algorithm.

**CODE:-**

import numpy as np

import matplotlib.pyplot as plt

def scanline\_fill(points):

    # Find the min and max y-coordinates

    ymin = int(min(points[:,1]))

    ymax = int(max(points[:,1]))

    # Initialize an array to store the x-coordinates of the intersections

    # between the scanline and the polygon edges

    x\_intersections = np.zeros((len(points),))

    # Iterate over each scanline

    for y in range(ymin, ymax+1):

        # Find the edges that intersect the scanline

        j = 0

        for i in range(len(points)):

            if i == len(points) - 1:

                k = 0

            else:

                k = i + 1

            if (points[i][1] <= y and points[k][1] > y) or (points[k][1] <= y and points[i][1] > y):

                x\_intersections[j] = int(points[i][0] + (y - points[i][1]) / (points[k][1] - points[i][1]) \* (points[k][0] - points[i][0]))

                j += 1

        # Sort the intersections by x-coordinate

        x\_intersections = np.sort(x\_intersections[:j])

        # Fill the scanline between pairs of intersections

        for i in range(0, len(x\_intersections), 2):

            plt.plot([x\_intersections[i], x\_intersections[i+1]], [y, y], color='black')

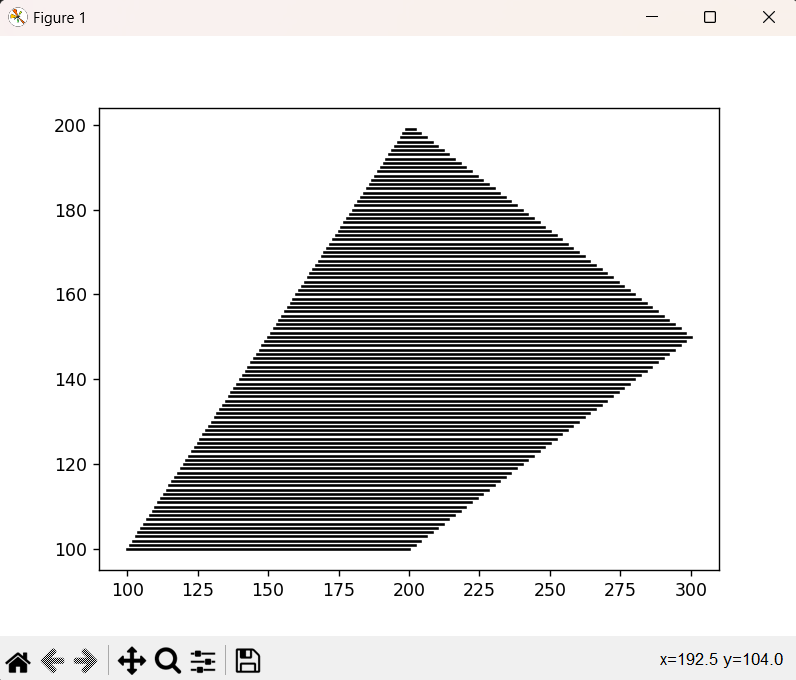
    plt.show()

#example usage

points = np.array([(100, 100), (200, 200), (300, 150), (200, 100)])

scanline\_fill(points)

**OUTPUT:-**



**Ques6:** Write a program to apply various 2D transformations on a 2D object (use homogenous Coordinates).

**CODE:-**

import numpy as np

import matplotlib.pyplot as plt

# Define the vertices of the original triangle

triangle = np.array([[0, 0], [0, 2], [2, 0], [0,0]])

# Define the translation vector

translation = np.array([2, 1])

# Translate the triangle by adding the translation vector to each vertex

new\_triangle = triangle + translation

# Plot the original and translated triangles

plt.plot(triangle[:,0], triangle[:,1], 'bo-', label='Original Triangle')

plt.plot(new\_triangle[:,0], new\_triangle[:,1], 'go-', label='Translated Triangle')

plt.axis('equal')

plt.legend()

plt.show()

#ROTATION

# Define the vertices of the original triangle

triangle = np.array([[0, 0], [0, 2], [2, 0], [0,0]])

# Define the rotation angle in degrees

angle\_deg = 45

# Convert the rotation angle to radians

angle\_rad = np.deg2rad(angle\_deg)

# Define the rotation matrix

rotation = np.array([[np.cos(angle\_rad), -np.sin(angle\_rad)],

                     [np.sin(angle\_rad), np.cos(angle\_rad)]])

# Rotate the triangle by multiplying each vertex by the rotation matrix

new\_triangle = np.dot(triangle, rotation)

# Plot the original and rotated triangles

plt.plot(triangle[:,0], triangle[:,1], 'yo-', label='Original Triangle')

plt.plot(new\_triangle[:,0], new\_triangle[:,1], 'go-', label='Rotated Triangle')

plt.axis('equal')

plt.legend()

plt.show()

#SCALING

# Define the vertices of the original triangle

triangle = np.array([[0, 0], [0, 2], [2, 0], [0,0]])

# Define the scaling factor

scale\_factor = 2

# Define the scaling matrix

scaling = np.array([[scale\_factor, 0],

                    [0, scale\_factor]])

# Scale the triangle by multiplying each vertex by the scaling matrix

new\_triangle = np.dot(triangle, scaling)

# Plot the original and scaled triangles

plt.plot(triangle[:,0], triangle[:,1], 'ro-', label='Original Triangle')

plt.plot(new\_triangle[:,0], new\_triangle[:,1], 'go-', label='Scaled Triangle')

plt.axis('equal')

plt.legend()

plt.show()

#SHEARING

# Define the vertices of the original triangle

triangle = np.array([[0, 0], [0, 2], [2, 0], [0,0]])

# Define the shearing factor

shear\_factor = 2

# Define the shearing matrix

shearing = np.array([[1, shear\_factor],

                     [0, 1]])

# Shear the triangle by multiplying each vertex by the shearing matrix

new\_triangle = np.dot(triangle, shearing)

# Plot the original and sheared triangles

plt.plot(triangle[:,0], triangle[:,1], 'bo-', label='Original Triangle')

plt.plot(new\_triangle[:,0], new\_triangle[:,1], 'go-', label='Sheared Triangle')

plt.axis('equal')

plt.legend()

plt.show()

#REFLECTION

# Define the vertices of the original triangle

triangle = np.array([[0, 0], [0, 2], [2, 0], [0,0]])

# Define the reflection axis

reflection\_axis = np.array([[1, 0], [0, -1]])

# Reflect the triangle by multiplying each vertex by the reflection axis

new\_triangle = np.dot(triangle, reflection\_axis)

# Plot the original and reflected triangles

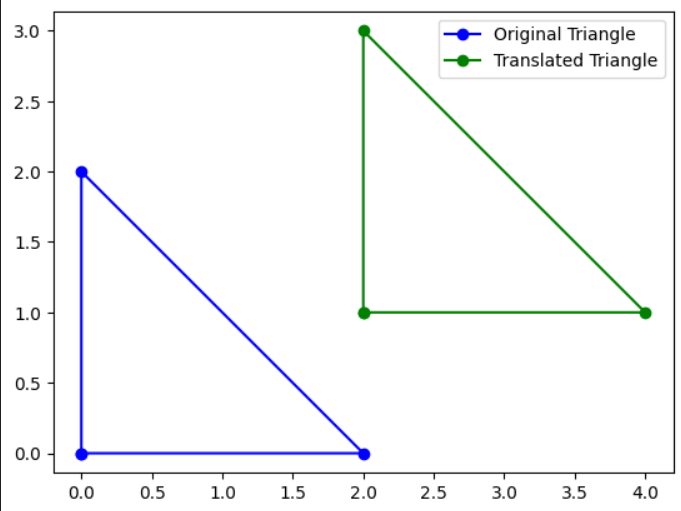
plt.plot(triangle[:,0], triangle[:,1], 'bo-', label='Original Triangle')

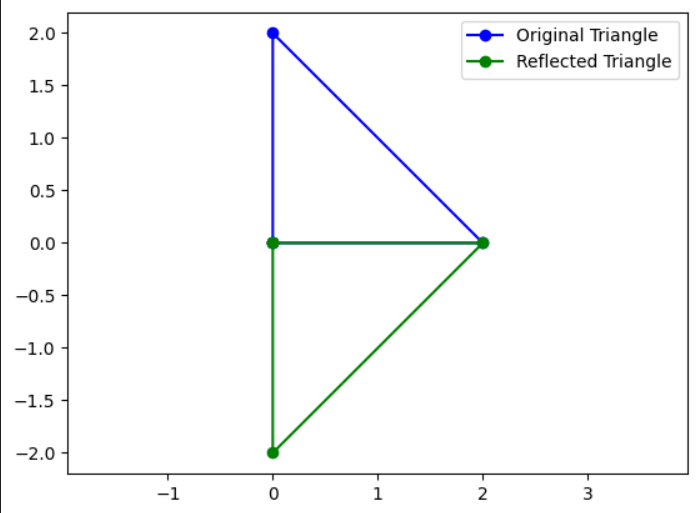
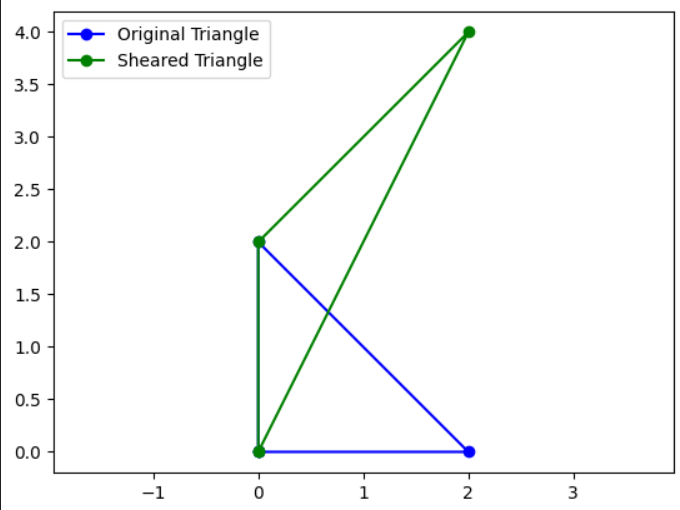
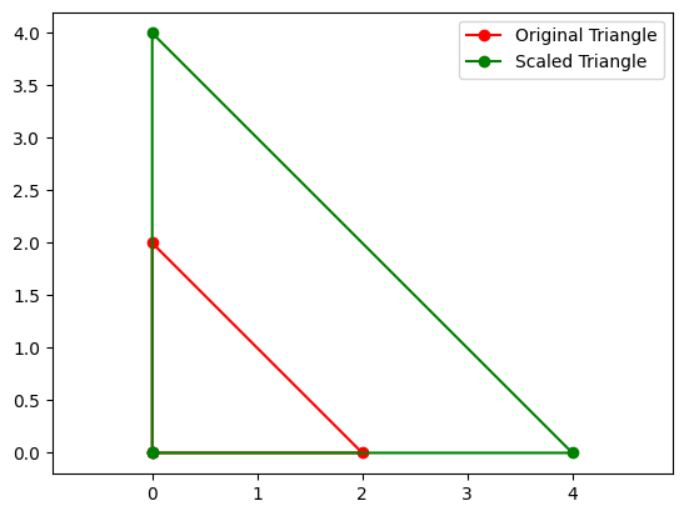
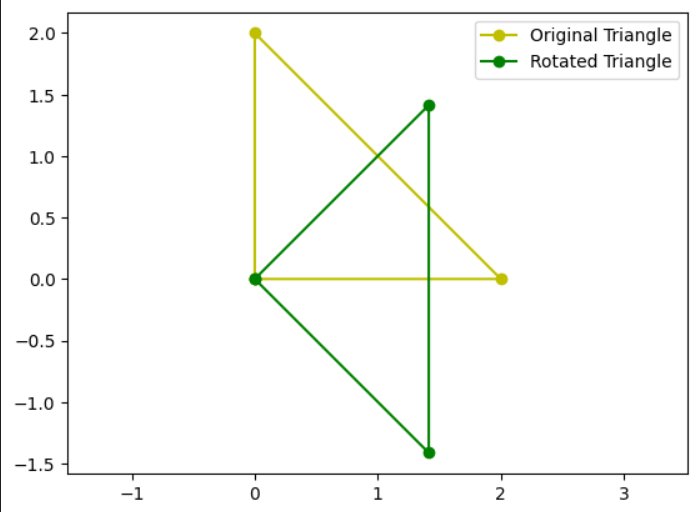
plt.plot(new\_triangle[:,0], new\_triangle[:,1], 'go-', label='Reflected Triangle')

plt.axis('equal')

plt.legend()

plt.show()

**OUTPUT:-** 



**Ques7:** Write a program to apply various 3D transformations on a 3D object and then apply parallel and perspective projection on it.

**CODE:-**

import numpy as np

import matplotlib.pyplot as plt

from mpl\_toolkits.mplot3d import Axes3D

# Define the 3D cube vertices

cube\_vertices = 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]

])

# Define the transformation matrices

scaling\_matrix = np.array([

    [2, 0, 0],

    [0, 2, 0],

    [0, 0, 2]

])

rotation\_matrix = np.array([

    [np.cos(np.pi/4), -np.sin(np.pi/4), 0],

    [np.sin(np.pi/4), np.cos(np.pi/4), 0],

    [0, 0, 1]

])

translation\_matrix = np.array([

    [1, 0, 0, 4],

    [0, 1, 0, 3],

    [0, 0, 1, 1],

    [0, 0, 0, 1]

])

# Apply the transformations to the cube vertices

transformed\_cube\_vertices = cube\_vertices.dot(scaling\_matrix).dot(rotation\_matrix) + translation\_matrix[:3,3]

# Perform parallel projection

parallel\_projection\_matrix = np.array([

    [1, 0, 0],

    [0, 1, 0]

])

projected\_cube\_vertices = transformed\_cube\_vertices[:, :2].dot(parallel\_projection\_matrix)

# Perform perspective projection

focal\_length = 5

perspective\_projection\_matrix = np.array([

    [focal\_length, 0, 0],

    [0, focal\_length, 0]

])

projected\_cube\_vertices\_perspective = transformed\_cube\_vertices[:, :2].dot(perspective\_projection\_matrix) / transformed\_cube\_vertices[:, 2:]

# Plot the original and transformed 3D cube

fig = plt.figure()

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

ax.scatter(cube\_vertices[:, 0], cube\_vertices[:, 1], cube\_vertices[:, 2], color='blue')

ax.scatter(transformed\_cube\_vertices[:, 0], transformed\_cube\_vertices[:, 1], transformed\_cube\_vertices[:, 2], color='red')

plt.show()

# Plot the parallel projection

plt.scatter(projected\_cube\_vertices[:, 0], projected\_cube\_vertices[:, 1])

plt.title('Parallel Projection')

plt.show()

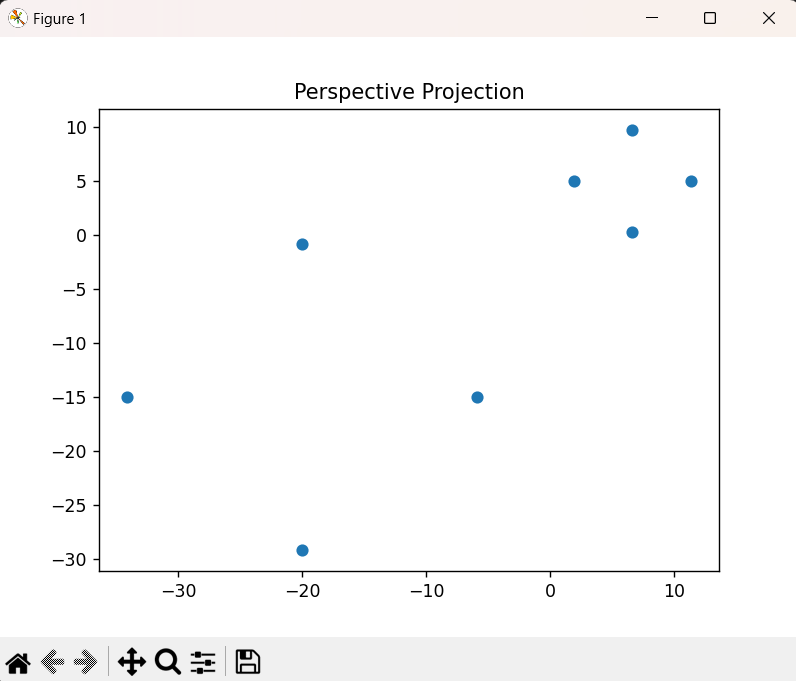
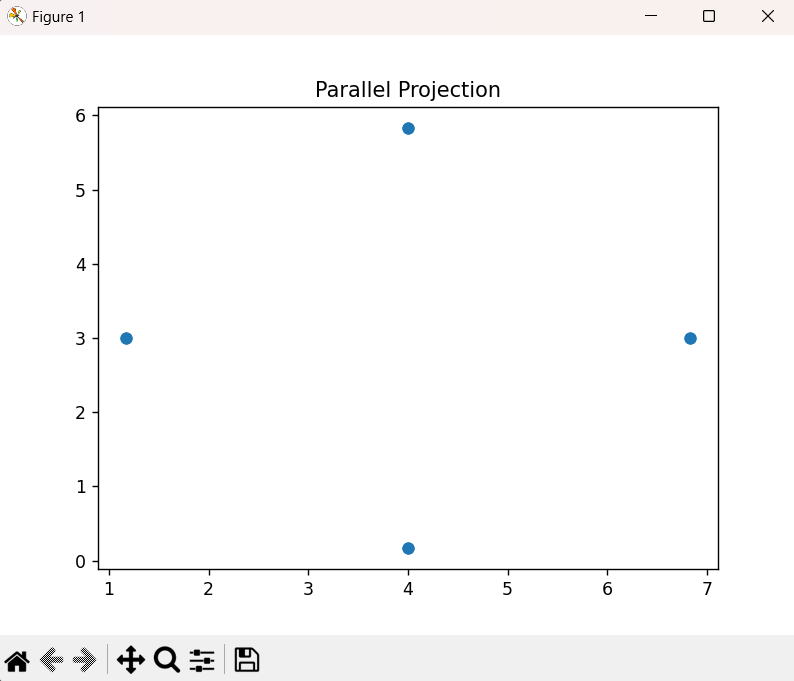
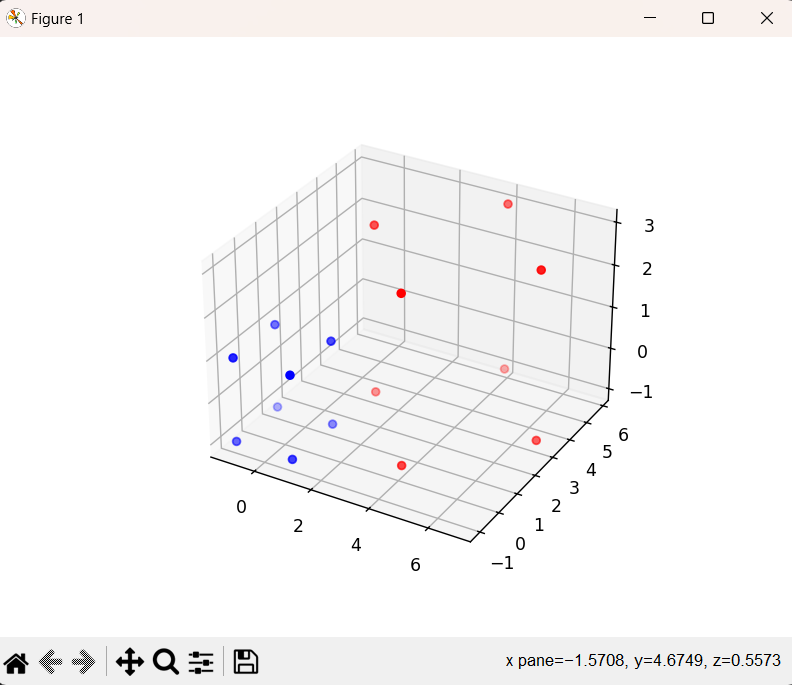
# Plot the perspective projection

plt.scatter(projected\_cube\_vertices\_perspective[:, 0], projected\_cube\_vertices\_perspective[:, 1])

plt.title('Perspective Projection')

plt.show()

**OUTPUT:-**



**Ques8:** Write a program to draw Hermite / Bezier curve.

**CODE:-**

**Hermite Curve-**

import numpy as np

import matplotlib.pyplot as plt

def hermite\_curve(P0, P1, P2, P3):

    #

    def H1(t):

        return 2\*t\*\*3 - 3\*t\*\*2 + 1

    def H2(t):

        return -2\*t\*\*3 + 3\*t\*\*2

    def H3(t):

        return t\*\*3 - 2\*t\*\*2 + t

    def H4(t):

        return t\*\*3 - t\*\*2

    # Create a range of values for t

    t\_values = np.linspace(0.0, 1.0, 100)

    # Evaluate the Hermite curve function for each value of t

    curve\_points = np.array([P0 \* H1(t) + P3 \* H2(t) + P1 \* H3(t) + P2 \* H4(t)

                             for t in t\_values])

    # Plot the Hermite curve

    plt.plot(curve\_points[:,0], curve\_points[:,1], 'b-', label='Hermite Curve')

    plt.plot([P0[0], P1[0]], [P0[1], P1[1]], 'ro-', label='Control Points')

    plt.plot([P2[0], P3[0]], [P2[1], P3[1]], 'ro-')

    plt.legend()

    plt.show()

# Example usage:

P0 = np.array([1, 1])

P1 = np.array([1, 2])

P2 = np.array([3, 4])

P3 = np.array([5, 3])

hermite\_curve(P0, P1, P2, P3)

**Bezier Curve-**

import matplotlib.pyplot as plt

import numpy as np

import math

def bezier\_curve(control\_points):

    #

    control\_points=np.array(control\_points)

    def B(t):

        n = len(control\_points)-1

        return np.sum([control\_points[i] \* math.comb(n,i) \* (t\*\*i) \*(1-t)\*\*(n-i)

                       for i in range(n+1)], axis=0)

    # Create a range of values for t

    t\_values = np.linspace(0.0, 1.0, 100)

    # Evaluate the Bezier curve function for each value of t

    curve\_points = np.array([B(t) for t in t\_values])

    # Plot the Bezier curve

    plt.plot(curve\_points[:,0], curve\_points[:,1], 'b-', label='Bezier Curve')

    plt.plot(control\_points[:,0], control\_points[:,1], 'ro-', label='Control Points')

    plt.legend()

    plt.show()

# Example usage:

control\_points = [(1,1), (2,3), (4,4), (6,1)]

bezier\_curve(control\_points)

**OUTPUT:-**

